

Hacking the Kinect

Created by lady ada



Last updated on 2018-08-22 03:30:10 PM UTC

Guide Contents

Guide Contents	2
Overview	3
Verify the VID & PID	4
Determine the Descriptors	5
Making a Driver	9
Installing Python & PyUSB	11
Fuzzing	12
USB Analyzer	14
Lookin' at Logs	15
Command #1 & 2 - LED blinky!	22
Command #3 & 4 - Let's move!	23
Bonus Accelerometer!	24
More Kinect Information	27

Overview

Everyone has seen the [Xbox 360 Kinect hacked in a matter of days after our "open source driver" bounty \(https://adafru.it/aLQ\)](https://adafru.it/aLQ) - here's how we helped the winner and here's how you can reverse engineer USB devices as well!

USB is a very complex protocol, much more complicated than Serial or Parallel, SPI and even I2C. USB uses only two wires but they are not used as 'receive' and 'transmit' like serial. Rather, data is bidirectional and differential - that is the data sent depends on the *difference* in voltage between the two data lines **D+** and **D-**. If you want to do more USB hacking, you'll need to read [Jan Axelson's USB Complete books \(https://adafru.it/eIG\)](https://adafru.it/eIG), they're easy to follow and discuss USB in both depth and breadth.

USB is also very structured. This is good for reverse engineering because it means that at least the format of packets is agreed upon and you won't have to deal with check-sums. The bad news is it means you have to have software assistance to decode the complex packet structure. The good news is that every computer now made has a USB host core, that does a lot of the tough work for you, and there are many software libraries to assist.

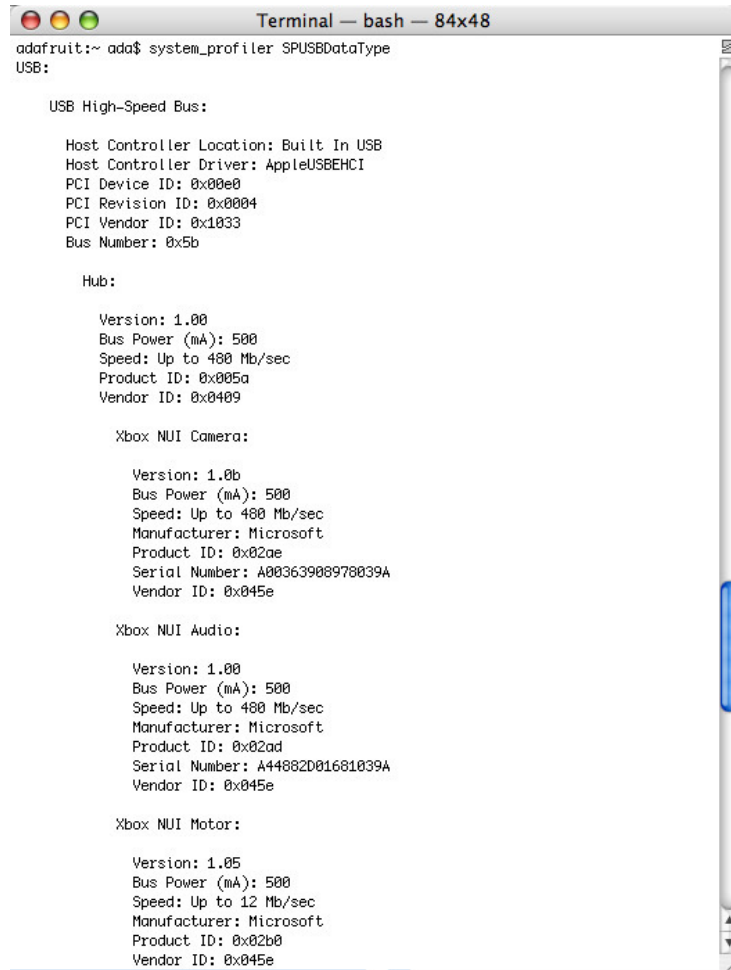
Today we're going to be reverse engineering the Xbox Kinect Motor, one part of the Kinect device.



Verify the VID & PID

The first place to start is to see what devices and "interfaces" or "configurations" are available for the USB device. The nicest way to do this is to use **lsusb** (Linux) or **system_profiler** (Mac) which is a "list usb" program available for Linux and mac. Sadly, it does not exist for windows, so find a mac or linux computer or friend, you'll only need it for a minute! [[edit: a reader has notified us that http://www.nirsoft.net/utis/usb_devices_view.html (<https://adafru.it/w4e>) may do the trick!]]

For linux, run **lsusb -vv** (ultra verbose) for Mac, run **system_profiler SPUSBDataType**



```
Terminal — bash — 84x48
adafruit:~ ada$ system_profiler SPUSBDataType
USB:

  USB High-Speed Bus:

    Host Controller Location: Built In USB
    Host Controller Driver: AppleUSBECFI
    PCI Device ID: 0x00e0
    PCI Revision ID: 0x0004
    PCI Vendor ID: 0x1033
    Bus Number: 0x5b

  Hub:

    Version: 1.00
    Bus Power (mA): 500
    Speed: Up to 480 Mb/sec
    Product ID: 0x005a
    Vendor ID: 0x0409

  Xbox NUI Camera:

    Version: 1.0b
    Bus Power (mA): 500
    Speed: Up to 480 Mb/sec
    Manufacturer: Microsoft
    Product ID: 0x02ae
    Serial Number: A00363908978039A
    Vendor ID: 0x045e

  Xbox NUI Audio:

    Version: 1.00
    Bus Power (mA): 500
    Speed: Up to 480 Mb/sec
    Manufacturer: Microsoft
    Product ID: 0x02ad
    Serial Number: A44882D01681039A
    Vendor ID: 0x045e

  Xbox NUI Motor:

    Version: 1.05
    Bus Power (mA): 500
    Speed: Up to 12 Mb/sec
    Manufacturer: Microsoft
    Product ID: 0x02b0
    Vendor ID: 0x045e
```

There's a bunch more stuff like USB keys and such installed but this is a good starting point. Note that the Kinect is actually 4 USB devices - a hub, a camera, a microphone (audio) and a motor. The hub is just an easy way for the device to combine three separate chips into a single cable. We'll be investigating the **Xbox NUI Motor** since its the simplest. Note the **Vendor ID = 0x045e** and **Product ID = 0x02b0**. Every type USB device must have a unique VID and PID. The VID is the manufacturer. In this case, **0x045e** is the VID for Microsoft. All Microsoft products will have that VID. Each product has a different PID, so all Kinect Motors use PID **0x02b0** this doesn't differ between two Kinects, they'll both have the same PID. The VID/PID are used as a way to have the proper driver find the product. Its a lot better than serial COM ports because COM ports change names but VID/PID are burned into the device firmware.

Determine the Descriptors

The next best thing to do after you've determined the VID/PID is to identify the **descriptor** of the device. A descriptor is a sort of 'menu' of what the device can do and how it likes to transfer data. In general, each device has one descriptor. *Sometimes* a device has more than one descriptor and you can choose which one you want but its not terribly common so we're just going to ignore it. A fantastic way to get the descriptor without having to write any software is to run **lsusb -vv** on a linux computer. (Try the "USB Prober" tool from Apple for Mac OS X or [USBView on Windows \(https://adafru.it/eup\)](https://adafru.it/eup))

Here is the output of **lsusb** for the NUI Motor

```
Device Descriptor:
  bLength                18
  bDescriptorType        1
  bcdUSB                 2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass        0
  bDeviceProtocol        0
  bMaxPacketSize0       64
  idVendor               0x045e Microsoft Corp.
  idProduct              0x02b0
  bcdDevice              1.05
  iManufacturer          1 Microsoft
  iProduct               2 Xbox NUI Motor
  iSerial                0
  bNumConfigurations     1
Configuration Descriptor:
  bLength                9
  bDescriptorType        2
  wTotalLength           18
  bNumInterfaces         1
  bConfigurationValue    1
  iConfiguration         0
  bmAttributes           0xc0
    Self Powered
  MaxPower               100mA
Interface Descriptor:
  bLength                9
  bDescriptorType        4
  bInterfaceNumber       0
  bAlternateSetting      0
  bNumEndpoints          0
  bInterfaceClass        255 Vendor Specific Class
  bInterfaceSubClass     0
  bInterfaceProtocol     0
  iInterface             0
Device Status:          0x0000
  (Bus Powered)
```

Let's see what we've got. You can see the VID and PID up there. Next we'll look at **bNumConfigurations** (how many different descriptors we have) and lucky for us the number is **1**. Next, look at the **Interface Descriptor** in particular, **bNumEndpoints** which is 0. This means there are no Endpoints.

Endpoints are a type of USB 'data pipe' - there are 4 kinds:

- **Bulk Endpoints** are for transferring a lot of data, like a disk drive. It's OK if it takes a little longer but we want big packets. This endpoint goes only in one direction (so to read and write you'd want two)
- **Interrupt Endpoints** are for transferring tiny amounts of data very quickly, like for a USB mouse. In this case, the device has to be responsive so we want fast movement. This endpoint goes only in one direction
- **Isochronous Endpoints** are for transferring a fair amount of data where the data must show up at the same time and if it can't it should just be dropped. This is for stuff like Audio and Video where timing is key. This endpoint goes only in one direction (so bidirectional audio for headphone and mic would have two EPs)
- **Control Endpoints** are this weird not-quite-an-Endpoint Endpoint. They are used to transfer small amounts of data to say turn a device on or off. They're very 'cheap' to develop, and every device has one even if its not mentioned.

For example, a serial port may have two Interrupt endpoints for transferring data in and out and then a control endpoint for setting the baud rate.

For more details we really do suggest reading everything at [janaxelson.com \(https://adafru.it/e1G\)](https://adafru.it/e1G) about USB as it's complex.

This motor device has no Endpoints, but that doesn't mean you can't communicate with it. It just means it only uses a bidirectional Control Endpoint. This isn't surprising, motors are slow and don't require a lot of data to control.

Contrast this to the Video/Camera device:

```

Device Descriptor:
  bLength                18
  bDescriptorType        1
  bcdUSB                 2.00
  bDeviceClass           0 (Defined at Interface level)
  bDeviceSubClass        0
  bDeviceProtocol        0
  bMaxPacketSize0       64
  idVendor               0x045e Microsoft Corp.
  idProduct              0x02ae
  bcdDevice              1.0b
  iManufacturer         2 Microsoft
  iProduct               1 Xbox NUI Camera
  iSerial                3 A00366A08793039A
  bNumConfigurations    1

```

Configuration Descriptor:

```

  bLength                9
  bDescriptorType        2
  wTotalLength          32
  bNumInterfaces        1
  bConfigurationValue    1
  iConfiguration        0
  bmAttributes           0xc0
    Self Powered
  MaxPower              16mA

```

Interface Descriptor:

```

  bLength                9
  bDescriptorType        4
  bInterfaceNumber      0
  bAlternateSetting     0
  bNumEndpoints        2
  bInterfaceClass       255 Vendor Specific Class
  bInterfaceSubClass    255 Vendor Specific Subclass
  bInterfaceProtocol    255 Vendor Specific Protocol
  iInterface            0

```

Endpoint Descriptor:

```

  bLength                7
  bDescriptorType        5
  bEndpointAddress      0x81 EP 1 IN
  bmAttributes          1
    Transfer Type        Isochronous
    Synch Type           None
    Usage Type           Data
  wMaxPacketSize        0x0bc0 2x 960 bytes
  bInterval             1

```

Endpoint Descriptor:

```

  bLength                7
  bDescriptorType        5
  bEndpointAddress      0x82 EP 2 IN
  bmAttributes          1
    Transfer Type        Isochronous
    Synch Type           None
    Usage Type           Data
  wMaxPacketSize        0x0bc0 2x 960 bytes
  bInterval             1

```

Device Qualifier (for other device speed):

```

  bLength                10
  bDescriptorType        6
  bcdUSB                 2.00
  bDeviceClass           0 (Defined at Interface level)
  bDeviceSubClass        0
  bDeviceProtocol        0
  bMaxPacketSize0       64
  bNumConfigurations    1

```

```

Device Status:      0x0001
  Self Powered

```

This device has two Isochronous endpoints **both** of which are **IN** type (data going **IN**to the computer). This makes sense: the Kinect has a IR depth camera and a normal VGA camera. Two cameras, two Endpoints. Of course, there is also a Control endpoint not mentioned here, the Control endpoint could be used to set stuff like aperture, gamma correction, any sort of built-in filter, etc.

Making a Driver

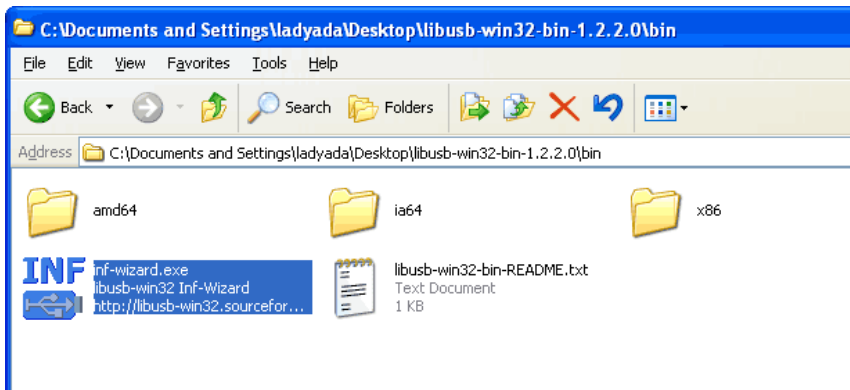
OK so back to our motor. We are ready to start sending data to it via the Control endpoint. For Mac and Linux type computers, a driver isn't necessary to send or receive data directly via USB.

For windows, however, there must be some sort of driver to 'grab' the device for us. Usually drivers are complex and have like, interfaces that plug into the operating system. Like the cameras would show up as a camera device, the microphones as an audio device. We're not quite ready for a detailed driver, what we'll do is make a 'shell driver' which has no operating system capabilities but does let us send commands to it from software.

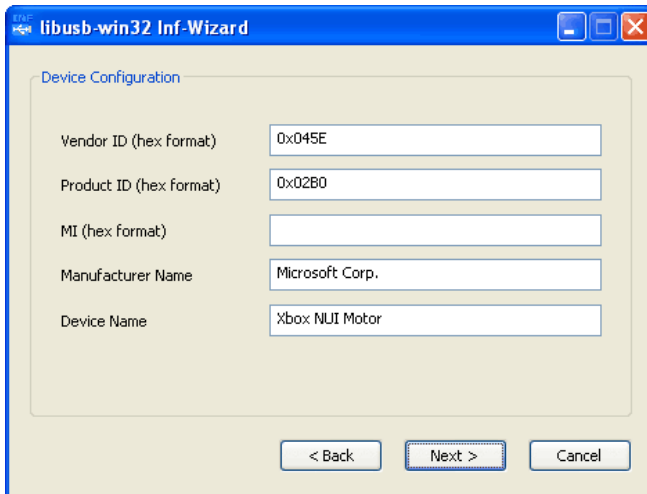
Again, Mac/Linux people have this built into the OS kernel so skip this part if you don't use windows.

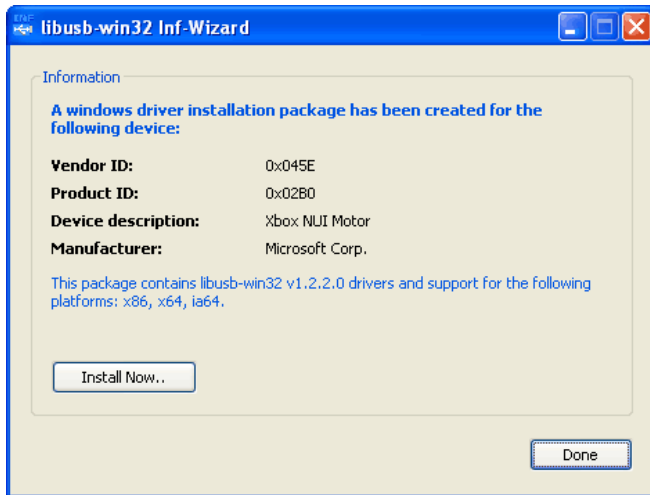
For our shell, we'll use **libusb** a USB library, which is available for windows as **libusb-win32** (<https://adafru.it/aLS>) go there and download it.

We'll run the **inf-wizard** (which will make our driver shell)

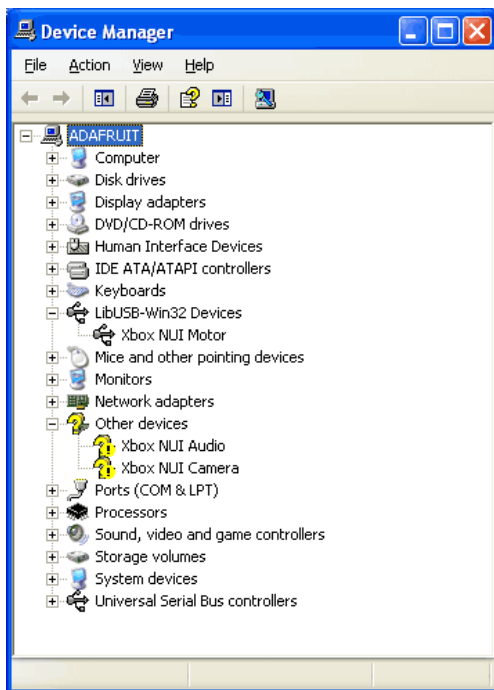


The important part is entering in the matching VID and PID we found before.





Now when you plug in the Kinect, it will attach itself the the LibUSB-win32 device driver.



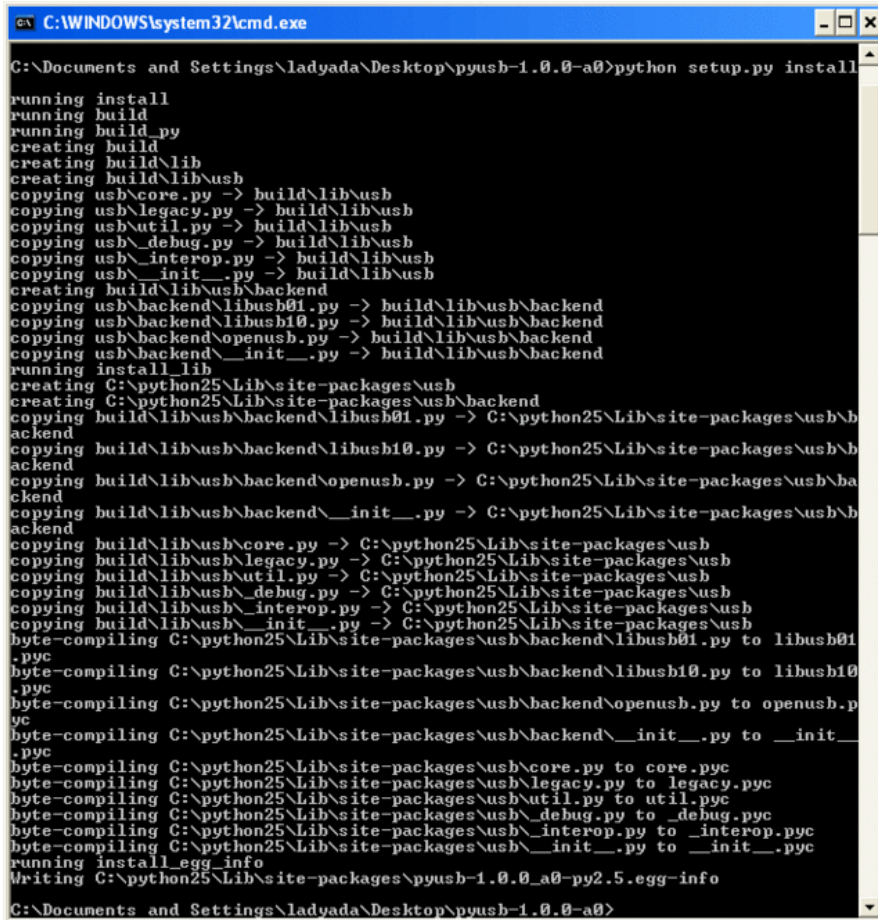
We didn't make matching drivers for the audio or camera so those are still driver-less.

Installing Python & PyUSB

Now we need to start sending commands to this USB device! The fastest and easiest way we know to do this is to use LibUSB with a scripting language such as Python. There are LibUSB bindings for C and C++ and Perl but I happen to like Python so follow along!

If you don't have python installed, do that now. (<https://adafru.it/aJA>)

Next up, install PyUSB (<https://adafru.it/aLT>) by downloading it and running `python setup.py install` in the expanded directory



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\ladyada\Desktop\pyusb-1.0.0-a0>python setup.py install
running install
running build
running build_py
creating build
creating build\lib
creating build\lib\usb
copying usb\core.py -> build\lib\usb
copying usb\legacy.py -> build\lib\usb
copying usb\util.py -> build\lib\usb
copying usb\_debug.py -> build\lib\usb
copying usb\_interop.py -> build\lib\usb
copying usb\_init_.py -> build\lib\usb
creating build\lib\usb\backend
copying usb\backend\libusb01.py -> build\lib\usb\backend
copying usb\backend\libusb10.py -> build\lib\usb\backend
copying usb\backend\openusb.py -> build\lib\usb\backend
copying usb\backend\_init_.py -> build\lib\usb\backend
running install_lib
creating C:\python25\Lib\site-packages\usb
creating C:\python25\Lib\site-packages\usb\backend
copying build\lib\usb\backend\libusb01.py -> C:\python25\Lib\site-packages\usb\ba
ackend
copying build\lib\usb\backend\libusb10.py -> C:\python25\Lib\site-packages\usb\ba
ackend
copying build\lib\usb\backend\openusb.py -> C:\python25\Lib\site-packages\usb\ba
ckend
copying build\lib\usb\backend\_init_.py -> C:\python25\Lib\site-packages\usb\ba
ckend
copying build\lib\usb\core.py -> C:\python25\Lib\site-packages\usb
copying build\lib\usb\legacy.py -> C:\python25\Lib\site-packages\usb
copying build\lib\usb\util.py -> C:\python25\Lib\site-packages\usb
copying build\lib\usb\_debug.py -> C:\python25\Lib\site-packages\usb
copying build\lib\usb\_interop.py -> C:\python25\Lib\site-packages\usb
copying build\lib\usb\_init_.py -> C:\python25\Lib\site-packages\usb
byte-compiling C:\python25\Lib\site-packages\usb\backend\libusb01.py to libusb01
.pyc
byte-compiling C:\python25\Lib\site-packages\usb\backend\libusb10.py to libusb10
.pyc
byte-compiling C:\python25\Lib\site-packages\usb\backend\openusb.py to openusb.p
yc
byte-compiling C:\python25\Lib\site-packages\usb\backend\_init_.py to _init_.p
yc
byte-compiling C:\python25\Lib\site-packages\usb\core.py to core.pyc
byte-compiling C:\python25\Lib\site-packages\usb\legacy.py to legacy.pyc
byte-compiling C:\python25\Lib\site-packages\usb\util.py to util.pyc
byte-compiling C:\python25\Lib\site-packages\usb\_debug.py to _debug.pyc
byte-compiling C:\python25\Lib\site-packages\usb\_interop.py to _interop.pyc
byte-compiling C:\python25\Lib\site-packages\usb\_init_.py to _init_.pyc
running install_egg_info
Writing C:\python25\Lib\site-packages\pyusb-1.0.0-a0-py2.5.egg-info
C:\Documents and Settings\ladyada\Desktop\pyusb-1.0.0-a0>
```

Fuzzing

Now we can use Python + LibUSB to send Control Endpoint packets with the command

`ctrl_transfer(bmRequestType, bmRequest, wValue, wIndex, nBytes)`

This command can do both sending and receiving depending on what **bmRequestType** says (input or output). Still, there is a lot of options here. To send the right command you need to know the **RequestType** and the right **Request** and the right **Value** as well as the **Index** and how many bytes to read or write.

If we were totally on our own, we would start by trying to read data from the device. This means we have to set the **RequestType** first

Direction	Type					Recipient	
D7	D6	D5	D4	D3	D2	D1	D0

For **bmRequestType** the value passed is very structured so that's not as hard to guess. (See lvr.com for more information (<https://adafru.it/aLU>))

- Bits 2, 3 and 4 are reserves so set them to 0.
- The direction is set by bit #7, 0 is a 'write' out to the device, 1 is a 'read' from the device
- The 'type' of message is two bits, 0 = Standard, 1 = Class, 2 = Vendor, 3 = Reserved. For many devices that are non-standard, you'll probably want 2 for vendor type. If it's a more standard type of device, like a camera or mic, try 0 or 1. 3 is unused
- The last two bits are used to determine the recipient for the message 0 = Device, 1 = Interface, 2 = Endpoint, 3 = Other. Go with 0 to start, you can try 2 if there are other endpoints

The safest thing to do is read data (no way to overwrite anything or configure) you can do that by sending packets with **0b11000000** (Read Vendor data from Device) = **0xC0**.

If I were to write a fuzzer, I'd start by setting **Index** to 0 and iterating through all the byte values (255 different values) of **bmRequest** and the first few hundred **wValues**. It's pretty safe to just read random data to a USB device. Start by reading one byte to see if anything shows up, then increase the value

```
import usb.core
import usb.util
import sys

# find our device
dev = usb.core.find(idVendor=0x045e, idProduct=0x02B0)

# was it found?
if dev is None:
    raise ValueError('Device not found')

# set the active configuration. With no arguments, the first
# configuration will be the active one
dev.set_configuration()

# Let's fuzz around!

# Lets start by Reading 1 byte from the Device using different Requests
# bRequest is a byte so there are 255 different values
for bRequest in range(255):
    try:
        ret = dev.ctrl_transfer(0xC0, bRequest, 0, 0, 1)
        print "bRequest ", bRequest
        print ret
    except:
        # failed to get data for this request
        pass
```

```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\ladyada\Desktop\kinect>python usbmotor1.py
bRequest 0
array('B', [128])
bRequest 5
array('B', [0])
bRequest 16
array('B', [34])
bRequest 50
array('B', [0])
bRequest 54
array('B', [87])
bRequest 64
array('B', [0])
bRequest 80
array('B', [0])
bRequest 112
array('B', [107])
C:\Documents and Settings\ladyada\Desktop\kinect>_
```

Looks like **Request** values 0, 5, 16, 50, 54, 64, 80 and 112 all return some sort of data. The rest had nothing to read

Next we'll try to read more data by changing the last argument to 100 bytes

```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\ladyada\Desktop\kinect>python usbmotor1.py
bRequest 0
array('B', [128])
bRequest 5
array('B', [0, 1, 0, 1, 7, 233, 0, 0])
bRequest 16
array('B', [34])
bRequest 50
array('B', [0, 0, 255, 182, 3, 40, 255, 247, 253, 0, 253, 254, 0, 0, 171, 0, 176, 4, 76, 4, 147, 252, 251, 252, 251, 252, 253, 253, 254, 254, 255, 228])
bRequest 54
array('B', [87, 241])
bRequest 64
array('B', [0])
bRequest 80
array('B', [0, 0, 1, 43, 0, 57, 0, 0, 110, 62, 0, 0, 1, 88, 0, 95, 0, 0, 255, 255])
C:\Documents and Settings\ladyada\Desktop\kinect>
```

OK lots of data, but what does it mean? This is where some guessing based on the device itself would come in handy. I'm terribly lazy though and if given an option to avoid a lot of guesswork, I'll take it!

USB Analyzer

Reverse-engineering the Kinect is a little easier since we have a known-working system (Xbox 360). Instead of guessing commands, we can just see what commands the Xbox sends and 'replay them'



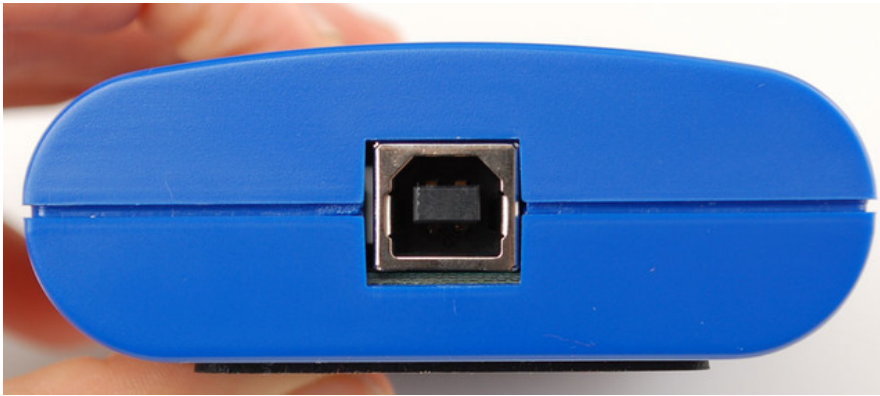
This requires being able to listen into those commands, however. With protocols such as SPI, Serial, Parallel and i2c, you can listen in with any logic analyzer or oscilloscope. USB is fast/complex enough to require its own kind of logic analyzer. The one we'll be using is called the [Beagle480 from TotalPhase](https://adafru.it/aLV). (<https://adafru.it/aLV>) This is the 'high speed' USB analyzer, which we splurged on. (For many devices, Low/Full speed is fast enough, and there's a lower cost analyzer available.)

The USB analyzer acts as a 'tap' that plugs in between the Xbox and the Kinect. A computer is connected as well. The computer receives all the data being transmitted into memory and logs it.

If you can connect the device to a computer, there are tons of free USB-packet-capture programs that don't require a hardware man-in-the-middle. In our case, the Kinect must connect to an Xbox and we can't run software on it, necessitating the Beagle

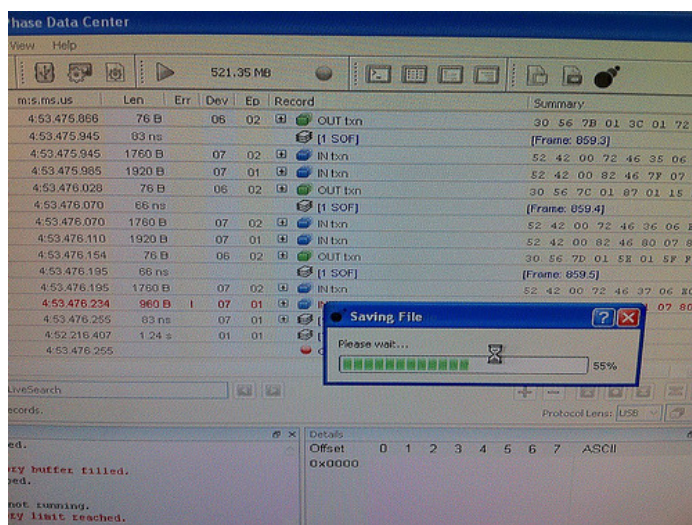


From left to right there is a **DIN** connector, **USB A** connector and **USB B** connector. The Xbox connects to the USB B and the Kinect connects to the USB A. The DIN connector is for other kinds of data sniffing (like SPI or i2c).



On the other side, a single B connector which goes to the listening computer

The best way we've found to get the right data is to make sure to get even the 'enumeration' (initialization) packets so plug in the listening computer and start up the software. Then plug in the other end to the devices you want to sniff.



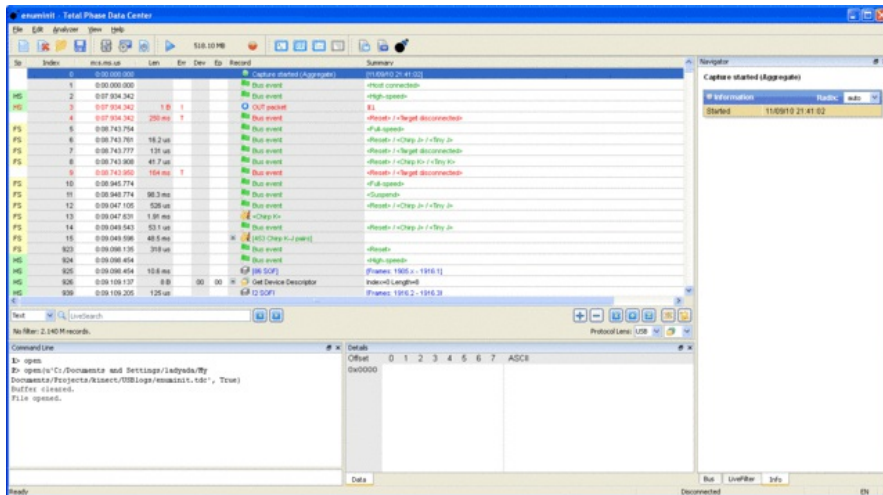
Lookin' at Logs

Since you probably don't have a USB analyzer, we have some logs that you can use to follow along with us. [Visit the GitHub repository](https://adafru.it/aLX) and click the ****Downloads**** button (<https://adafru.it/aLX>)

Make yourself a sandwich, its a big file!

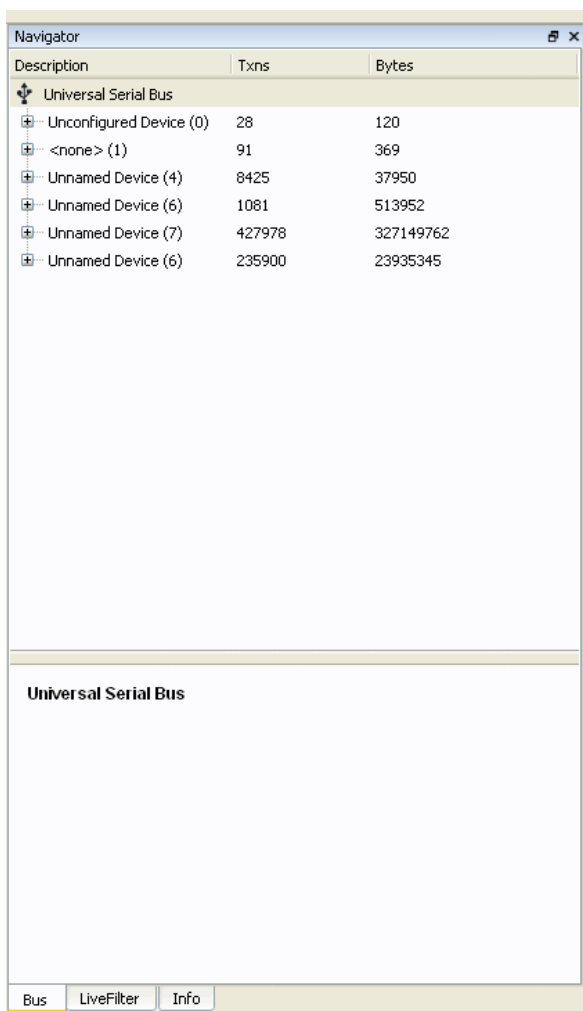
Also download the Beagle Data Center software (Mac/Win/Linux) (<https://adafru.it/aLV>) and install it

OK now that you've eaten, lets open up the `enuminit.tdc` file. This is the full enumeration and initialization.



Remember that when we log the data, there's a lot of it that we can then pare down!

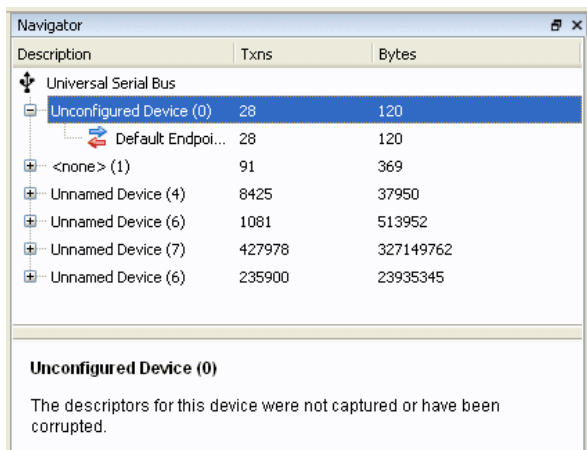
Let start by remembering that there are **four devices** (hub, camera, mic, motor) but we only need to listen to one (motor). Click on the **Bus** tab on the lower right



We have a few devices. Lets explore each one

If you click on **Unconfigured device (0)** you'll see that it was not captured. This is probably because I jiggled the cable when inserting it so

it started to create a device and then got disconnected. Its not important.

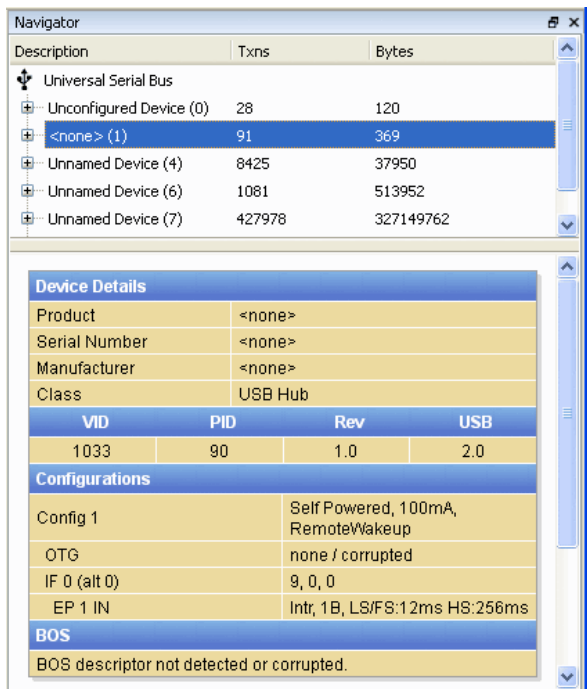


Description	Txns	Bytes
Universal Serial Bus		
Unconfigured Device (0)	28	120
Default Endpoi...	28	120
<none> (1)	91	369
Unnamed Device (4)	8425	37950
Unnamed Device (6)	1081	513952
Unnamed Device (7)	427978	327149762
Unnamed Device (6)	235900	23935345

Unconfigured Device (0)

The descriptors for this device were not captured or have been corrupted.

Click on **(1)** This device is a **Class** device type USB Hub. That's the internal hub. We can ignore this as well.



Description	Txns	Bytes
Universal Serial Bus		
Unconfigured Device (0)	28	120
<none> (1)	91	369
Unnamed Device (4)	8425	37950
Unnamed Device (6)	1081	513952
Unnamed Device (7)	427978	327149762

Device Details

Product	<none>
Serial Number	<none>
Manufacturer	<none>
Class	USB Hub

VID	PID	Rev	USB
1033	90	1.0	2.0

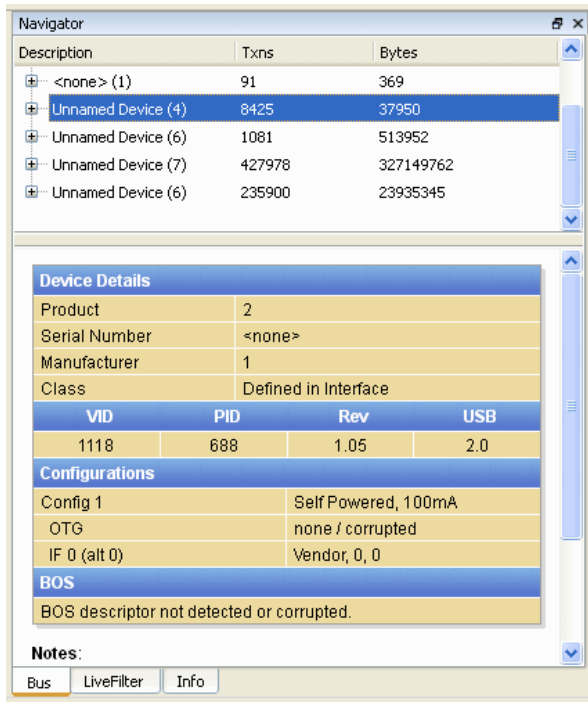
Configurations

Config 1	Self Powered, 100mA, RemoteWakeUp
OTG	none / corrupted
IF 0 (alt 0)	9, 0, 0
EP 1 IN	Intr, 1B, LS/FS:12ms HS:256ms

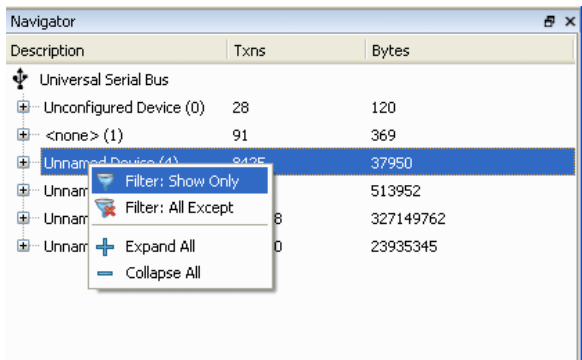
BOS

BOS descriptor not detected or corrupted.

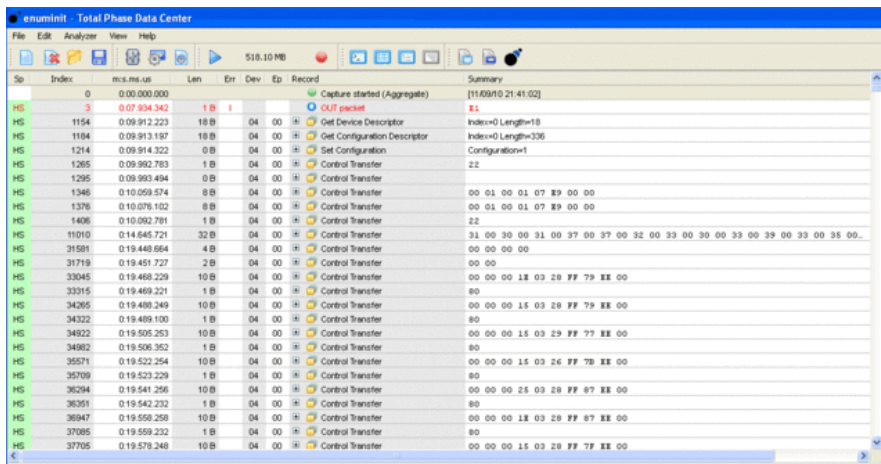
Device #4 has a PID of 688, that's in decimal. If we convert it to hex we get **0x02b0** - this is the Motor device!



Now we can filter so that only this device's logs are shown.



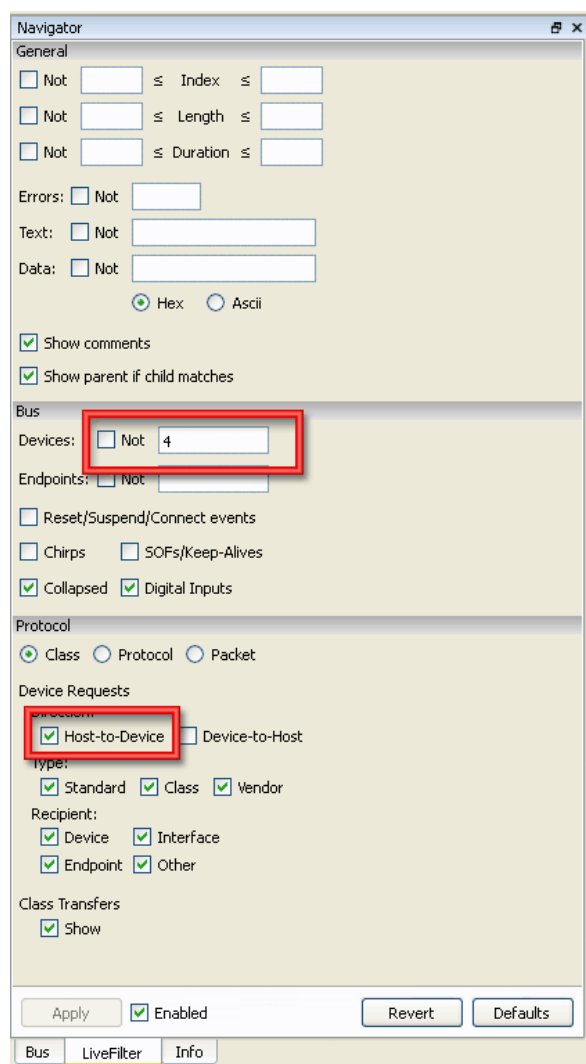
Our log screen is much shorter now.



You can see that there's some initialization and then just two repeating motifs: a 1 byte message alternated with a 10 byte message.

For the motor to move according to the xbox's wishes, there must be some command sent from the xbox to the kinect. Lets filter some

more to see just commands sent to the device



Go to the LiveFilter and select Host-to-Device.

Sp	Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
	0	0:00.000.000					Capture started (Aggregate)	[11.09/10 21:41:02]
HS	1214	0:09.914.322	0B		04	00	Set Configuration	Configuration=1
HS	1295	0:09.993.494	0B		04	00	Control Transfer	
HS	922558	0:34.505.982	0B		04	00	Control Transfer	
HS	952254	0:34.774.137	0B		04	00	Control Transfer	
HS	1237129	0:37.341.197	0B		04	00	Control Transfer	
	2140385	0:45.487.878					Capture stopped	[11.09/10 21:41:49]

Now we've really pared it down. There are only **four** commands sent to the kinect motor, since the motor moves during initialization we can just try each one. Lets look at each command

Command 1 has a **bRequest** of 0x06 and a **wValue** of 4, the **wLength** is 0 which means no data is written, the entire command is the **Request** and **Value**.

Transaction		Radix: auto
Timestamp	0:09.993.494.633	
Duration	17.516 us	
Length	8 Bytes	
Type	SETUP	
Device	4	
Endpoint	0	
Data	0x40 0x06 0x04 ...	
Status	ACK	

SETUP Data		Radix: auto
bmRequestType.Recipient	Device (0b00000)	
bmRequestType.Type	Vendor (0b10)	
bmRequestType.Direction	Host-to-Device (0b0)	
bRequest	0x06	
wValue	0x0004	
wIndex	0x0000	
wLength	0x0000	

Command #2 uses the same **bRequest** but with a different **wValue** of 0x01.

Transaction		Radix: auto
Timestamp	0:34.505.982.950	
Duration	17.400 us	
Length	8 Bytes	
Type	SETUP	
Device	4	
Endpoint	0	
Data	0x40 0x06 0x01 ...	
Status	ACK	

SETUP Data		Radix: auto
bmRequestType.Recipient	Device (0b00000)	
bmRequestType.Type	Vendor (0b10)	
bmRequestType.Direction	Host-to-Device (0b0)	
bRequest	0x06	
wValue	0x0001	
wIndex	0x0000	
wLength	0x0000	

Command #3 is a different **bRequest** of 0x31 and a **wValue** of 0xffd0.

Transaction		Radix: auto
Timestamp	0:34.774.137.033	
Duration	17.716 us	
Length	8 Bytes	
Type	SETUP	
Device	4	
Endpoint	0	
Data	0x40 0x31 0xD0 ...	
Status	ACK	

SETUP Data		Radix: auto
bmRequestType.Recipient	Device (0b00000)	
bmRequestType.Type	Vendor (0b10)	
bmRequestType.Direction	Host-to-Device (0b0)	
bRequest	0x31	
wValue	0xffff0	
wIndex	0x0000	
wLength	0x0000	

Command #4 is the same **bRequest** and a **wValue** of 0xffff0.

Transaction		Radix: auto
Timestamp	0:37.341.197.566	
Duration	17.416 us	
Length	8 Bytes	
Type	SETUP	
Device	4	
Endpoint	0	
Data	0x40 0x31 0xF0 ...	
Status	ACK	

SETUP Data		Radix: auto
bmRequestType.Recipient	Device (0b00000)	
bmRequestType.Type	Vendor (0b10)	
bmRequestType.Direction	Host-to-Device (0b0)	
bRequest	0x31	
wValue	0xffff0	
wIndex	0x0000	
wLength	0x0000	

Now we've determined there are two request commands we can send. One is 0x06 and the other is 0x31

Time to experiment!

Command #1 & 2 - LED blinky!

We'll edit our python code to just send command #1 and see what happens. From our logs we know that for sending commands from host-to-device, we should use **bRequestType** of 0x40 (verify this by looking at the **bmRequestType** bits of the command packets), **wIndex** and **wLength** of zero

For command #1, set **bRequest** to 0x06 and a **wValue** to 4. The final argument is now an empty array `[]` to indicate no data is transmitted

```
import usb.core
import usb.util
import sys

# find our device
dev = usb.core.find(idVendor=0x045e, idProduct=0x02B0)

# was it found?
if dev is None:
    raise ValueError('Device not found')

# set the active configuration. With no arguments, the first
# configuration will be the active one
dev.set_configuration()

ret = dev.ctrl_transfer(0x40, 0x06, 0x01, 0, [])
print ret
```

We ran our python code and...nothing happened!

OK well maybe that was some initialization command. Lets replace it with the next command #2, set **bRequest** to 0x06 and a **wValue** to 1

```
ret = dev.ctrl_transfer(0x40, 0x06, 0x01, 0, [])
```

We ran this command and the motor didn't move but the LED stopped blinking.

For fun we ran the previous command again and the LED started blinking again.

Now we have an idea: maybe this **bRequest** 0x06 controls the LED?

On your own, continue this line of thought by trying different **wValues** from 0 on up to see what other **wValues** do, keep track of them all in a notebook or project file.

Command #3 & 4 - Let's move!

Having conquered one of the commands, we'll now tackle the other one. Try to replicate command #3, set **bRequest** to 0x31 and a **wValue** to 0xffd0 (also known as -48 for a 2-byte word)

```
ret = dev.ctrl_transfer(0x40, 0x31, 0xffd0, 0, [])
```

Running the python script made the motor move its 'head' down.

Now try command #4, 0xffff0 (also known as -16 for a 2-byte word)

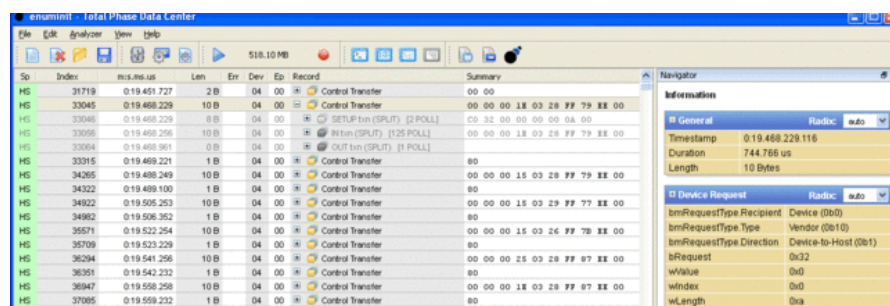
```
ret = dev.ctrl_transfer(0x40, 0x31, 0xffff0, 0, [])
```

This makes the head move up. Now we have both the motor and LED under our control! Here is a video we shot a few minutes after getting the motor working, using a python script to move it up and down.



Bonus Accelerometer!

We're going to go back and revisit the mysterious Read command 0x32 that we fuzzed with for a bit. Its also in the logs, be sure to set your filter to show both Host-to-Device and Device-to-Host since its a 'read' not a 'write'



We were pretty close with our commands, it looks like we should be reading only 10 bytes. It also looks like the data doesn't really change much except for a bit further down...

	Summary
Control Transfer	00 00 00 15 03 28 FF 7B EE 00
Control Transfer	20
Control Transfer	00 00 00 15 03 29 FF 77 ED 00
Control Transfer	20
Control Transfer	00 00 00 15 03 28 FF 7F ED 00
Control Transfer	20
Control Transfer	00 00 00 15 03 28 FF 7F EE 00
Control Transfer	20
Control Transfer	00 00 00 1E 03 27 FF 7F EE 00
Control Transfer	20
Control Transfer	
SETUP tsn (SPLIT) [3 POLL]	40 31 D0 FF 00 00 00 00
IN tsn (SPLIT) [61 POLL]	
Control Transfer	00 07 00 25 03 28 FF 7F 30 04
Control Transfer	21
Control Transfer	00 14 00 22 03 28 FF 98 30 04
Control Transfer	21
Control Transfer	00 16 00 22 03 28 FF A7 30 04
Control Transfer	21
Control Transfer	00 16 00 15 03 28 FF 87 30 04
Control Transfer	21
Control Transfer	00 16 00 06 03 28 FF 57 30 04
Control Transfer	21
Control Transfer	00 18 00 15 03 18 FF 67 30 04
Control Transfer	21
Control Transfer	00 1A 00 22 03 20 FF 79 30 04
Control Transfer	21
Control Transfer	00 19 00 15 03 28 FF 7F 30 04
Control Transfer	21
Control Transfer	00 1B 00 15 03 28 FF 6F 30 04
Control Transfer	21
Control Transfer	00 1A 00 24 03 18 FF 57 30 04
Control Transfer	21
Control Transfer	00 16 00 15 03 18 FF 57 30 04
Control Transfer	21
Control Transfer	00 17 00 15 03 20 FF 5B 30 04
Control Transfer	21

The 7'th byte changes a lot right after we send it that **bRequest** 0x31 (motor movement). That implies that this data read is somehow affected by the motor, possibly a motor feedback byte?

Checking out a tear-down of the device (from iFixit) (<https://adafru.it/aLY>) we see that there is an 'inclinometer'/accelerometer (<https://adafru.it/xDH>) (Kionix KXSD9). The datasheet indicates it is used for image stabilization, and it has 3 axes (X Y and Z) with 10 bits of data per axis.

Lets continuously read that data


```
import usb.core
import usb.util
import sys
import time

# find our device
dev = usb.core.find(idVendor=0x045e, idProduct=0x02B0)

# was it found?
if dev is None:
    raise ValueError('Device not found')

dev.set_configuration()

while True:
    # Get data from brequest 0x32
    ret = dev.ctrl_transfer(0xC0, 0x32, 0x0, 0x0, 10)
    print map(hex, ret)
```

Shaking the Kinect while running the script you'll see clearly that the data changes with movement.

To identify the accelerometer axes, rotate it only one way at a time and note what changes. You can also see how this data is in bytes but the accelerometer data should be a signed word because there are flips from 0xffff to 0x0007 which would indicate a negative to positive conversion.

```
[ 0x0 0x0 0x0 0x76 0xfc 0xe8 0xff 0xbf 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xe8 0xff 0xbf 0x80 0x0 ]
[ 0x0 0x0 0x0 0x7e 0xfc 0xe8 0xff 0xd7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xe8 0xff 0xe7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xe8 0xff 0xe7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xec 0xff 0xa1 0x80 0x0 ]
[ 0x0 0x0 0x0 0x6d 0xfc 0xea 0xff 0xd7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x6d 0xfc 0xea 0xff 0xd7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xe0 0xff 0xe7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x7e 0xfc 0xe6 0xff 0xdb 0x80 0x0 ]
[ 0x0 0x0 0x0 0x7e 0xfc 0xe6 0xff 0xdb 0x80 0x0 ]
[ 0x0 0x0 0x0 0x86 0xfc 0xca 0xff 0xe7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x86 0xfc 0xca 0xff 0xe7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x86 0xfc 0xd8 0xff 0xf3 0x80 0x0 ]
[ 0x0 0x0 0x0 0x7e 0xfc 0xc8 0xff 0xf7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x7e 0xfc 0xc8 0xff 0xf7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x85 0xfc 0xc8 0xff 0xf7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xc8 0x0 0x7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xc8 0x0 0xf 0x80 0x0 ]
[ 0x0 0x0 0x0 0x67 0xfc 0xc8 0x0 0x7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x67 0xfc 0xc8 0x0 0x7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x6d 0xfc 0xc6 0x0 0x7 0x80 0x0 ]
[ 0x0 0x0 0x0 0x66 0xfc 0xc8 0xff 0xff 0x80 0x0 ]
[ 0x0 0x0 0x0 0x66 0xfc 0xc8 0xff 0xff 0x80 0x0 ]
[ 0x0 0x0 0x0 0x64 0xfc 0xc4 0x0 0x17 0x80 0x0 ]
[ 0x0 0x0 0x0 0x6d 0xfc 0xb0 0x0 0x1b 0x80 0x0 ]
[ 0x0 0x0 0x0 0x6d 0xfc 0xb0 0x0 0x1b 0x80 0x0 ]
[ 0x0 0x0 0x0 0x66 0xfc 0xaa 0x0 0x33 0x80 0x0 ]
[ 0x0 0x0 0x0 0x66 0xfc 0xaa 0x0 0x33 0x80 0x0 ]
[ 0x0 0x0 0x0 0x76 0xfc 0xb0 0x0 0x2f 0x80 0x0 ]
[ 0x0 0x0 0x0 0x69 0xfc 0xa8 0x0 0x1b 0x80 0x0 ]
[ 0x0 0x0 0x0 0x69 0xfc 0xa8 0x0 0x1b 0x80 0x0 ]
[ 0x0 0x0 0x0 0x66 0xfc 0xb8 0x0 0x27 0x80 0x0 ]
[ 0x0 0x0 0x0 0x66 0xfc 0xb8 0x0 0x27 0x80 0x0 ]
[ 0x0 0x0 0x0 0x64 0xfc 0xc0 0x0 0x37 0x80 0x0 ]
[ 0x0 0x0 0x0 0x5e 0xfc 0xc0 0x0 0x35 0x80 0x0 ]
```

We can cast two bytes to a signed value by 'hand' (in C this is a little easier, we know)

```

import usb.core
import usb.util
import sys
import time

# find our device
dev = usb.core.find(idVendor=0x045e, idProduct=0x02B0)

# was it found?
if dev is None:
    raise ValueError('Device not found')

dev.set_configuration()

while True:
    # Get data from brequest 0x32
    ret = dev.ctrl_transfer(0xC0, 0x32, 0x0, 0x0, 10)
    #print map(hex, ret)

    x = (ret[2] << 8) | ret[3]
    x = (x + 2 ** 15) % 2**16 - 2**15    # convert to signed 16b
    y = (ret[4] << 8) | ret[5]
    y = (y + 2 ** 15) % 2**16 - 2**15    # convert to signed 16b
    z = (ret[6] << 8) | ret[7]
    z = (z + 2 ** 15) % 2**16 - 2**15    # convert to signed 16b

    print x, "\t", y, "\t", z

```

Now when you run the script you'll see the signed data appear properly.

```

-74    312    -425
-74    312    -425
-58    332    -441
-58    332    -441
-58    332    -441
-26    360    -441
-26    360    -441
13     384    -449
13     384    -449
45     396    -441
45     396    -441
69     425    -425
69     425    -425
69     425    -425
102    472    -401
102    472    -401
102    488    -425
102    488    -425
126    504    -457

```

More Kinect Information

We hope you enjoyed this reverse-engineering tutorial. For more information about Open Kinect, [please visit the github repository \(https://adafru.it/aM0\)](https://adafru.it/aM0) and [google group \(https://adafru.it/aM1\)](https://adafru.it/aM1).