# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>3</td>
</tr>
<tr>
<td>Verify the VID &amp; PID</td>
<td>3</td>
</tr>
<tr>
<td>Determine the Descriptors</td>
<td>5</td>
</tr>
<tr>
<td>Making a Driver</td>
<td>7</td>
</tr>
<tr>
<td>Installing Python &amp; PyUSB</td>
<td>10</td>
</tr>
<tr>
<td>Fuzzing</td>
<td>10</td>
</tr>
<tr>
<td>USB Analyzer</td>
<td>13</td>
</tr>
<tr>
<td>• Lookin' at Logs</td>
<td></td>
</tr>
<tr>
<td>Command #1 &amp; 2 - LED blinky!</td>
<td>23</td>
</tr>
<tr>
<td>Command #3 &amp; 4 - Let's move!</td>
<td>24</td>
</tr>
<tr>
<td>Bonus Accelerometer!</td>
<td>25</td>
</tr>
<tr>
<td>More Kinect Information</td>
<td>28</td>
</tr>
</tbody>
</table>
Overview

Everyone has seen the Xbox 360 Kinect hacked in a matter of days after our "open source driver" bounty - 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, 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/utils/usb_devices_view.html () may do the trick]]

For Linux, run `lsusb -vv` (ultra verbose) for Mac, run `system_profiler SPUSBDataType`.

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 it's the simplest. Note the Vendor ID = 0x045e and Product ID = 0x2b0. 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)

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 it's 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](http://janaxelson.com) 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
 iVendor            0x045e Microsoft Corp.
 iProduct           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
```
This device has two Isochronous endpoints both of which are IN type (data going IN to the computer). This makes sense: the Kinect has an 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 () 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.

Next up, install PyUSB by downloading it and running python setup.py install in the expanded directory

Fuzzing

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

```python
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.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Type</th>
<th>Recipient</th>
</tr>
</thead>
<tbody>
<tr>
<td>D7</td>
<td>D6</td>
<td>D5</td>
</tr>
</tbody>
</table>

For `bmRequestType` the value passed is very structured so that's not as hard to guess. ([See lvr.com for more information](https://lvr.com))

- 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 its 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`. Its 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.

```python
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!

# Let's 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

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

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. 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 and click the **Downloads** button.

Make yourself a sandwich, its a big file!

Also download the Beagle Data Center software (Mac/Win/Linux) and install it.

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

![Image of enuminit.tdc file](image)

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. Let's 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. It's not important.
Click on (1) This device is a Class device type USB Hub. That's the internal hub. We can ignore this as well.

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. Let's filter some more to see just commands sent to the device.
Go to the LiveFilter and select Host-to-Device.
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.

Command #2 uses the same bRequest but with a different wValue of 0x01.
Command #3 is a different bRequest of 0x31 and a wValue of 0xffd0.

Command #4 is the same bRequest and a wValue of 0xfff0.
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 bmRequestType 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.

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

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

```
ret = dev.ctrl_transfer(0x40, 0x6, 0x1, 0, [])
print ret
```

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 0x6 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, 0xfff0, 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. It's also in the logs, be sure to set your filter to show both Host-to-Device and Device-to-Host since it's 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...
The 7th 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) we see that there is an 'inclinometer'/accelerometer (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

```python
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 0xfff7 to 0x0007 which would indicate a negative to positive conversion.

We can cast two bytes to a signed value by 'hand' (in C this is a little easier, we know)
while True:
    # Get data from brequest 0x32
    ret = dev.ctrl_transfer(0xC0, 0x32, 0x0, 0x0, 10)
    #print map(hex, ret)
    x = (ret[2] &lt;&lt; 8) | ret[3] # convert to signed 16b
    x = (x + 2 ** 15) % 2**16 - 2**15
    y = (y + 2 ** 15) % 2**16 - 2**15
    z = (ret[6] &lt;&lt; 8) | ret[7] # convert to signed 16b
    z = (z + 2 ** 15) % 2**16 - 2**15
    print x, "\t", y, "\t", z

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>512</td>
<td>-425</td>
</tr>
<tr>
<td>74</td>
<td>312</td>
<td>-425</td>
</tr>
<tr>
<td>-58</td>
<td>332</td>
<td>-441</td>
</tr>
<tr>
<td>-58</td>
<td>332</td>
<td>-441</td>
</tr>
<tr>
<td>-58</td>
<td>332</td>
<td>-441</td>
</tr>
<tr>
<td>26</td>
<td>360</td>
<td>-441</td>
</tr>
<tr>
<td>26</td>
<td>360</td>
<td>-441</td>
</tr>
<tr>
<td>13</td>
<td>384</td>
<td>-449</td>
</tr>
<tr>
<td>13</td>
<td>384</td>
<td>-449</td>
</tr>
<tr>
<td>46</td>
<td>396</td>
<td>-441</td>
</tr>
<tr>
<td>46</td>
<td>396</td>
<td>-441</td>
</tr>
<tr>
<td>59</td>
<td>425</td>
<td>-425</td>
</tr>
<tr>
<td>59</td>
<td>425</td>
<td>-425</td>
</tr>
<tr>
<td>69</td>
<td>425</td>
<td>-425</td>
</tr>
<tr>
<td>69</td>
<td>425</td>
<td>-425</td>
</tr>
<tr>
<td>102</td>
<td>472</td>
<td>-401</td>
</tr>
<tr>
<td>102</td>
<td>472</td>
<td>-401</td>
</tr>
<tr>
<td>102</td>
<td>488</td>
<td>-425</td>
</tr>
<tr>
<td>102</td>
<td>488</td>
<td>-425</td>
</tr>
<tr>
<td>126</td>
<td>504</td>
<td>-457</td>
</tr>
</tbody>
</table>

More Kinect Information

We hope you enjoyed this reverse-engineering tutorial. For more information about Open Kinect, please visit the github repository and google group.