R also: https://github.com/Free60Project/wiki/blob/master/GamePad.md

- no longer a HID device (Xinput / XUsb device)
- 360 controller is a HID controller with the HID report descriptor removed and the device class, subclass, and protocol set to 0xFF (255)
  (ensures that the HID driver does not attach to the device so the Microsoft driver can pick it up)
- vendor ID is 0x45e (Microsoft) and the product ID is 0x028e
  (Once you set the device to the same VID and PID the driver will attach to the driver and attempt to enumerate it)
- two descriptors you need to worry about for the controller (because there is no HID report descriptor):
  -> the device descriptor, which tells the host things such as
  USB version,
  device class,
  device subclass,
  device protocol,
  max packet size for endpoint 0,
  VID,
  PID,
  etc

  -> the configuration descriptor, which tells the host about the device configuration, each interface, and their endpoints
There is some padding included in the controller's actual configuration and that padding needs to be there for the device to correctly enumerate

used on Teensyduino to define USB devices

```
// usb_desc.h L868
#define DEVICE_CLASS        0xFF
#define DEVICE_SUBCLASS     0xFF
#define DEVICE_PROTOCOL     0xFF
#define BCD_DEVICE 0x0114 // oldly 'DEVICE_VERSION'
#define DEVICE_ATTRIBUTES 0xA0
#define DEVICE_POWER        0xFA
#define VENDOR_ID           0x045e
#define PRODUCT_ID          0x028e
#define MANUFACTURER_NAME {'(','C',')','M','i','c','r','o','s','o','f','t'} // with LEN 10 or 11, show the default S/N
#define MANUFACTURER_NAME_LEN      10 // just in case this 'd fix the S/N .. ( then, try without a o ..)
#define PRODUCT_NAME        {'C','o','n','t','r','o','l','l','e','r'}
#define PRODUCT_NAME_LEN    10
#define EP0_SIZE            8
#define NUM_ENDPOINTS       6
#define NUM_USB_BUFFERS 24
#define NUM_INTERFACE       4
#define XINPUT_INTERFACE  0
#define XINPUT_RX_ENDPOINT      2
#define XINPUT_RX_SIZE 8 // oldly '3' ?!
#define XINPUT_TX_ENDPOINT      1
#define XINPUT_TX_SIZE 20
#define CONFIG_DESC_SIZE 153
#define ENDPOINT1_CONFIG ENDPOINT_TRANSMIT_ONLY
#define ENDPOINT2_CONFIG ENDPOINT_RECEIVE_ONLY
#define ENDPOINT3_CONFIG ENDPOINT_TRANSMIT_ONLY
#define ENDPOINT4_CONFIG ENDPOINT_RECEIVE_ONLY
#define ENDPOINT5_CONFIG ENDPOINT_TRANSMIT_AND_RECEIVE
#define ENDPOINT6_CONFIG ENDPOINT_TRANSMIT_ONLY
```

```
// usb_desc.c L66 is the start of USB Device Descriptor
static uint8_t device_descriptor[] = {
    18,                     // bLength
    1,                      // bDescriptorType
    0x00, 0x02,             // bcdUSB
#ifdef DEVICE_CLASS
    DEVICE_CLASS,           // bDeviceClass
#else
    0,
#endif
#ifdef DEVICE_SUBCLASS
    DEVICE_SUBCLASS,        // bDeviceSubClass
#else
    0,
#endif
#ifdef DEVICE_PROTOCOL
    DEVICE_PROTOCOL,        // bDeviceProtocol
#else
    0,
#endif
    EP0_SIZE,               // bMaxPacketSize0
    LSB(VENDOR_ID), MSB(VENDOR_ID),     // idVendor
    LSB(PRODUCT_ID), MSB(PRODUCT_ID),   // idProduct
#ifdef BCD_DEVICE
    LSB(BCD_DEVICE), MSB(BCD_DEVICE),   // bcdDevice
#else
  // For USB types that don't explicitly define BCD_DEVICE,
  // use the minor version number to help teensy_ports
  // identify which Teensy model is used.
  #if defined(__MKL26Z64__)
    0x73, 0x02,
  #elif defined(__MK20DX128__)
    0x74, 0x02,
  #elif defined(__MK20DX256__)
    0x75, 0x02,
  #elif defined(__MK64FX512__)
    0x76, 0x02,
  #elif defined(__MK66FX1M0__)
    0x77, 0x02,
  #else
    0x00, 0x02,
  #endif
#endif
    1,                      // iManufacturer
    2,                      // iProduct
    3,                      // iSerialNumber
    1                       // bNumConfigurations
};

// These descriptors must NOT be "const", because the USB DMA
// has trouble accessing flash memory with enough bandwidth
// while the processor is executing from flash.
```

#define LSB(n) ((n) & 255)
#define MSB(n) (((n) >> 8) & 255)

```
// usb_desc.c L123 is the start of HID  Report descriptors
-> not used for Xbox360 controller
```

These exist because our device is not HID nor is it generic.
Normally for a HID device the class, subclass, and protocol are all the same,
however we need to set them to whats known as Vendor Specific

The device class, subclass, and protocol are all set to 0xFF.
This signifies to the host that these are vendor specific and handled by a driver

Endpoint 0 is a reserved endpoint that is used to perform control transfers.
This is what the host uses to probe the device and initialize it. Endpoint 0 has its size set to 8 bytes

```
// usb_desc.c L582 is the start of USB Configuration Descriptor
static uint8_t config_descriptor[CONFIG_DESC_SIZE] = {
    // configuration descriptor, USB spec 9.6.3, page 264-266, Table 9-10
    9,                      // bLength
    2,                      // bDescriptorType ( 2 == configuration descriptor )
    LSB(CONFIG_DESC_SIZE),  // wTotalLength
    MSB(CONFIG_DESC_SIZE),
    NUM_INTERFACE,          // bNumInterfaces
    1,                      // bConfigurationValue
    0,                      // iConfiguration
#ifdef DEVICE_ATTRIBUTES
    DEVICE_ATTRIBUTES,                                  // bmAttributes
#else
    0xC0,
#endif
#ifdef DEVICE_POWER
    DEVICE_POWER,                               // bMaxPower
#else
    50,
#endif

#ifdef CDC_IAD_DESCRIPTOR
    // interface association descriptor, USB ECN, Table 9-Z
    8,                      // bLength
    11,                     // bDescriptorType
    CDC_STATUS_INTERFACE,   // bFirstInterface
    2,                      // bInterfaceCount
    0x02,                   // bFunctionClass
    0x02,                   // bFunctionSubClass
    0x01,                   // bFunctionProtocol
    4,                      // iFunction
#endif

#ifdef CDC_DATA_INTERFACE
    // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
    9,                      // bLength
    4,                      // bDescriptorType
    CDC_STATUS_INTERFACE,                       // bInterfaceNumber
    0,                      // bAlternateSetting
    1,                      // bNumEndpoints
    0x02,                   // bInterfaceClass
    0x02,                   // bInterfaceSubClass
    0x01,                   // bInterfaceProtocol
    0,                      // iInterface
    // CDC Header Functional Descriptor, CDC Spec 5.2.3.1, Table 26
    5,                      // bFunctionLength
    0x24,                   // bDescriptorType
    0x00,                   // bDescriptorSubtype
    0x10, 0x01,             // bcdCDC
    // Call Management Functional Descriptor, CDC Spec 5.2.3.2, Table 27
    5,                      // bFunctionLength
    0x24,                   // bDescriptorType
    0x01,                   // bDescriptorSubtype
    0x01,                   // bmCapabilities
    1,                      // bDataInterface
    // Abstract Control Management Functional Descriptor, CDC Spec 5.2.3.3, Table 28
    4,                      // bFunctionLength
    0x24,                   // bDescriptorType
    0x02,                   // bDescriptorSubtype
    0x06,                   // bmCapabilities
    // Union Functional Descriptor, CDC Spec 5.2.3.8, Table 33
    5,                      // bFunctionLength
    0x24,                   // bDescriptorType
    0x06,                   // bDescriptorSubtype
    CDC_STATUS_INTERFACE,   // bMasterInterface
    CDC_DATA_INTERFACE,     // bSlaveInterface0
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,                      // bLength
    5,                      // bDescriptorType
    CDC_ACM_ENDPOINT | 0x80,    // bEndpointAddress
    0x03,                   // bmAttributes (0x03=intr)
    CDC_ACM_SIZE, 0,        // wMaxPacketSize
    64,                     // bInterval
    // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
    9,                      // bLength
    4,                      // bDescriptorType
    CDC_DATA_INTERFACE,                         // bInterfaceNumber
    0,                      // bAlternateSetting
    2,                      // bNumEndpoints
    0x0A,                   // bInterfaceClass
    0x00,                   // bInterfaceSubClass
    0x00,                   // bInterfaceProtocol
    0,                      // iInterface
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,                      // bLength
    5,                      // bDescriptorType
    CDC_RX_ENDPOINT,        // bEndpointAddress
    0x02,                   // bmAttributes (0x02=bulk)
    CDC_RX_SIZE, 0,         // wMaxPacketSize
    0,                      // bInterval
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,                      // bLength
    5,                      // bDescriptorType
    CDC_TX_ENDPOINT | 0x80,     // bEndpointAddress
    0x02,                   // bmAttributes (0x02=bulk)
    CDC_TX_SIZE, 0,         // wMaxPacketSize
    0,                      // bInterval
#endif // CDC_DATA_INTERFACE

#ifdef XINPUT_INTERFACE
//Interface 0
    9,                          //bLength (length of interface descriptor 9 bytes)
    4,                          //bDescriptorType (4 is interface)
    0,                          //bInterfaceNumber (This is interface 0)
    0,                          //bAlternateSetting (used to select alternate setting.  notused)
    2,                          //bNumEndpoints (this interface has 2 endpoints)
    0xFF,                       //bInterfaceClass (Vendor Defined is 255)
    0x5D,                       //bInterfaceSubClass
    0x01,                       //bInterfaceProtocol
    0,                          //interface (Index of string descriptor for describing this notused)
    //Some sort of common descriptor? I pulled this from Message Analyzer dumps of an actual controller
    17,33,0,1,1,37,129,20,0,0,0,0,19,2,8,0,0,
    //Endpoint 1 IN
    7,                          //bLength (length of ep1in in descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x81,                       //bEndpointAddress (0x81 is IN1)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    4,                          //bInterval (polling interval in frames 4 frames)
    //Endpoint 2 OUT
    7,                          //bLength (length of ep2out in descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x02,                       //bEndpointAddress (0x02 is OUT2)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    8,                          //bInterval (polling interval in frames 8 frames)
//Interface 1
    9,                          //bLength (length of interface descriptor 9 bytes)
    4,                          //bDescriptorType (4 is interface)
    1,                          //bInterfaceNumber (This is interface 1)
    0,                          //bAlternateSetting (used to select alternate setting.  notused)
    4,                          //bNumEndpoints (this interface has 4 endpoints)
    0xFF,                       //bInterfaceClass (Vendor Defined is 255)
    0x5D,                       //bInterfaceSubClass (93)
    0x03,                       //bInterfaceProtocol (3)
    0,                          //interface (Index of string descriptor for describing this notused)
    //A different common descriptor? I pulled this from Message Analyzer dumps of an actual controller
    27,33,0,1,1,1,131,64,1,4,32,22,133,0,0,0,0,0,22,5,0,0,0,0,0,
    //Endpoint 3 IN
    7,                          //bLength (length of ep3in descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x83,                       //bEndpointAddress (0x83 is IN3)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    2,                          //bInterval (polling interval in frames 2 frames)
    //Endpoint 4 OUT
    7,                          //bLength (length of ep4out descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x04,                       //bEndpointAddress (0x04 is OUT4)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    4,                          //bInterval (polling interval in frames 4 frames)
    //Endpoint 5 IN
    7,                          //bLength (length of ep5in descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x85,                       //bEndpointAddress (0x85 is IN5)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    64,                         //bInterval (polling interval in frames 64 frames)
    //Endpoint 5 OUT (shares endpoint number with previous)
    7,                          //bLength (length of ep5out descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x05,                       //bEndpointAddress (0x05 is OUT5)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    16,                         //bInterval (polling interval in frames 16 frames)
//Interface 2
    9,                          //bLength (length of interface descriptor 9 bytes)
    4,                          //bDescriptorType (4 is interface)
    2,                          //bInterfaceNumber (This is interface 2)
    0,                          //bAlternateSetting (used to select alternate setting.  notused)
    1,                          //bNumEndpoints (this interface has 4 endpoints)
    0xFF,                       //bInterfaceClass (Vendor Defined is 255)
    0x5D,                       //bInterfaceSubClass (93)
    0x02,                       //bInterfaceProtocol (3)
    0,                          //Interface (Index of string descriptor for describing this notused)
    //Common Descriptor. Seems that these come after every interface description?
    9,33,0,1,1,34,134,7,0,
    //Endpoint 6 IN
    7,                          //bLength (length of ep6in descriptor 7 bytes)
    5,                          //bDescriptorType (5 is endpoint)
    0x86,                       //bEndpointAddress (0x86 is IN6)
    0x03,                       //bmAttributes (0x03 is interrupt no synch, usage type data)
    0x20, 0x00,                 //wMaxPacketSize (0x0020 is 1x32 bytes)
    16,                         //bInterval (polling interval in frames 64 frames)+
//Interface 3
//This is the interface on which all the security handshaking takes place
//We don't use this but it could be used for man-in-the-middle stuff
    9,                          //bLength (length of interface descriptor 9 bytes)
    4,                          //bDescriptorType (4 is interface)
    3,                          //bInterfaceNumber (This is interface 3)
    0,                          //bAlternateSetting (used to select alternate setting.  notused)
    0,                          //bNumEndpoints (this interface has 0 endpoints ???)
    0xFF,                       //bInterfaceClass (Vendor Defined is 255)
    0xFD,                       //bInterfaceSubClass (253)
    0x13,                       //bInterfaceProtocol (19)
    4,                          //interface (Computer never asks for this, but an x360 would. so include one day?)
    //Another interface another Common Descriptor
    6,65,0,1,1,3
#endif // XINPUT_INTERFACE
};
```

only interface that concerns us on the PC

directions are based from the host's point of view and are described in the MSB of the endpoint address.
With setting the most significant bit to a 1 making the endpoint an IN.
This means that all button reports going TO the PC with be sent on endpoint 0x81,
while any LED patterns the PC sends the controller are being RECEIVED by the device on endpoint 0x02

other endpoints are used for other peripheral such as headsets, chatpads, etc

only other interface that might be of use to you is the last one, interface 4 (3 ?)
It claims it has 0 endpoints, but this is the interface that handles the security handshake with the Infineon security chip. It is the only interface with a string descriptor

added string descriptor just in case someone decides they want to attempt to replicate a man in the middle
device similar to what BrandonW created.
In emails between myself and Brandon,
he indicated that the 360 will just stop communicating with the controller if this string descriptor for interface 4 doesn't exist.

```
// usb_desc.c L1576
struct usb_string_descriptor_struct usb_string_xinput_security_descriptor = {
    178,
    3,
    {
      0x58, 0x00, 0x62, 0x00, 0x6F, 0x00, 0x78, 0x00, 0x20, 0x00, 0x53, 0x00, 0x65, 0x00, 0x63, 0x00,
      0x75, 0x00, 0x72, 0x00, 0x69, 0x00, 0x74, 0x00, 0x79, 0x00, 0x20, 0x00, 0x40, 0x00, 0x65, 0x00,
      0x74, 0x00, 0x68, 0x00, 0x6F, 0x00, 0x64, 0x00, 0x20, 0x00, 0x33, 0x00, 0x2C, 0x00, 0x20, 0x00,
      0x56, 0x00, 0x65, 0x00, 0x72, 0x00, 0x73, 0x00, 0x69, 0x00, 0x6F, 0x00, 0x6E, 0x00, 0x20, 0x00,
      0x31, 0x00, 0x2E, 0x00, 0x30, 0x00, 0x30, 0x00, 0x2C, 0x00, 0x20, 0x00, 0x49, 0x00, 0x20, 0x00,
      0x32, 0x00, 0x30, 0x00, 0x30, 0x00, 0x35, 0x00, 0x20, 0x00, 0x43, 0x00, 0x69, 0x00, 0x63, 0x00,
      0x72, 0x00, 0x6F, 0x00, 0x73, 0x00, 0x6F, 0x00, 0x66, 0x00, 0x74, 0x00, 0x20, 0x00, 0x43, 0x00,
      0x6F, 0x00, 0x72, 0x00, 0x70, 0x00, 0x6F, 0x00, 0x72, 0x00, 0x61, 0x00, 0x74, 0x00, 0x69, 0x00,
      0x6F, 0x00, 0x6E, 0x00, 0x2E, 0x00, 0x20, 0x00, 0x41, 0x00, 0x6C, 0x00, 0x6C, 0x00, 0x20, 0x00,
      0x72, 0x00, 0x69, 0x00, 0x67, 0x00, 0x68, 0x00, 0x74, 0x00, 0x73, 0x00, 0x20, 0x00, 0x72, 0x00,
      0x65, 0x00, 0x73, 0x00, 0x65, 0x00, 0x72, 0x00, 0x76, 0x00, 0x65, 0x00, 0x64, 0x00, 0x2E, 0x00
    }
};
```

the lInterface value for interface 3 states a value of 4. This means that string 4 contains a descriptor for this interface.
This is the security interface used to handshake with an actual XBOX 360.
We really do not need this, however, anyone that might try a man in the middle style device will need this.
The 360 will cease communication with the device if this string is not returned.
So we define a structure of type usb_string_descriptor_struct and fill it with the correct string.

```
// from zlittell.com tut'
struct usb_string_descriptor_struct usb_string_xinput_security_descriptor = {
    0xB2, 0x03, 0x58, 0x00, 0x62, 0x00, 0x6F, 0x00, 0x78, 0x00, 0x20, 0x00, 0x53, 0x00, 0x65, 0x00,
    0x63, 0x00, 0x75, 0x00, 0x72, 0x00, 0x69, 0x00, 0x74, 0x00, 0x79, 0x00, 0x20, 0x00, 0x40, 0x00,
    0x65, 0x00, 0x74, 0x00, 0x68, 0x00, 0x6F, 0x00, 0x64, 0x00, 0x20, 0x00, 0x33, 0x00, 0x2C, 0x00,
    0x20, 0x00, 0x56, 0x00, 0x65, 0x00, 0x72, 0x00, 0x73, 0x00, 0x69, 0x00, 0x6F, 0x00, 0x6E, 0x00,
    0x20, 0x00, 0x31, 0x00, 0x2E, 0x00, 0x30, 0x00, 0x30, 0x00, 0x2C, 0x00, 0x20, 0x00, 0x49, 0x00,
    0x20, 0x00, 0x32, 0x00, 0x30, 0x00, 0x30, 0x00, 0x35, 0x00, 0x20, 0x00, 0x43, 0x00, 0x69, 0x00,
    0x63, 0x00, 0x72, 0x00, 0x6F, 0x00, 0x73, 0x00, 0x6F, 0x00, 0x66, 0x00, 0x74, 0x00, 0x20, 0x00,
    0x43, 0x00, 0x6F, 0x00, 0x72, 0x00, 0x70, 0x00, 0x6F, 0x00, 0x72, 0x00, 0x61, 0x00, 0x74, 0x00,
    0x69, 0x00, 0x6F, 0x00, 0x6E, 0x00, 0x2E, 0x00, 0x20, 0x00, 0x41, 0x00, 0x6C, 0x00, 0x6C, 0x00,
    0x20, 0x00, 0x72, 0x00, 0x69, 0x00, 0x67, 0x00, 0x68, 0x00, 0x74, 0x00, 0x73, 0x00, 0x20, 0x00,
    0x72, 0x00, 0x65, 0x00, 0x73, 0x00, 0x65, 0x00, 0x72, 0x00, 0x76, 0x00, 0x65, 0x00, 0x64, 0x00,
    0x2E, 0x00
};
```

Finally we need to add the command used to ask for this string and specify a response in the usb_descriptor_list[] array.
The host will send a request for 0x0304.
The 0x03 portion indicates the host wants a string.
The 0x04 is the index, which as you remember, is the index we specified as interface 3's lInterface.
We then tell the device to return security descriptor string we just set up.

```
#ifdef XINPUT_INTERFACE
  {0x0304, 0x0409, (const uint8_t *)&usb_string_xinput_security_descriptor, 0},
#endif
```

next thing that we need to look at, is the way the controller sends button reports to the host.
Once again, Free60 is on top of things with this.
However, pay close attention to the indexes provided. The standard bit format of a byte goes:
bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1, and bit 0.
Remember this is sent from the device to the host on endpoint 1 in a 20 byte packet.

last endpoint we really care about is endpoint 2 out.
This comes into the device from the PC and contains messages.
The only message we care about is message type 0x01, which is LED control.
The packet will come in 3 bytes with byte 1 being message type and byte 2 as the length.
The 3rd byte contains the pattern to be performed

We need to add 3 simple lines to insert our device into the TeensyLC usb menu,
which when selected defines USB_XINPUT as the usb type

```
==== for Arduino only ====
// boards.txt
teensyLC.menu.usb.xinput=(MS!) Shoryuken! (XINPUT DEVICE)
teensyLC.menu.usb.xinput.build.usbtype=USB_XINPUT
teensyLC.menu.usb.xinput.fake_serial=teensy_gateway
```

We still need to add a couple of pieces of code so that the correct headers are loaded and objects defined.
We need to tell the compiler that if USB_XINPUT is defined,
then it should created an instance of usb_xinput_class labeled Xinput.

```
==== for Arduino only ====
// usb_inst.cpp
#ifdef USB_XINPUT
  usb_xinput_class XInput;
#endif
```

all we have left to do is include usb_xinput.h in the file Wprogram.h.
(for usb_xinput.h & usb_xinput.c, these two files contain the libraries
we use to bridge our code with the standard USB code provided by teensyduino)

```
==== for Arduino only ====
// WProgram.h
#include "usb_xinput.h"
```