First Consulting, Inc.

CUSTOM USB HID FIRMWARE DESIGN GUIDE

Brendan Mulverna

Embedded Systems Engineer

First Consulting Inc.

Rev 1.1

March 25, 2024

First Consulting, Inc.

CONTENTS

1 Introduction

1.1 USB HID

USB communication is a very useful way for a host computer to talk to a device or product. USB HID is short for a USB "Human Interface Device" class. These are devices that are specified as computer peripherals. This can be a device like a keyboard, mouse, or custom HID device allowing the designer to provide USB functions that are desired by a consumer or engineer. A designer that builds the firmware to incorporate a well-designed API and supporting USB communication functions can enable the device to be updated in the field, expose useful debug data to be read during development, control the device through a host PC application, and many other uses.

2 Problem

2.1 Overview

USB is designed to be a standardized communication protocol but can be difficult to get working if the firmware designer is unfamiliar with some of the nuances. There are many microcontroller chip sets that offer USB enabled hardware and example code. To access the most from these USB resources requires an understanding of USB communication and the ability to apply those changes to customize the examples. Example code discussed in this paper is based on NXP's Kinetis MCUXpresso SDKs.

3 Design

3.1 Overview of USB Communication

USB allows communication between host and device with simple byte arrays. The meaning and function of each of these bytes is described to the host through a set of descriptors. A USB HID mouse and keyboard may send the same packet of data to a host computer, but because the devices are configured differently the host computer will perform different functions based on the device.

3.2 USB Descriptors

3.2.1 Overview

Descriptors are byte arrays that are requested by the host during enumeration. The device will respond to each request with data telling the host what type of device it is, manufacturer ID data, number of endpoints, and other useful data.

3.2.2 Device Descriptor

Each device will have one device descriptor which gives vendor and product ID information. Appendix #1 shows an example of a HID Mouse Device Descriptor.

```
uint8_t g_UsbDeviceDescriptor[] = {
...
USB_SHORT_GET_LOW(USB_DEVICE_VID),
USB_SHORT_GET_HIGH(USB_DEVICE_VID), /* Vendor ID (assigned by the USB-IF) */
USB_SHORT_GET_LOW(USB_DEVICE_PID),
USB_SHORT_GET_HIGH(USB_DEVICE_PID), /* Product ID (assigned by the manufacturer) */
...
};
```

Figure 1- Code Snippet from Appendix #1

3.2.3 Configuration Descriptor

This descriptor has a lot of important information about the power requirements for the USB device. This also contains the interface and endpoint descriptors which will explain how the device is laid out and the different endpoints that will send and receive data. These are important to understand because this is how the host identifies the device it's sending and receiving data to and from. Appendix #2 shows an example of a HID Mouse Configuration Descriptor.

```
uint8_t g_UsbDeviceConfigurationDescriptor[] = {
...
  (USB_DESCRIPTOR_CONFIGURE_ATTRIBUTE_D7_MASK) |
    (USB_DEVICE_CONFIG_SELF_POWER << USB_DESCRIPTOR_CONFIGURE_ATTRIBUTE_SELF_POWERED_SHIFT) |
    (USB_DEVICE_CONFIG_REMOTE_WAKEUP << USB_DESCRIPTOR_CONFIGURE_ATTRIBUTE_REMOTE_WAKEUP_SHIFT),
  /* Configuration characteristics
     D7: Reserved (set to one)
     D6: Self-powered
     D5: Remote Wakeup
     D4...0: Reserved (reset to zero)
  */
  USB_DEVICE_MAX_POWER,
...
  USB_DESCRIPTOR_LENGTH_ENDPOINT, /* Size of this descriptor in bytes */
  USB_DESCRIPTOR_TYPE_ENDPOINT,   /* ENDPOINT Descriptor Type */
  USB_HID_MOUSE_ENDPOINT_IN | (USB_IN << USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_SHIFT),
  /* The address of the endpoint on the USB device
    described by this descriptor. */
  USB_ENDPOINT_INTERRUPT, /* This field describes the endpoint's attributes */
  USB_SHORT_GET_LOW(FS_HID_MOUSE_INTERRUPT_IN_PACKET_SIZE),
USB_SHORT_GET_HIGH(FS_HID_MOUSE_INTERRUPT_IN_PACKET_SIZE),
  /* Maximum packet size this endpoint is capable of
    sending or receiving when this configuration is
    selected. */
  FS_HID_MOUSE_INTERRUPT_IN_INTERVAL, /* Interval for polling endpoint for data transfers. */
};
```

Figure 2- Code Snippet from Appendix #2

3.2.4 Report Descriptor

The report descriptor is one of the most important and difficult to get right. This descriptor defines a map used by the host and device to tell what the bytes represent, what are the max values to expect, whether the data is meant as an input or output, and other important parameters. This descriptor can be simple as in a HID mouse with only a few buttons or as complicated as in a HID Sensor collection which could combine data from multiple sensors in a device. Appendix #3 represents a USB descriptor for a HID mouse. The Usage Page and Usage bytes determine what type of device to expect, and the available usages can be searched online at usb.org. We see two main collections of data that combine to create a mouse collection. The first collection describes a mouse that has 3 buttons that'll be reported as a bit field. The second collection describes the mouse movement in the X,Y, and Z coordinates(Z coordinates are used for scrolling and panning).

```
uint8_t g_UsbDeviceHidMouseReportDescriptor[] = {
  0x05U, 0x01U,     /* Usage Page (Generic Desktop)*/
  0x09U, 0x02U,     /* Usage (Mouse) */
  0xA1U, 0x01U,      /* Collection (Application) */
  0x09U, 0x01U,     /* Usage (Pointer) */
...
};
```

Figure 3- Code Snippet from Appendix #3

3.2.5 Custom HID Report Descriptor

The mouse report descriptor is a very common report descriptor to help a designer understand how to write a report descriptor. For a custom HID that we want to use to transfer data and build out an API from, we want to use something like Appendix #4. This sets up a collection made up of an input and output buffer for data transfer. The report ID number will be the first byte in the data transfer telling the direction of the data to the firmware. The firmware will be designed to parse out any other data in the buffer and use it in different functions.

```
uint8_t g_UsbDeviceHidReportDescriptor[USB_DESCRIPTOR_LENGTH_HID_REPORT] = {
                                0x05, 0x01, // USAGE_PAGE (Generic Desktop)
                                0x09, 0x00,                     // USAGE (Undefined)
                                0xa1, 0x01,                     // COLLECTION (Application)
                                0x09, 0x3a,                     //  USAGE (Counted Buffer)
...
};
```

Figure 4- Code Snippet from Appendix #4

3.3 Communication Structure

When sending data between host and device the designer needs an easy way to understand how to correctly pack the data. In C this can be done using typedef

structures and unions. Appendix #5 is one example of doing this. Setting a variable to this structure type allows firmware on each side of USB communication to synchronize the data. If a Windows application sets “repid” and “pktcmd” to a value. The data is then sent across to the microcontroller firmware and the microcontroller can read out the “repid” and “pktcmd” in its own local variable and get the correctly transferred values if using the same type defined structure. This is very important because it makes sure the devices are speaking the same language to each other and bytes aren’t misinterpreted.

```
typedef struct _commandpktID {
        unsigned char repid;// The report id for hid to and from pipes.
        unsigned char pktcmd;// the ID of the command this is a response to.
        union {
                unsigned char       payload[MAX_PKT_PAYLOAD]; // Unformatted data.
                DATAPKT   datapkt;     // Data to write to flash (bootloader only)
                VERSIONPKT          Version;    // Version id's
                unsigned char       value;      // Use for commands that need to send 1 byte of data
  } pktdata;
} IDCOMMANDPKT, *PIDCOMMANDPKT;
...
```

Figure 5- Code Snippet from Appendix #5

3.4 API Protocol

A valuable way in designing the USB data transfer firmware is to assign a specific byte to be a command byte. In the structure of Appendix #5 there is a byte referred to a “pktcmd”. This command is used by the micro to determine what you want to do on the microcontroller. Appendix #6 shows a few example using “pktcmd” to get the firmware version (GET VERSION), to start or end a firmware update (PC_UPDATE_GET_ACTIVE_FW , PC_UPDATE_START and PC_UPDATE_END), to reset the microcontroller (CPU_RESET), or receiving an error report from the microcontroller (ERROR_REPORT). The microcontroller firmware will receive the USB packet in an interrupt functions USB_DeviceHidInterruptOut() shown in Appendix #7. “Out” depicts a message transferred from the host to device (out from the host). This interrupt calls the custom function USB_DeviceHidAction() which parses out the “pktcmd” value into whatever task the host wants the device to perform. Then data if any can be returned to the host through the USB_DeviceSendRequest() function and the USB_DeviceHidInterruptIn(). “In” depicts a message transferred from the device to host (into the host).

```
static usb_status_t USB_DeviceHidAction(void)
{...
        status = USB_DeviceRecvRequest(s_UsbDeviceComposite->deviceHandle, USB_HID_VIEWER_ENDPOINT_OUT,
                s_UsbDeviceHidViewer.buffer, USB_HID_VIEWER_OUT_BUFFER_LENGTH);
...
        switch(opcode)
        {
                case GET_VERSION:
                respPkt->pktdata.Version.USB_vmajor = USB_REV_MAJOR;
                respPkt->pktdata.Version.USB_vminor = USB_REV_MINOR;
                respPkt->pktdata.Version.ProductID = PRODUCT_ID;
                break;
...
        }
        status = USB_DeviceSendRequest(s_UsbDeviceComposite->deviceHandle,
                USB_HID_VIEWER_ENDPOINT_IN, (uint8_t *)respPkt, USB_HID_VIEWER_REPORT_LENGTH);
         return status;
}
```

Figure 6- Code Snippet from Appendix #6

```
static usb_status_t USB_DeviceHidInterruptOut(usb_device_handle deviceHandle,
usb_device_endpoint_callback_message_struct_t *event,
void *arg)
{
        if (s_UsbDeviceComposite->attach)
        {
                USB_DeviceHidViewerAction();
                return kStatus_USB_Success;
        }
         return kStatus_USB_Error;
}
...
```

Figure 7- Code Snippet from Appendix #7

4 Testing

4.1 Software

A handy USB debugging tool is made by Perisoft and is available for free online at their website. This tool allows you to send custom bytes to any USB device enumerated on your system and even capture the data sent across the USB bus.

4.2 Python

Python has a package called PyUSB which is an easy way of testing out connecting to your USB device.

4.3 Windows Visual Studio

Windows has a host of SDKs to create an application in Windows Visual Studio which can handle USB enumeration and sending USB commands to a device to debug.

5 Conclusion

Enabling USB on products is a very handy tool for many different teams that support the product. It allows customers to be able to update the product with new firmware, engineers can debug issues in development, and quality can diagnose error or issues on returns. It can be difficult to create the firmware, but a good programmer can design a user-friendly interface if they understand some USB basics.

Appendix #1 – HID Mouse Device Descriptor

```
uint8_t g_UsbDeviceDescriptor[] = {
        USB_DESCRIPTOR_LENGTH_DEVICE, /* Size of this descriptor in bytes */
        USB_DESCRIPTOR_TYPE_DEVICE,   /* DEVICE Descriptor Type */
        USB_SHORT_GET_LOW(USB_DEVICE_SPECIFIC_BCD_VERSION),
        USB_SHORT_GET_HIGH(USB_DEVICE_SPECIFIC_BCD_VERSION), /* USB Specification Release
        Number in
        Binary-Coded Decimal (i.e., 2.10 is 210H). */
        USB_DEVICE_CLASS,                   /* Class code (assigned by the USB-IF). */
        USB_DEVICE_SUBCLASS,                /* Subclass code (assigned by the USB-IF). */
        USB_DEVICE_PROTOCOL,                /* Protocol code (assigned by the USB-IF). */
        USB_CONTROL_MAX_PACKET_SIZE,        /* Maximum packet size for endpoint zero
        (only 8, 16, 32, or 64 are valid) */
        USB_SHORT_GET_LOW(USB_DEVICE_VID),
        USB_SHORT_GET_HIGH(USB_DEVICE_VID), /* Vendor ID (assigned by the USB-IF) */
        USB_SHORT_GET_LOW(USB_DEVICE_PID),
        USB_SHORT_GET_HIGH(USB_DEVICE_PID), /* Product ID (assigned by the manufacturer) */
        USB_SHORT_GET_LOW(USB_DEVICE_DEMO_BCD_VERSION),
        USB_SHORT_GET_HIGH(USB_DEVICE_DEMO_BCD_VERSION), /* Device release number in binary-
        coded decimal */
        0x01U,                  /* Index of string descriptor describing manufacturer */
        0x02U,                  /* Index of string descriptor describing product */
        0x00U,                  /* Index of string descriptor describing the
        device's serial number */
        USB_DEVICE_CONFIGURATION_COUNT,     /* Number of possible configurations */
        };
```

Appendix #2- HID Mouse Configuration Descriptor

```
uint8_t g_UsbDeviceConfigurationDescriptor[] = {

    USB_DESCRIPTOR_LENGTH_CONFIGURE, /* Size of this descriptor in bytes */
    USB_DESCRIPTOR_TYPE_CONFIGURE,   /* CONFIGURATION Descriptor Type */
    USB_SHORT_GET_LOW(USB_DESCRIPTOR_LENGTH_CONFIGURE +
USB_DESCRIPTOR_LENGTH_INTERFACE + USB_DESCRIPTOR_LENGTH_HID +
            USB_DESCRIPTOR_LENGTH_ENDPOINT),
    USB_SHORT_GET_HIGH(USB_DESCRIPTOR_LENGTH_CONFIGURE +
USB_DESCRIPTOR_LENGTH_INTERFACE + USB_DESCRIPTOR_LENGTH_HID +
             USB_DESCRIPTOR_LENGTH_ENDPOINT), /* Total length of data returned for this
configuration. */
    USB_HID_MOUSE_INTERFACE_COUNT,              /* Number of interfaces supported by this
configuration */
    USB_HID_MOUSE_CONFIGURE_INDEX,              /* Value to use as an argument to the
                                    SetConfiguration() request to select this configuration */
    0x00U,                                      /* Index of string descriptor describing this configuration */
    (USB_DESCRIPTOR_CONFIGURE_ATTRIBUTE_D7_MASK) |
      (USB_DEVICE_CONFIG_SELF_POWER <<
USB_DESCRIPTOR_CONFIGURE_ATTRIBUTE_SELF_POWERED_SHIFT) |
      (USB_DEVICE_CONFIG_REMOTE_WAKEUP <<
USB_DESCRIPTOR_CONFIGURE_ATTRIBUTE_REMOTE_WAKEUP_SHIFT),
    /* Configuration characteristics
       D7: Reserved (set to one)
       D6: Self-powered
       D5: Remote Wakeup
       D4...0: Reserved (reset to zero)
    */
    USB_DEVICE_MAX_POWER,         /* Maximum power consumption of the USB
                        * device from the bus in this specific
                        * configuration when the device is fully
                        * operational. Expressed in 2 mA units
                        *  (i.e., 50 = 100 mA).
                        */
    USB_DESCRIPTOR_LENGTH_INTERFACE, /* Size of this descriptor in bytes */
    USB_DESCRIPTOR_TYPE_INTERFACE,   /* INTERFACE Descriptor Type */
    USB_HID_MOUSE_INTERFACE_INDEX,   /* Number of this interface. */
    0x00U,                    /* Value used to select this alternate setting
                         for the interface identified in the prior field */
    USB_HID_MOUSE_ENDPOINT_COUNT,    /* Number of endpoints used by this
                          interface (excluding endpoint zero). */
    USB_HID_MOUSE_CLASS,          /* Class code (assigned by the USB-IF). */
    USB_HID_MOUSE_SUBCLASS,       /* Subclass code (assigned by the USB-IF). */
    USB_HID_MOUSE_PROTOCOL,       /* Protocol code (assigned by the USB). */
    0x00U,                    /* Index of string descriptor describing this interface */

    USB_DESCRIPTOR_LENGTH_HID,     /* Numeric expression that is the total size of the
```

```
                    HID descriptor. */
  USB_DESCRIPTOR_TYPE_HID,      /* Constant name specifying type of HID
                    descriptor. */
  0x00U, 0x01U,             /* Numeric expression identifying the HID Class
                    Specification release. */
  0x00U,                /* Numeric expression identifying country code of
                    the localized hardware */
  0x01U,                /* Numeric expression specifying the number of
                    class descriptors(at least one report descriptor) */
  USB_DESCRIPTOR_TYPE_HID_REPORT, /* Constant name identifying type of class descriptor. */
  USB_SHORT_GET_LOW(USB_DESCRIPTOR_LENGTH_HID_MOUSE_REPORT),
  USB_SHORT_GET_HIGH(USB_DESCRIPTOR_LENGTH_HID_MOUSE_REPORT),
  /* Numeric expression that is the total size of the
    Report descriptor. */
  USB_DESCRIPTOR_LENGTH_ENDPOINT, /* Size of this descriptor in bytes */
  USB_DESCRIPTOR_TYPE_ENDPOINT,   /* ENDPOINT Descriptor Type */
  USB_HID_MOUSE_ENDPOINT_IN | (USB_IN <<
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_SHIFT),
  /* The address of the endpoint on the USB device
    described by this descriptor. */
  USB_ENDPOINT_INTERRUPT, /* This field describes the endpoint's attributes */
  USB_SHORT_GET_LOW(FS_HID_MOUSE_INTERRUPT_IN_PACKET_SIZE),
USB_SHORT_GET_HIGH(FS_HID_MOUSE_INTERRUPT_IN_PACKET_SIZE),
  /* Maximum packet size this endpoint is capable of
    sending or receiving when this configuration is
    selected. */
  FS_HID_MOUSE_INTERRUPT_IN_INTERVAL, /* Interval for polling endpoint for data transfers. */
};
```

Appendix #3 - HID Mouse Report Descriptor

```
uint8_t g_UsbDeviceHidMouseReportDescriptor[] = {
  0x05U, 0x01U,          /* Usage Page (Generic Desktop)*/
  0x09U, 0x02U,          /* Usage (Mouse) */
  0xA1U, 0x01U,          /* Collection (Application) */
  0x09U, 0x01U,          /* Usage (Pointer) */

  0xA1U, 0x00U,          /* Collection (Physical) */
  0x05U, 0x09U,          /* Usage Page (Buttons) */
  0x19U, 0x01U,          /* Usage Minimum (01U) */
  0x29U, 0x03U,          /* Usage Maximum (03U) */

  0x15U, 0x00U,          /* Logical Minimum (0U) */
  0x25U, 0x01U,          /* Logical Maximum (1U) */
  0x95U, 0x03U,          /* Report Count (3U) */
  0x75U, 0x01U,          /* Report Size (1U) */

  0x81U, 0x02U,          /* Input(Data, Variable, Absolute) 3U button bit fields */
  0x95U, 0x01U,          /* Report Count (1U) */
  0x75U, 0x05U,          /* Report Size (5U) */
  0x81U, 0x01U,          /* Input (Constant), 5U constant field */

  0x05U, 0x01U,          /* Usage Page (Generic Desktop) */
  0x09U, 0x30U,          /* Usage (X) */
  0x09U, 0x31U,          /* Usage (Y) */
  0x09U, 0x38U,          /* Usage (Z) */

  0x15U, 0x81U,          /* Logical Minimum (-127) */
  0x25U, 0x7FU,          /* Logical Maximum (127) */
  0x75U, 0x08U,          /* Report Size (8U) */
  0x95U, 0x03U,          /* Report Count (3U) */

  0x81U, 0x06U,          /* Input(Data, Variable, Relative), Three position bytes (X & Y & Z)*/
  0xC0U,        /* End collection, Close Pointer collection*/
  0xC0U          /* End collection, Close Mouse collection */
};
```

Appendix #4 - HID Custom Report Descriptor

```
uint8_t g_UsbDeviceHidReportDescriptor[USB_DESCRIPTOR_LENGTH_HID_REPORT] = {
                        0x05, 0x01, // USAGE_PAGE (Generic Desktop)
                        0x09, 0x00,                     // USAGE (Undefined)
                        0xa1, 0x01,                     // COLLECTION (Application)
                        0x09, 0x3a,                     //  USAGE (Counted Buffer)
                        0x15, 0x00,                     // LOGICAL_MINIMUM (0)
                        0x26, 0xff, 0x00,               // LOGICAL_MAXIMUM (255)
                        0x85, 0x01,                     // REPORT_ID (1)
                        0x75, 0x08,                     // REPORT_SIZE (8)
                        0x95, 0x3F,                     // REPORT_COUNT (63)
                        0x81, 0x02,                     // INPUT (Data,Var,Abs)
                        0x09, 0x3a,                     // USAGE (Undefined)
                        0x85, 0x02,                     // REPORT_ID (2)
                        0x95, 0x3f,                     // REPORT_COUNT (63)
                        0x91, 0x02,                     // OUTPUT (Data,Var,Abs)
                        0x09, 0x3a,                     // USAGE (Counted Buffer)
                        0x15, 0x00,                     // LOGICAL_MINIMUM (0)
                        0x26, 0xff, 0x00,               // LOGICAL_MAXIMUM (255)
                        0xc0,                           // END_COLLECTION
};
```

Appendix #5 – USB API Structure

```
typedef struct _commandpktID {
        unsigned char repid;// The report id for hid to and from pipes.
        unsigned char pktcmd;// the ID of the command this is a response to.
        union {
                unsigned char   payload[MAX_PKT_PAYLOAD]; // Unformatted data.
                DATAPKT         datapkt;      // Data to write to flash (bootloader only)
                VERSIONPKT      Version; // Version id's
                unsigned char   value;    // Use for commands that need to send 1 byte of data
  } pktdata;
} IDCOMMANDPKT, *PIDCOMMANDPKT;

typedef struct _datapkt{
  unsigned short bytecnt;             // # of valid bytes in data buffer to write to Flash
  unsigned char databytes[MAX_UPDPKT_LEN]; // Data to write to Flash
} DATAPKT, *PDATAPKT;

typedef struct _versionpkt {
        unsigned char USB_vmajor;       // Devices major version number (usually hardware version)
        unsigned char USB_vminor;       // Devices minor version number (usually software version)
        unsigned char ProductID;                                        // Used to identify Wraps from one another
} VERSIONPKT, *PVERSIONPKT;
```

Appendix #6 – USB API Parsing

```
static usb_status_t USB_DeviceHidAction(void)
{
	usb_status_t status = kStatus_USB_Success;
	static uint8_t opcode = 0;
	uint8_t *pBytes;
	IDCOMMANDPKT *pkt = (IDCOMMANDPKT *)(s_UsbDeviceHidViewer.buffer);
	IDCOMMANDPKT *respPkt = (IDCOMMANDPKT *)(g_outBuf);

	status = USB_DeviceRecvRequest(s_UsbDeviceComposite->deviceHandle,
		USB_HID_VIEWER_ENDPOINT_OUT,
		 s_UsbDeviceHidViewer.buffer,
		USB_HID_VIEWER_OUT_BUFFER_LENGTH);

	//Clear Buffer
	memset(g_outBuf, 0xFF, sizeof(g_outBuf));
	//Get Report ID (Sending or Receiving)
	respPkt->repid = GET_REPORT_ID;
	//Get the API Cmd
	opcode = respPkt->pktcmd = pkt->pktcmd;

	if (pkt->repid == SEND_CMD_REPORT_ID)
	{
		switch(opcode)
		{
		case GET_VERSION:
			respPkt->pktdata.Version.USB_vmajor = USB_REV_MAJOR;
			respPkt->pktdata.Version.USB_vminor = USB_REV_MINOR;
			respPkt->pktdata.Version.ProductID = PRODUCT_ID;
			break;
		case BRIGHTNESS_SET:
			Set_Brightness(pkt->pktdata.value);
			break;
		case BRIGHTNESS_GET:
			respPkt->pktdata.value = Get_Brightness();
			break;
		case PC_UPDATE_GET_ACTIVE_FW:
			tmp = PC_Get_Firmware_Id(respPkt->pktdata.datapkt.databytes);
			respPkt->pktdata.value = tmp ? UPD_FAILED : UPD_SUCCESS;
			break;
		case PC_UPDATE_START:
			tmp = PC_Update_Init();
			respPkt->pktdata.value = tmp ? UPD_FAILED : UPD_SUCCESS;
			break;
		case PC_UPDATE_PKT:
			tmp = PC_Fill_Buffer(pkt->pktdata.datapkt.databytes,
```

```
                                                    pkt->pktdata.datapkt.bytecnt);
                        respPkt->pktdata.value = tmp ? UPD_FAILED : UPD_SUCCESS;
                        break;
                case PC_UPDATE_END:
                        tmp = PC_Validate_Firmware(*pkt->pktdata.datapkt.databytes);
                        respPkt->pktdata.value = tmp ? UPD_FAILED : UPD_SUCCESS;
                        break;
                case CPU_RESET:
                        //Return cpu reset
                        reset_wdog();
                        break;
                case DEBUG_MODE_SET:
                        g_debug_lock = pkt->pktdata.value;
                        break;
                case DEBUG_MODE_GET:
                        respPkt->pktdata.value = g_debug_lock;
                        break;
                case ERROR_REPORT:
                        respPkt->repid = VUZIXAPI_GET_REPORT_ID;
                        respPkt->pktcmd = ERROR_REPORT;
                        respPkt->pktdata.payload[0]  = Get_Error_State();
                        break;
                default:
                        respPkt->pktcmd = UNKNOWN_CMD;
                        break;
                }

        status = USB_DeviceSendRequest(s_UsbDeviceComposite->deviceHandle,
                                       USB_HID_VIEWER_ENDPOINT_IN,
                                       (uint8_t *)respPkt,
                                       USB_HID_VIEWER_REPORT_LENGTH);
        return status;
}
```

Appendix #7 – USB API Communication

```
static usb_status_t USB_DeviceHidInterruptOut(usb_device_handle deviceHandle,
        usb_device_endpoint_callback_message_struct_t *event,
        void *arg)
{
        if (s_UsbDeviceComposite->attach)
        {
                USB_DeviceHidViewerAction();
                return kStatus_USB_Success;
        }
          return kStatus_USB_Error;
}

static usb_status_t USB_DeviceHidInterruptIn(usb_device_handle deviceHandle,
        usb_device_endpoint_callback_message_struct_t *event,
        void *arg)
{
        if (s_UsbDeviceComposite->attach)
        {
          g_EPInPending = 0;
          return kStatus_USB_Success;
        }
          return kStatus_USB_Error;
}
```

Appendix #8 – Links

Example Code from NXP-

https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE

Article Describing USB Protocol-

https://www.beyondlogic.org/usbnutshell/usb5.shtml

Usage Tables for Report Descriptors-

https://usb.org/document-library/hid-usage-tables-15

PyUsb Page-

https://pypi.org/project/pyusb/

BusHound Tool from Perisoft-

https://perisoft.net/bushound/