Project 4 required me to implement a user space driver for the Chompstick device. The first major component of implementing this driver was retrieving the data from the USB connection. This was done using libusb. In order to use libusb, libusb must be initialized using **libusb_init.**

Afterwards **libusb_device_handle** is acquired bypassing the product id and vendor id to **libusb_open_device_with_vid_pid**. With this handle we can detach the device from the kernel and claim the usb interface using **libusb_claim_interface**. At this point we are ready to start reading data using **libusb_interrupt_transfer.** This function requires the handle acquired earlier, an endpoint, a buffer for the data, a memory address to store the value of bytes received., and the expected number of bytes to be received. I acquired the information on the endpoint (**0x81**), by using **lsusb -v -d GA1A:BA17** in the terminal. The data received was always a single byte, stored as a char. I casted the char to an int to be able to acquire the information on the 8 bits. I converted the int into 5 bits and stored the 5 bits into an array for processing.

Another major component of creating the userspace driver was re-routing the data as a conventional joystick that Linux can understand. This was done by using the **input.h** and **uinput.h** libraries. Our version of Linux does not have access to the **UI_DEV_SETUP** ioctl, so the legacy **struct user_dev** must be initialized. We make also make a call to **open**, opening **"/dev/uinput"** in **write only** and **nonblocking** mode. The **ioctl** function is called with the flag **UI_SET_EVBIT**. This is done twice, once for **EV_ABS** and for **EV_KEY.** Then **ioctl** is called 3 more times with the **UI_SET_KEYBIT** and **UI_SETABSBIT**(twice) flags. This process successfully sets up button 0 , the x-axis, and the y-axis. We then **write** the **user_dev** struct to the file before a final call to **ioctl** with the **UI_DEV_CREATE** flag. At this point, we are setup to pass events to input. These input events will appear as a joystick and been readable from **/dev/input/js0**.

After final processing of the data, we can pass the events to the OS as an **input_event.** I wrote a helper function called **emit** to do this. A struct **input_event** has a **type**, **value,** and **code**. The **type** for this joystick would be **EV_ABS**, **EV_SYN,** or **JS_EVENT_BUTTON. EV_ABS** is the type for absolute changes in axis values, **JS_EVENT_BUTTON** reports the state of the button, and **EV_SYN** is used to synchronized data. The **code** for the axis information would b**e ABS_X** for the x-axis and **ABS_Y** for the y-axis. In the case of the button it was **BTN_0**. The code **SYN_REPORT** was needed for **EV_SYN** type events. I would send the SYN reports twice for each processing the of the data. Once after updating axis information and once after updating the button information.

This process would be done repeatedly until the device disconnected. Afterwards calls to **libusb_release_interface**, **libusb_close**, and **libusb_exit** were made to release the device and close libusb. At that point, I would then use **ioctl** with flag **UI_DEV_DESTROY** to destroy the virtual joystick and then close the file.

Testing was done by running chompapp, and trying all combinations of directions and button. By running **jstest /dev/input/js0**, I could confirm that the values were to specification.

To summarize, Project 4 required opening using the library **libusb** to claim a device. In the meantime, **uinput.h** provides the ability to create virtual joystick. Once the device was claimed data could be processed. After the processing of this data, the data would be stored in **input events** and then re-routed as joystick information.