Abstract

Our project has two main parts; part one consists of reverse engineering the USB protocol of the X-Plorer guitar controller manufactured for use with the Microsoft©Xbox 360, and part two is implementing a USB driver to make the guitar available to applications running on top of the Linux kernel, with the specific goal of being able to use the X-Plorer guitar with the open-source game Frets on Fire (Kyostila, 2007).

Introduction

Proprietary game consoles have offered games along with peripherals to make the game more immersive since Duck Hunt was released for the NES. Due to the nature of consoles, these peripherals have been tied to the platform on which they originated, damning them to 15 minutes of utility coinciding with the period of popularity a given console experiences.

Despite their numerous sins, Microsoft made a unique and enlightened choice by foregoing the standard arbitrary proprietary connector and using an existing standard, USB, for its wired peripherals. This allowed Microsoft to sell wired gamepads for the PC market as well as for its console. Unfortunately, though they were kind enough to provide a driver for the Windows XP and Vista operating systems, they neglected to provide a similar driver for the Linux operating system, which is counter-intuitive as presumably they get just as much money from a Linux user purchasing an Xbox 360 peripheral as they do from the corresponding Windows user.

One of the peripherals available for the Xbox 360 is the X-Plorer guitar made available for Red Octane’s Guitar Hero 2. This guitar is used to play a rhythm-based game in which the player tries to match chord structures and strum patterns to those being presented on the screen. An open-source clone of Guitar Hero called Frets on Fire features similar game play to Guitar Hero, but has the added benefits of a community who creates additional songs for the game as well as the ability to create our own songs as the inclination moves us. For comparison, there are under 100 songs within the Xbox 360 Guitar Hero 2 game, while there are over 1500 songs available for Frets on Fire.

Project Overview

Our project consists of writing a USB driver for the X-Plorer USB guitar for Linux, specifically targeting the 2.6.20.1 kernel. This kernel version was chosen as it is the most recent version of the kernel supported by the Cross-Referencing Linux project (Gleditsch and Gjermshus, 2007). Our prime interest in getting this game working is to use it with the freely available game Frets on Fire (Kyostila, 2007). This game is similar to the previously mentioned Guitar Hero 2 except that instead of a console it is targeted for the PC, and instead of the guitar controller, a keyboard would be used in its place.
Figure 1: The X-Plorer USB guitar. As can be seen, the guitar has 5 fret buttons, a up-down strum bar, an analog whammy bar, start button, select button, the X-button, and the D-Pad directional buttons. Additionally, two accelerometers are inside the guitar casing.

Detailed Functionality

All aspects of functionality including the fret buttons, strum bar, D-pad, start button, select button, whammy bar, and tilt sensor are supported by our driver and supported within Frets on Fire as necessary.

The analog components of the guitar, specifically the whammy bar and tilt sensor, are not used in Frets on Fire, and so another test was necessary to demonstrate this portion of the driver’s functionality. We can demonstrate the analog components of the driver’s functionality in two different ways. First, a test application prints the events that are produced by a given device in the kernel input subsystem. Second, we adapted the analog components to be used as with a mouse driver, so we are able to demonstrate functionality through mouse navigation.

Background

In this section we cover the necessary background knowledge needed to get our driver working along with the steps we took. We give an introduction to drivers in Linux, the USB stack, and the input subsystem in Linux. Then we cover the reverse engineering of the guitar and the implementation of the mouse driver.

Drivers in Linux

One of the functions of a general-purpose operating system is to provide device abstraction and resource management. Early versions of Linux provided limited, low-level functionality to reserve ranges of I/O addresses or install an interrupt handler for an IRQ line. This was sufficient for the hardware of the time, which was usually difficult to program and had little opportunity for sharing code between drivers. Current common device connection technologies like PCI allow more common code between drivers, so Linux now has a stronger device driver model with layers of abstraction. In addition to exposing a device to the system, drivers should typically handle power management, resource allocation, and, where relevant, hot-plugging. (Bovet and Cesati, 2006)

A device driver consists of a struct device_driver with fields for identifying and managing the driver, as well as function pointers that point to implementations of primitive device operations. These functions include:

- probe(): See if the driver can handle the device.
- remove(): Called when a device is removed.
- shutdown(): Called when a device is powered off.
- suspend(): Called when a device is put in a low-power state.
- resume(): Called when a device returns from a low-power state.

device_driver objects are registered with the kernel with device_register. device_driver objects are usually embedded in a larger structure for more specific drivers, or more specifically-purposed structures are registered with the device’s subsystem, which has either already registered a device_driver, or will call device_register on the driver’s behalf. For example USB drivers are described by usb_device structures and are registered with the USB subsystem using usb_register.
**Linux Device Driver Model**

Linux's device driver model manages a hierarchy of devices, drivers, buses, and device classes. One role of this hierarchy is to ensure that devices change power state in the proper order. For example, it is important to power off a hard disk before powering off its controller. This hierarchy is implemented using the *kobject* infrastructure. A view of the kobject hierarchical relationships is presented by the *sysfs* filesystem. Each *kobject* structure corresponds to a directory within */sysfs* (Bovet and Cesati, 2006). Each *kobject* contains a pointer to a *kobj_type* kobject type descriptor that defines *sysfs* file operations to show and store the attributes of the kobject through userspace (Gleditsch and Gjermshus, 2007).

**udev**

*udev* is a set of userspace tools that manages the */dev* filesystem on modern Linux systems. Before *udev*, the */dev* filesystem was manually populated with device files with nearly every conceivable major and minor number combination. Instead, *udev* scans */sysfs/class* on boot for *dev* files, which contain the major and minor numbers for a device. *udev* then uses a database of rules to assign filenames and create symbolic links. Now, */dev* contains only device nodes for devices that actually exist. *udev* rules can match on *sysfs* attributes, and can be a powerful tool to create persistent device names. To support hot-plugging, when a device is plugged in to the system, the kernel executes a userspace hotplug program (*/sbin/hotplug* by default), which usually runs *udev* tools to create a device file in */dev* (Bovet and Cesati, 2006).

**USB Background**

USB was originally conceived to replace a plethora of older peripheral buses, but has since grown into an ubiquitous, cheap peripheral bus capable of handling everything from printers to hot plates and refrigerators. Its ubiquity can be attributed to the emphasis on low cost and ease of use, with support for powering devices, hot-plugging, and automatic device detection and configuration.

USB is organized into a tiered-star topology, similar to Ethernet. In order to reduce complexity, there are only a handful of connector types, and they are organized in such a fashion as to discourage users from accidentally creating cyclical USB networks. USB is also host-centric, meaning devices cannot put data onto the bus unless allowed to do so by the host controller. This further reduces the complexity of the devices, and simplifies hot-plugging.

While USB consists of several layers of protocols, the device driver developer generally only needs to know the top few, as the lower layers are often implemented in hardware or the host controller firmware. The detail most pertinent to a driver developer is how a USB device is built out of lower-level abstractions. Specifically, it is necessary to understand how configurations, interfaces, endpoints, and pipes interact, as well as the transfer types an endpoint can handle.

An endpoint is essentially a data source or sink on the slave device. The host controller talks to a specific endpoint instead of a device. Each endpoint is associated with a direction and a transfer type. A device can have multiple endpoints with different transfer types, in order to offer diverse functionality. The four transfer types are:

- **Control** - Used for device configuration messages.
- **Interrupt** - Used for interrupt-paradigm transfers; offers guaranteed latency and automatic error detection.
- **Bulk** - Used for large, time-insensitive data transfers.
- **Isochronous** - Offers bounded latency and guaranteed access to a certain amount of bandwidth. Often used for audio or video streams.

An endpoint also has settings associated with it, which control parameters such as bandwidth...
reservation. These settings are dependant upon endpoint type.

Interfaces are groupings of endpoints. An interface often encapsulates the endpoints for a specific function of a device—for example, a USB speaker system could have an interface for the audio data and an interface for an attached keypad. Under Linux, each interface can actually be handled by a separate driver. Different settings can be associated with an interface, which will simply reconfigure the endpoints contained by the interface.

Configurations are groupings of interfaces, and correspond to whole-device states, such as having a configuration for normal operation and a configuration for firmware flashing. However, they are rarely used and not very well supported.

**Linux USB Interface**

Since USB is so ubiquitous, the Linux kernel contains a subsystem named the USB core to encapsulate most of the low-level complexity. Developers of USB device drivers can use the USB core to handle most of the low-level details, and concentrate on handling the messages to and from their device. This corresponds to the “lower half” of the driver being done already, which simplifies driver development quite a bit.

The USB Core API operates in a similar fashion to other kernel core facilities; it provides a register function, as well as a few key callbacks. In the USB Core `usb_device` structure, the two required callbacks are `probe` and `disconnect`, which roughly correspond to device initialization and disconnect. Additionally, there are three optional callbacks: `ioctl`, `suspend`, and `resume`. These were not necessary to implement for the USB X-Plorer guitar.

The `probe` callback takes two parameters, of type `struct usb_interface` and `struct usb_device_id`. The former represents the interface the driver is being probed against; recall that drivers bind to `interfaces`, not devices. The latter struct contains information about the device being probed, including the ID. The responsibility of the `probe` function is to determine if the parameters correspond to a device it can handle, and if so, initialize it. If it is probed against an incompatible device or encounters an error, `probe` should return a negative number. The `disconnect` callback should clean up any resources allocated by a successful `probe` callback.

While simply plugging and unplugging USB devices is great fun, another common operation is to send messages between host and device. The USB Core provides two mechanisms to do this. First, there are the synchronous functions `usb_bulk_msg` and `usb_control_msg`, which handle most of the details of the transfer. Unfortunately, there is no such simple interface for receiving messages over an interrupt endpoint. As such, interrupt and isochronous transfers are forced to use asynchronous USB Request Blocks, or URBs.

**The Linux Input Subsystem**

The ability for an external device to exert control (act as an input device) over the computer is an important feature. Because our ultimate goal was to use the guitar as an input device, an understanding of how Linux input subsystems function as well as how they interact with device drivers like the one we created is required.

There are three main elements of the Linux input subsystem: device drivers, the input kernel code, and device handlers. The writer of a driver should understand how these three components interact and work together to allow a device to become an input device. The relationship between the elements is shown in Figure 3. Device drivers are responsible for low level interaction with the hardware. They format device output into the core defined API (events) and the input core events into low-level device-understandable data. In other words, a driver acts as the custom interface from the Linux input core to the device and vice-versa. The last element in the equation is the input device handler, of which there are many. These handlers propagate the input events to user space. Handlers act as another sort of translator, converting data between the standard event format used by the input core and the spec-
specific user API. For example, mousedev translates input events to emulate a standard PS/2 mouse. Since the guitar is most akin to a joystick, we report events that the joydev input handler will deliver to a /dev/input/jsX device file.

A discussion regarding how device drivers interact with the input core is necessary. All interaction occurs using event structures, namely struct input_event. It is defined in input.h along with many other useful core input subsystem components. Every input device must be registered with the kernel as being an input device. This is done using input_register_device. Conversely, an input device is unregistered using input_unregister_device. Both of these functions take a pointer to a struct input_dev. A struct input_dev can be allocated using input_allocate_device which will return a pointer to a new struct input_dev. This structure is populated by the device driver when the guitar is connected (in the probe function). The fields within this structure specify the operations of this input device and its specific capabilities to a standard input core event format. The probe function also registers the device.

The operation of actually sending events from the driver to the input core is done using input_event or a wrapper such as input_report_key for key or button presses, or input_report_abs which is used for the analog guitar sensors such as the whammy bar and accelerometers. When a button is depressed, the driver uses input_report_key to tell the input core that this device has completed an action that can be interpreted as the pressing of a key. The specific key is reported as a parameter. Key names are defined in input.h. When the guitar is loaded in mouse mode, input_report_rel is used, as relative changes in position of the mouse pointer are used by applications to determine cursor motion. After all events for this USB interrupt cycle are registered, input_sync is called. This function informs the event handler that the device has transmitted an internally consistent set of data (Hards, 2003) to the input core.

The interaction between the input core and the input handler is fairly straightforward. A handler is able to specify which events it wants the input core to send to it. For example, the joydev handler only receives events that are pertinent to joystick style devices.

The necessary steps from a device being plugged in to a device’s input action being received by the input handler are summarized below:

1. Create a struct input_dev using input_allocate_device and populate it with the proper functionality.
2. Call input_register_device passing a pointer to the struct input_dev. Device is now registered with the input core.
3. Within the device driver’s interrupt callback function, device actions are sampled and
events are created and sent to the input core using `input_report_key`, `input_report_abs`, or `input_report_rel`.

4. The input core propagates these events to the handlers which act upon them accordingly.

The Linux input subsystem is straightforward for the most part and provides a standard way for a hardware device driver writer to make his device work as an input device. Almost any piece of hardware can be made to work with Linux, provided the knowledge outlined above.

**Reverse Engineering of X-Plorer**

The X-Plorer guitar operates over the well-documented USB protocol. However, the format of the data within each USB data packet is not documented publicly. To produce a working driver, we needed to reverse-engineer the protocol so that we could properly decode the USB packets in the driver. To do this, we used the Windows-based open-source application `USB Sniffer for Windows` (Valette et al., 2007). This application connects to a USB device on the system and prints out the USB header and raw data of each USB transfer. For the X-Plorer guitar, interrupt transfers are used to send the state data for the guitar. Interrupt polling occurs a thousand times per second for USB, so this very quickly produced a lot of data. `USB Sniffer for Windows` did not have the capability to remove duplicate packets or to do advanced filtering, so we had to resort to not changing the state of the guitar while doing the capturing. Our analysis cycle followed the pattern:

1. Put the guitar into a known state (e.g., horizontal while pressing the green button).
2. Begin capturing USB packets.
3. After about a second, stop capturing USB packets.
4. Record the raw data from one of the interrupt packets into a file.

After manually iterating through a variety of useful guitar states, we analyzed the data. By correlating the state of the guitar when the data was captured with the raw data bit-stream, we
were able to figure out the protocol used by the guitar.

An additional resource that proved helpful was the XBox 360 Controller driver for Mac OS X (Munro, 2007). We were able to observe how playing with the X-Plorer guitar influenced the various input values using the test application provided with the OS X driver. Particularly useful was the ability to watch how rotating and tilting the guitar changed the analog values in the data packet. We were able to use this knowledge to better understand how to use the accelerometers for use as a mouse.

**Guitar Protocol**

The X-Plorer guitar uses the interrupt transfer mechanism of the USB protocol for transmitting state data from the guitar. Consequently, new guitar state data is available every millisecond. Each interrupt data packet from the guitar contains 20 bytes of data (after stripping off the USB header). As we expected, the data format primarily consists of a bitmask containing the state of each individual button as well as multi-bit fields for any analog data (such as the accelerometer data or the whammy bar).

Based on the arrangement of the analog data bytes, it appears that the microcontroller inside the X-Plorer guitar is a 16-bit little-endian microcontroller. As a result, multi-byte values read in from the guitar must be endian-swapped as necessary for the particular system architecture.

Figure 4 shows the bit-layout of the guitar status update packet. Some of the data fields are always zeroed out. We believe that this is due to the fact that the guitar data packet probably follows the same data format as all XBox 360 controllers. Due to the X-Plorer guitar not having all the buttons of a full controller, some of the data is simply zeroed out.

**Mouse Implementation**

Because the X-Plorer guitar was not designed to be used as a pointing device, it was an interesting exercise to get it to operate as one in the X Window System. We found that the most effective way to hook into the mouse device system was to use the Linux event device (evdev) interface. This required a couple of steps. First, we registered the guitar as a relative event input (EV_REL) with the event system during device probe. Second, we pushed relative input events to the event system based on the actions of the guitar during the USB interrupt callback routine. The mouse system then picked up these events allowing the guitar to be used as a mouse. All of the mouse code that we incorporated into the driver is optional so that plugging in the guitar to the system does not hijack a much more sensible pointing device plugged in to the system (i.e., a mouse). The guitar can be enabled as a pointing device by passing the module parameter `mouse=1` to the `insmod` command.

We initially believed that the guitar contained a 3-axis accelerometer with each axis correlating to an analog section of the data packet shown in 4. However, after some experimentation, we discovered that this was not the case. Instead, the guitar seems to have a 2-axis accelerometer with the 2-byte tilt value being a cooked value from the raw accelerometer data. We finally decided to use the guitar as a joystick-style pointing device. Holding the guitar straight up allows normalization of the raw accelerometer values and simplification in the calculation of the x-axis and y-axis velocities for the mouse pointer. When the guitar is in the upright position (with the guitar neck pointing away from the ground), the two raw accelerometer values center on the value of 0x80. By subtracting this offset, moving the guitar like a joystick allows the mouse to move appropriately on the x and y axes. To provide a reasonable rate of mouse velocity, the accelerometer data is divided by eight. Additionally, to avoid rounding issues due to division, the remainder bits are accumulated and added to the next value set from the accelerometers. This smooths the output of the mouse pointer and allows it to be used with higher precision. To make the guitar less touchy as an input device, a dead zone is introduced around the
Figure 4: The data transfer protocol of the X-Plorer guitar. The up and down strum buttons on the guitar are equivalent to the up and down D-Pad buttons. Header information is shown in orange, button information is shown in green, analog data is shown in yellow, and zero fields are shown in gray.

region where the guitar is pointing approximately straight up.

Implementation

Probably the easiest way to start writing a device driver is to look at an example. We started by looking at the usb-skeleton driver provided in the Linux kernel. While this is a good start to learn the basics of writing a driver, it only has examples of using bulk transfer endpoints. Since the X-Plorer guitar uses interrupt endpoints, this wasn’t useful to us after we had initialized the guitar in the probe function. Next, we found the generic usbmouse driver (also included with the Linux kernel). This proved to be a much more valuable resource when dealing with interrupt transfers.

Module Initialization

First thing’s first: the module has to be initialized and the driver has to be registered to receive the probe requests from the USB subsystem. The first thing to do is to provide a table of information for different devices that the driver supports. There are some macros, such as USB_DEVICE, to make constructing this table easier. In our case, our driver only supports one device so the table is small. After the table is created, the MODULE_DEVICE_TABLE macro can be used to export the list of devices to user-space (e.g., to allow hotplug to load your driver when a device that it supports is plugged in).

```
#define GH_VENDOR_ID 0x1430
#define GH_PRODUCT_ID 0x4748

/* device types */
static struct usb_device_id usb_gh_id_table [] = {
    { USB_DEVICE(GH_VENDOR_ID, GH_PRODUCT_ID) },
    {} /* Terminating entry */
};
```
The next step is to register the driver with the USB subsystem so that the driver will be probed when a supported device is plugged in. First, a `struct usb_driver` is filled in with pointers to the drivers `probe` and `disconnect` functions and a pointer to the table of supported devices. Then, in the module’s `init` function, a call to `usb_register` registers the driver with the USB subsystem. The driver’s `probe` function will now be called when a supported device is plugged in.

```c
static struct usb_driver usb_gh_driver = {
    .name = "usbguitar",
    .probe = usb_gh_probe,
    .disconnect = usb_gh_disconnect,
    .id_table = usb_gh_id_table,
};
```

Notice that the `id_table` member of the `usb_driver` struct is the same table that was constructed above. The `probe` and `disconnect` members are pointers to functions that are required for all USB drivers.

### The probe Function

The `probe` function is where most of the magic happens in the driver. The first responsibility of the `probe` function is to determine if it can control the device that it’s being probed for. This usually just means checking the number, type, and direction of endpoints as shown below.

```c
interface = intf->cur_altsetting;
if (interface->desc.bNumEndpoints != 2)
    return -ENODEV;
endpoint = &interface->endpoint[0].desc;
if (!usb_endpoint_is_int_in(endpoint))
    return -ENODEV;
```

Checking that the interface has two endpoints ensures that the driver only connects to the first of the X-Plorer’s four endpoints. Once we have established that we’re connected to a device that the driver can interact with, it’s time to allocate and fill in the following data structures:

1. **struct usb_gh**: Defined by our driver, it contains device-specific information. Other structures such as `struct input_dev` have a private field which allows drivers to store a pointer to device-specific data so it’s available in callback functions.

2. **struct input_dev**: This structure is used to register the device as an input device and emit input events. It’s allocated with `input_allocate_device`. Setting its `evbit` field tells the input subsystem what type of device it is. For example, setting `EV_ABS` will register the device as a joystick. For each of the different types of events, there are additional fields to set to indicate which specific events (e.g., specific buttons and axes) the device will emit. For `EV_ABS` events (absolute axes), there are fields to set the limits, zero, and deadband of the axes. Below is an example of setting these parameters for a joystick.

```c
input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
ininput_dev->keybit[LONG(BTN_GAMEPAD)] = 
    BIT(BTN_A) | BIT(BTN_B) | BIT(BTN_C);
ininput_dev->absbit[0] = BIT(ABS_X) | BIT(ABS_Y);
ininput_dev->absmin[ABS_X] = 0x80;
ininput_dev->absmax[ABS_X] = 0x7f;
ininput_dev->absfuzz[ABS_X] = 0;
ininput_dev->absflat[ABS_X] = 0;
```

Additionally, the `struct input_dev` requires pointers to `open` and `close` functions which are called when the device is opened or closed by a program.

3. **struct urb and u8 ***: The URB and its associated buffer are what let the driver talk to the device. URPs are Linux’s way to talk to USB devices. They are essentially a request to talk to a device and they are submitted to the USB core which handles talking to the device and calls a callback function when
events happen related to the request, such as data being ready. In the case of interrupt UR Bs, the same URB is used for more than one transfer because interrupt UR Bs happen at such a fast rate. When an interrupt URB is submitted, it will call the driver’s callback function at an interval specified in the data structure. Generally, this interval is set to the endpoint’s bInterval value. UR Bs are allocated with usb_alloc_urb. There are many functions in the Linux kernel to help fill in a struct urb, including usb_fill_int_urb.

```c
    guitar->data = usb_buffer_alloc(dev, maxp, GFP_ATOMIC, &guitar->data_dma);
    guitar->irq = usb_alloc_urb(0, GFP_KERNEL);
    usb_fill_int_urb(guitar->irq, dev, pipe, guitar->data, maxp, usb_gh_irq, guitar, endpoint->bInterval);
    /* enable DMA */
    guitar->irq->transfer_dma = guitar->data_dma;
    guitar->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

At this point, the URB is ready to be submitted to the USB subsystem, but it’s not submitted in the probe function. Rather, it is submitted when the device is opened. After the URB is submitted, the callback function is called every interval and the data is available in the buffer that was allocated.

Related Work

Before starting this project we searched for an existing driver that would have support for the X-Plorer guitar and were unable to find anything. The xpad.c code for more general Xbox peripherals we were able to find was from an earlier version of the kernel that had been forked with the purpose of getting Linux running on an Xbox.

Other related drivers include any portion of the input subsystem, we heavily referenced the USB mouse driver in particular in the implementation of our driver. Originally we had used the provided USB skeleton driver as a starting point for our driver, but we found that since the USB skeleton driver was based around the sending and receiving of bulk messages, it wasn’t an appropriate resource for our purposes.

There is a driver for the X-Plorer on the Mac OS X that was released after we chose our project for this quarter. Further there are several joystick and gamepad drivers which already existed on Linux, which we didn’t think to look at until we had completed the project.

Conclusion

The end result of this project is a working USB device driver for the X-Plorer guitar. We’ve also developed several testing programs and extended functionality such that we could use it as a mouse driver. We were able to reverse engineer an undocumented protocol in order for our driver to function correctly. We were exposed to a huge amount of information pertaining to the USB stack, the input subsystem, and device drivers in particular. Linux users will now be able to utilize this USB device without booting into Windows and being forced to use a proprietary driver.

Future Work

Though most of the work on this particular driver is complete, there are other USB peripherals which would use a similar interface and could be incorporated into our driver. Specific examples include the Microsoft Steering Wheel, the Madcatz old school gaming joystick, and third party guitars as they become available, which they surely will.
References


