Driver Writing With Python - Stealth-x

Stealth-

Joysticks and Python, what could be better?

Hey guys, today I'm going to outline my experience writing an interface for the USB joystick I have. My reason for starting this in the first place was a mix of curiosity and of my frustration with the lag or lack of functionality I've dealt with when trying to use a lot of other joystick mappers. A lot of games don't support joysticks out of the box, so the most common method of dealing with this is, of course, to map the joystick actions to simulate key presses and mouse movements. So in the end, this isn't so much a "driver" as a "converter".

I'm going to assume you already have you joystick device up and working, and that you know the location of the device. For me, and likely a lot of you, the device will be located at /dev/input/js0. The exact model I used for this was the USB Logitech Dual Action, but the theory should apply to anyone looking to make their own as well.

Alright, so, the first thing we need to do is find out what type of information is coming from the joystick. If we run a quick 'cat /dev/input/js0', and move/press the joystick buttons we should see a nice bunch of complete jibberish flying by. Not to worry though, we can make sense of all that. Let's start out by making our initial python code:

```
import sys
pipe = open('/dev/input/js0','r')
while 1:
        for character in pipe.read(1):
                sys.stdout.write(repr(character))
                sys.stdout.flush()
```

Now, if we run that program, we see what looks like even more gibberish. Great! What's actually happening, though, is that the device file /dev/input/js0 can be opened and read like a file. So we open the device file in read mode, then continue to pipe the output to the terminal with repr(). Repr() is a function that returns the exact literal of a object. If it's a string, we will see quotation marks around it. If it's a list, we will see the list container marks ([,]) around it. It also replaces funky characters (like the ones we saw earlier) with their hex value (things like \x82). So, in summary, we read through one character at a time and print the literal version of the string.

Next comes the part that could, and likely will, vary depending on your controller. To interpret the data, we need to find patterns in it. We're just going to worry about the buttons for now, so if you have joysticks or a directional pad just ignore those. What I did for mine was I used the script above, pressed enter a few times to clear some space on the terminal, and pressed down a button. I then looked at the data, pressed enter again to start a new line, and let go of the button. After trying a few more buttons I got data something like this:

```
'\xf0''\xe1''\xf2''\x00''\x01''\x00''\x01''\x00'
'P''\xed''\xf2''\x00''\x00''\x00''\x01''\x00'

'\x18''\xf3''\xf2''\x00''\x01''\x00''\x01''\x01'
'\xd0''\xf7''\xf2''\x00''\x00''\x00''\x01''\x01'

'\x84''\xfe''\xf2''\x00''\x01''\x00''\x01''\x02'
'd''\x02''\xf3''\x00''\x00''\x00''\x01''\x02'

'''\x06''\xf3''\x00''\x01''\x00''\x01''\x03'
'\x94''\n''\xf3''\x00''\x00''\x00''\x01''\x03'

'\xac''$''\xf3''\x00''\x01''\x00''\x01''\x04'
'\xac''"''\xf3''\x00''\x00''\x00''\x01''\x04'

'\x94''.''\xf3''\x00''\x01''\x00''\x01''\x05'
'\x0c''2''\xf3''\x00''\x00''\x00''\x01''\x05'
```

What I noticed was that each action press/release is 8 bytes long (hex values look like more than one character but are actually only one). For buttons, the only values that appear to be relevant are the last byte and fifth byte. The fifth byte is \x01 if the button is being pressed down and \x00 if it's being released, while the last byte is a indication of what button is being pressed/unpressed.

If you go ahead and try out the other devices on the gamepad like the joystick(s) and the directional pad, you'll see that things do indeed get a bit more complicated than that. Before we worry about that though, I think it would be much easier to work with these values if we displayed them a bit nicer and made it so we didn't have to press enter all the time. I updated my python script to look like this:

```
import sys
pipe = open('/dev/input/js0','r')
action = []
spacing = 0
while 1:
        for character in pipe.read(1):
                action += [character]
                if len(action) == 8:
                        for byte in action:
                                sys.stdout.write('%02X ' % ord(byte))
                        spacing += 1
                        if spacing == 2:
                                sys.stdout.write('\n')
                                spacing = 0
                        sys.stdout.write('\n')
                        sys.stdout.flush()
                        action = []
```

If you run that script, you can see that makes things much more easier to read. It should all be relatively straight forward, I added in a mechanism to wait till the count gets up to eight and automatically create a new line, and then automatically space apart every two lines (which makes it easier to separate key presses/releases). The line "sys.stdout.write('%02X ' % ord(byte))" takes the integer value of a byte and translates it into a hex form, if you were unfamiliar with that bit. After that is when I took a look at the directional pad input/output:

```
AC AF 15 01 01 00 01 00
```

```
14 B0 15 01 00 00 01 00  <- Button 1

A4 B2 15 01 01 00 01 01
24 B3 15 01 00 00 01 01  <- Button 2

4C B5 15 01 01 00 01 02
DC B5 15 01 00 00 01 02  <- Button 3

C0 D5 15 01 01 80 02 00
A8 D6 15 01 00 00 02 00  <- D-pad left

E8 D8 15 01 FF 7F 02 01
C0 D9 15 01 00 00 02 01  <- D-pad down

D8 DC 15 01 FF 7F 02 00
E0 DD 15 01 00 00 02 00  <- D-pad right

A8 E0 15 01 01 80 02 01
A0 E1 15 01 00 00 02 01  <- D-pad up
```

So it appears the easiest way to tell the difference between a button push and another sort of action is going to be with the 7th byte. It is 01 (or \x01) when in reference to a button and 02 (\x02) when in reference to some other action. The D-pad up-down axis seems to be represented with an 8th byte of 01 and the left-right with a 8th byte of 00. Bytes 5 and 6 are either FF and 7F (down/right) or 01 and 80 (up/left) when the user pushes down on a button, and they will be 00 and 00 when the user is releasing a button. Then, a look at the left joystick:

(Ps: If your controller has a "mode" switch/button, it will be relevant to the output you see here. I had my mode button on for this, and I'd recommend you do as well.)

```
94 10 22 01 01 80 02 04
9C 12 22 01 00 00 02 04  <- Left js left

BC 16 22 01 FF 7F 02 05
BC 18 22 01 00 00 02 05  <- Left js down

DC 1C 22 01 FF 7F 02 04
C4 1E 22 01 00 00 02 04  <- Left js right

EC 22 22 01 01 80 02 05
AC 24 22 01 00 00 02 05  <- Left js up
```

These seem to be almost identicle to the D-pad with the exception that the 8th byte is 04 for the left-right axis and 05 for the up-down axis. Everything appears to be so far so good, but the right joystick is where things became a little more complicated. If you havn't taken a look at it before, take a look now. As you can see, it generates 5 times the output as it's left counterpart. It makes sense, as the right joystick is commonly used for aiming in first person shooters, that thing joystick should require much more accuracy than the "it's either on or off" approach of the d-pad or the left joystick. Unfortunately, though, that complicates our job a little bit more.

The eighth byte of the right joystick follows the same pattern as the D-pad and left joystick being 03 for up/down and 02 for left/right, and the 7th byte is also identicle with 02, but the fifth and sixth bytes are what differ. Unfortunately, I couldn't find much of a pattern for the fifth byte. My best guess is that it's a measure for acceleration of some sort. However, I was able to figure out how the sixth byte worked in the program. It's value changes as the axis is moved as such in the image below. The numbers represent the integer value of the character (ord(character)). It's also worth noting that, when replaced back in the middle position, the value will be 0.

Alrighty, so we have all of that figured out. Let's code it all in and see what we get, shall we?

```
import sys
pipe = open('/dev/input/js0','r')
action = []
spacing = 0
while 1:
        for character in pipe.read(1):
                action += ['%02X' % ord(character)]
                if len(action) == 8:

                        num = int(action[5], 16) # Translate back to integer form
                        percent254 = str(((float(num)-128.0)/126.0)-100)[4:6] # Calculate the percentage of push
                        percent128 = str((float(num)/127.0))[2:4]

                        if percent254 == '.0':
                                percent254 = '100'
                        if percent128 == '0':
                                percent128 = '100'

                        if action[6] == '01': # Button
                                if action[4] == '01':
                                        print 'You pressed button: ' + action[7]
                                else:
                                        print 'You released button: ' + action[7]

                        elif action[7] == '00': # D-pad left/right
                                if action[4] == 'FF':
                                        print 'You pressed right on the D-pad'
                                elif action[4] == '01':
                                        print 'You pressed left on the D-pad'
                                else:
                                        print 'You released the D-pad'


                        elif action[7] == '01': # D-pad up/down
                                if action[4] == 'FF':
                                        print 'You pressed down on the D-pad'
                                elif action[4] == '01':
                                        print  'You pressed up on the D-pad'
                                else:
                                        print 'You released the D-pad'


                        elif action[7] == '04': # Left Joystick left/right
                                if action[4] == 'FF':
                                        print 'You pressed right on the left joystick'
                                elif action[4] == '01':
                                        print 'You pressed left on the left joystick'
                                else:
                                        print 'You released the left joystick'

                        elif action[7] == '05': # Left Joystick up/down
                                if action[4] == 'FF':
                                        print 'You pressed down on the left joystick'
```

```
        elif action[4] == '01':
                print 'You pressed up on the left joystick'
        else:
                print 'You released the left joystick'

elif action[7] == '02': # Right Joystick left/right
        num = int(action[5], 16) # Translate back into integer form
        if num >= 128:
                print 'You moved the right joystick left to %' + percent254
        elif num <= 127 \
        and num != 0:
                print 'You moved the right joystick right to %' + percent128
        else:
                print 'You stopped moving the right joystick'

elif action[7] == '03': # Right Joystick up/ down
        if num >= 128:
                print 'You moved the right joystick upward to %' + percent254
        elif num <= 127 \
        and num != 0:
                print 'You moved the right joystick downward to %' + percent128
        else:
                print 'You stopped moving the right joystick'
action = []
```

The code above is a very basic structure of how to interpret readings. As you can tell, it outputs what has happened on each axis and calculates the percentage of push behind the right joystick. It's probably a good idea to create a simple outline like this before you go mapping any keys to make sure everything looks right, and that everything is outputting what it should be. Once you're ready to move on, we can get to mapping the buttons. There are a lot of different python libraries that can simulate key presses and mouse movements to X, but for this project I decided to go with the X automation library (Link here) for it's nice compatability with controlling the mouse speed and that it doesn't require calling programs through a shell. You'll need a few extra libraries, depending on your setup. Ubuntu users will want swig, x11proto-xext-dev, libx11-dev, libxtst-dev, and python2.6-dev. Other distribution users can find the equivalent files through your package manager.

The Xautomation library needs to be download and compiled on it's own, so head over, grab the tar file, and run a few commands:

```
tar xvvf xaut-?????.tar.gz
cd xaut-?????
./configure
make
```

Then you have to copy the files from the xaut-?????/python/xaut/ directory (xaut.py and _xautpy.so) into your project directory.

The xautomation library can now be imported into your script and we can continue coding.

```
# Keyboard presses
import xaut
keyboard = xaut.keyboard()
keyboard.click(38) # press the 'a' key
keyboard.down(38) # hold down the 'a' key
keyboard.up(38) # let up the 'a' key

# Mouse simulation
mouse = xaut.mouse()
mouse.click(1) # Clicks mouse button 1
mouse.btn_down(2) # Press button 2 down
mouse.btn_up(2) # Raise button 2
print mouse.x() # Mouse's current X position
print mouse.y() # Mouse's current Y position
mouse.move_delay(30) # Sets the delay in mouse movement (miliseconds per pixel)
mouse.move(100, 100) # Move to coordinates 100, 100 with delay 30
```

That's the standard usage of the Xautomation python library, and you can always view the help() docs if you need more info. Aside from the annoying key code usage (those can be found here by the way, or by calling keyboard.print_keycodes()) it's a very straightforward and easy to use library. After I had integrated it into my code, my finishing result was something like this:

JoystickMapper.py

The script above is available as a downloadable file, rather than printed here, for the sake of brevity. It works very similar to our last controller code, except that instead of simply printing out the occurring action it activates the equivalent mouse or keyboard action. I created signals (eg: MSBTN1) to make the script clearer, and the "keymap" dictionary is created so the script knows which joypad actions to map to which desktop actions.

The mouse handling works in a slightly different way than you might expect. Considering that we only receive a controller action when the joystick position *changes*, we would only be able to move the mouse on the desktop whenever we moved the joystick. If we stopped moving the joystick (such as holding it completely to the right), the mouse would not continue moving as expected. To solve this: I created a separate thread that constantly moves the mouse and checks for mouse updates, which are set via when the main thread receives a right joystick action. That way, the mouse can be constantly in motion without the joystick needing to be. It also calculates the percentage of push on the joystick, which is made relative to the speed of mouse movement.

So there we go. We've now achieved a functioning prototype for a controller mapper. Unfortunately, the script I created has been rather hacked together. One major flaw I've noticed is that if the mouse comes into contact with the border of the screen, it will vibrate and loop up the script. This is a particularly annoying issue with running games in full screen, where I noticed this almost instantly occurs. I would also like to redesign the script to become more object oriented, and provide the threads with better checking for when a Ctrl-C has been called. However, this is likely going to be a future venture for me to make.

Either way, I hope it's been a learning experience for you, because it sure was for me ;)

Here is a video of the script in action:
(It's not actually as jerky as I make it look, by the way)

Stealth-

Comments:

Posted by francis, on 03/02/2011
Thank you so much for this post mate! :)

Posted by ThorstenS , on 03/18/2011
really nice dude! best wishes from germany :)