Joystick API Documentation

Ragnar Hojland Espinosa
<ragnar@macula.net>

7 Aug 1998

1. Initialization

Open the joystick device following the usual semantics (that is, with open). Since the driver now reports events instead of polling for changes, immediately after the open it will issue a series of synthetic events (JS_EVENT_INIT) that you can read to check the initial state of the joystick.

By default, the device is opened in blocking mode.

```
int fd = open ("/dev/input/js0", O_RDONLY);
```

2. Event Reading

```
struct js_event e;
read (fd, &e, sizeof(e));
```

where js_event is defined as

```
struct js_event {
        __u32 time;     /* event timestamp in milliseconds */
        __s16 value;    /* value */
        __u8 type;      /* event type */
        __u8 number;    /* axis/button number */
};
```

If the read is successful, it will return sizeof(e), unless you wanted to readmore than one event per read as described in section 3.1.

2.1 js_event.type

The possible values of ``type'' are

```
#define JS_EVENT_BUTTON      0x01    /* button pressed/released */
#define JS_EVENT_AXIS        0x02    /* joystick moved          */
#define JS_EVENT_INIT        0x80    /* initial state of device */
```

As mentioned above, the driver will issue synthetic JS_EVENT_INIT ORed events on open. That is, if it's issuing a INIT BUTTON event, the current type value will be

```
int type = JS_EVENT_BUTTON | JS_EVENT_INIT;   /* 0x81 */
```

If you choose not to differentiate between synthetic or real events you can turn off the JS_EVENT_INIT bits

```
type &= ~JS_EVENT_INIT;                                  /* 0x01 */
```

2.2 js_event.number

The values of ``number'' correspond to the axis or button that generated the event. Note that they carry separate numeration (that is, you have both an axis 0 and a button 0). Generally,

```
                number
1st Axis X      0
1st Axis Y      1
2nd Axis X      2
2nd Axis Y      3
...and so on
```

Hats vary from one joystick type to another. Some can be moved in 8 directions, some only in 4, The driver, however, always reports a hat as two independent axis, even if the hardware doesn't allow independent movement.

2.3 js_event.value

For an axis, ``value'' is a signed integer between -32767 and +32767 representing the position of the joystick along that axis. If you don't read a 0 when the joystick is `dead', or if it doesn't span the full range, you should recalibrate it (with, for example, jscal).

For a button, ``value'' for a press button event is 1 and for a release button event is 0.

Though this

```
if (js_event.type == JS_EVENT_BUTTON) {
        buttons_state ^= (1 << js_event.number);
}
```

may work well if you handle JS_EVENT_INIT events separately,

```
if ((js_event.type & ~JS_EVENT_INIT) == JS_EVENT_BUTTON) {
        if (js_event.value)
                buttons_state |= (1 << js_event.number);
        else
                buttons_state &= ~(1 << js_event.number);
}
```

is much safer since it can't lose sync with the driver. As you would have to write a separate handler for JS_EVENT_INIT events in the first snippet, this ends up being shorter.

2.4 js_event.time

The time an event was generated is stored in ``js_event.time''. It's a time in milliseconds since ... well, since sometime in the past. This eases the task of detecting double clicks, figuring out if movement of axis and button presses happened at the same time, and similar.

3. Reading

If you open the device in blocking mode, a read will block (that is, wait) forever until an event is generated and effectively read. There are two alternatives if you can't afford to wait forever (which is, admittedly, a long time;)

a) use select to wait until there's data to be read on fd, or until it timeouts. There's a good example on the select(2) man page.

b) open the device in non-blocking mode (O_NONBLOCK)

3.1 O_NONBLOCK

If read returns -1 when reading in O_NONBLOCK mode, this isn't necessarily a "real" error (check errno(3)); it can just mean there are no events pending to be read on the driver queue. You should read all events on the queue (that is, until you get a -1).

For example,

```
while (1) {
        while (read (fd, &e, sizeof(e)) > 0) {
                process_event (e);
        }
        /* EAGAIN is returned when the queue is empty */
        if (errno != EAGAIN) {
                /* error */
        }
        /* do something interesting with processed events */
}
```

One reason for emptying the queue is that if it gets full you'll start missing events since the queue is finite, and older events will get overwritten.

The other reason is that you want to know all what happened, and not delay the processing till later.

Why can get the queue full? Because you don't empty the queue as mentioned, or because too much time elapses from one read to another and too many events to store in the queue get generated. Note that high system load may contribute to space those reads even more.

If time between reads is enough to fill the queue and lose an event, the driver will switch to startup mode and next time you read it, synthetic events (JS_EVENT_INIT) will be generated to inform you of the actual state of the joystick.

[As for version 1.2.8, the queue is circular and able to hold 64 events. You can increment this size bumping up JS_BUFF_SIZE in joystick.h and recompiling the driver.]

In the above code, you might as well want to read more than one event at a time using the typical read(2) functionality. For that, you would replace the read above with something like

```
struct js_event mybuffer[0xff];
int i = read (fd, mybuffer, sizeof(mybuffer));
```

In this case, read would return -1 if the queue was empty, or some other value in which the number of events read would be i /sizeof(js_event) Again, if the buffer was full, it's a good idea to process the events and keep reading it until you empty the driver queue.

4. IOCTLs

The joystick driver defines the following ioctl(2) operations.

```
                            /* function                    3rd arg  */
#define JSIOCGAXES          /* get number of axes          char     */
#define JSIOCGBUTTONS       /* get number of buttons       char     */
#define JSIOCGVERSION       /* get driver version          int      */
#define JSIOCGNAME(len)     /* get identifier string       char     */
#define JSIOCSCORR          /* set correction values       &js_corr */
#define JSIOCGCORR          /* get correction values       &js_corr */
```

For example, to read the number of axes

```
char number_of_axes;
ioctl (fd, JSIOCGAXES, &number_of_axes);
```

4.1 JSIOGCVERSION

JSIOGCVERSION is a good way to check in run-time whether the running driver is 1.0+ and supports the event interface. If it is not, the IOCTL will fail. For a compile-time decision, you can test the JS_VERSION symbol

```
#ifdef JS_VERSION
#if JS_VERSION > 0xsomething
```

4.2 JSIOCGNAME

JSIOCGNAME(len) allows you to get the name string of the joystick - the same as is being printed at boot time. The 'len' argument is the length of the buffer provided by the application asking for the name. It is used to avoid possible overrun should the name be too long.

```
char name[128];
if (ioctl(fd, JSIOCGNAME(sizeof(name)), name) < 0)
        strncpy(name, "Unknown", sizeof(name));
printf("Name: %s\n", name);
```

4.3 JSIOC[SG]CORR

For usage on JSIOC[SG]CORR I suggest you to look into jscal.c They are not needed in a normal program, only in joystick calibration software such as jscal or kcmjoy. These IOCTLs and data types aren't considered to be in the stable part of the API, and therefore may change without warning in following releases of the driver.

Both JSIOCSCORR and JSIOCGCORR expect &js_corr to be able to hold information for all axis. That is, struct js_corr corr[MAX_AXIS];

struct js_corr is defined as

```
struct js_corr {
        __s32 coef[8];
        __u16 prec;
        __u16 type;
};
```

and ``type''

```
#define JS_CORR_NONE            0x00    /* returns raw values */
#define JS_CORR_BROKEN          0x01    /* broken line */
```

5. Backward compatibility

The 0.x joystick driver API is quite limited and its usage is deprecated. The driver offers backward compatibility, though. Here's a quick summary:

```
struct JS_DATA_TYPE js;
while (1) {
        if (read (fd, &js, JS_RETURN) != JS_RETURN) {
                /* error */
        }
        usleep (1000);
}
```

As you can figure out from the example, the read returns immediately, with the actual state of the joystick.

```
struct JS_DATA_TYPE {
        int buttons;    /* immediate button state */
        int x;          /* immediate x axis value */
        int y;          /* immediate y axis value */
};
```

and JS_RETURN is defined as

```
#define JS_RETURN       sizeof(struct JS_DATA_TYPE)
```

To test the state of the buttons,

```
first_button_state  = js.buttons & 1;
second_button_state = js.buttons & 2;
```

The axis values do not have a defined range in the original 0.x driver, except for that the values are non-negative. The 1.2.8+ drivers use a fixed range for reporting the values, 1 being the minimum, 128 the center, and 255 maximum value.

The v0.8.0.2 driver also had an interface for 'digital joysticks', (now called Multisystem joysticks in this driver), under /dev/djsX. This driver doesn't try to be compatible with that interface.