Linux Input Core - past, present and future

Vojtech Pavlik

Table of Contents

Past

Historical origins

At the very beginning, there were the letters 'A' and 'B'. Later, there was a kernel, and that kernel ran on an i386 and a very limited set of hardware: AT harddrive, floppy, VGA, and AT keyboard controller - the i8042.

At that time, it was very reasonable to model the Linux console driver around what VGA text mode and the i8042 keyboard controller offered.

As time went on, and PCs got more powerful, the keyboard controller and the code supporting in the kernel stayed. Mice, which came later than keyboards were delegated to userspace, and the 'selection' (later 'gpm') program included the drivers for the first Mouse Systems and Microsoft serial mice.

Mice were appearing in many varieties, and where handling serial port mice was easy, other kinds (busmice, PS/2) needed special drivers. No effort was taken at that time to unify the outputs of these drivers to a common protocol - which was logical, because the serial mice, too already used many different protocols.

When X11 in the form of XFree86 started becoming more and more common, it needed to bring its own mouse drivers with it, interfacing to the many protocols the kernel offered for different mice. It also had its own keyboard driver, which switched the kernel one to 'raw' mode and interpreted the keyboard protocol itself.

Problems on the horizon

There was not just GPM and XFree86, there was also svgalib and a bunch of other programs, like games which needed to access the mouse and because of that needed their own mouse drivers. And this was not good, as it meant duplication of code, every driver existed in many different instances. And code duplication reliably leads

to bugs, different behaviors between implementations, etc. It was quite a chore to get ones mouse working in all the applications.

And for those who already sat back and were happy after setting it all up, there was a nasty surprise in store: Sharing a PS/2 mouse is virtually impossible, as the protocol is bidirectional, and one application (XFree86) talking to the mouse will confuse another one (GPM) which is just listening.

Graphic tablets and digitizers, as well as other special devices were supported directly by the applications that used them - CAD software, etc.

And with the dawn of 3D shooters a new generation of joysticks came pouring in, with digital interfaces over the old PC gameport, requiring a bunch of kernel drivers, each joystick having a different set of buttons and analog axes.

First attempt of an event-based input protocol

Since keyboards and mice are a rather touchy issue, because if they don't work, the system becomes more or less unusable, my first attempts of a protocol that would overcome the limitations of the current implementation were directed towards the joystick driver.

The original Linux driver was implemented as a character device, and its protocol consisted of packets with the following format:

```
struct JS_DATA_TYPE {
 int buttons;
 int x;
 int y;
};
```

This was well enough for the standard PC joystick which had a single paddle movable in along the X and Y axes and two buttons. It still was usable with joysticks that added additional buttons. But joysticks with rudder/throttle control, or any adding extra features couldn't fit into this scheme.

Thus there was no other way than to create a new protocol to read the joystick data and make that one extensible enough and easy enough to implement both in the driver and application side.

I created a protocol, that instead of always sending the whole device state would instead send 'event' packets each describing a single change in a button or axis value:

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

This of course is not enough to tell an application everything about the device - the API included ioctl()s to query the device name, protocol version and number of buttons and axes.

And this protocol has proven to be a good one - covering all the new beasts of joysticks that were created over time and also easy for game developers to use.

At that time I started considering whether a protocol like this would be good enough not just for joysticks, but also for all other input devices - keyboards, mice, tablets, touchscreens ...

USB HID - a blessing and a disaster in one

USB - Universal Serial Bus and it's HID - Human Input Device Class of devices were a great blessing for Linux, because USB HID was finally an open standard for input devices. Many joystick manufacturers started adopting it, throwing away proprietary (and error prone) digital gameport protocols, because switching to USB meant that every USB HID device will work with an OS supporting USB HID without specific drivers supplied by the manufacturer.

This meant that if Linux supported HID, all HID devices - all USB keyboards, mice, joysticks, etc would work with Linux.

HID was also a disaster, because it offers an unprecedent flexibility in the device layout - it can have unlimited number of buttons, degrees of freedom, scroller wheels and whatever else can be thought of. It's so generic that an UPS (uninterruptible power supply) can be described with it. Actually, USB UPSes do use HID.

This flexibility met a bunch of different static kernel to userspace protocols under Linux, of which none could be used for transfering input data from an USB HID device to userspace applications.

Present

Linux input core - goals

The goals of Linux input core are to cover all the problematic cases described in the previous chapters with a simple protocol, easy to implement in userspace applications, a thin core driver in the kernel, and a very easy API for writing new kernel drivers for input devices.

And most of all, the Linux input core was designed in simplicity in mind. What is simple is understood by many people, simple things have less bugs in them, because bugs often hide in complex and dark places of code, and because more people can spot them, simple usually also means less code, and faster code, too.

Last but not least there is always a need for backward compatibility - any system that replaces an old working one must not stop working and must support the same applications out of the box, lest users will not use it and stay with the old one.

Design decisions

Based on the above goals I decided to use a similar approach to the event based joystick interface and to extend it where necessary. The new events are very similar:

```
struct input_event {
 struct timeval time; /* Time when the event happened */
 unsigned short type; /* Type of the event - EV_KEY, EV_REL, EV_ABS, ... */
 unsigned short code; /* Code of the event - KEY_ENTER, REL_X, ... */
 unsigned int value; /* Value - press/release, position ... */
};
```

What needed to be extended from the joystick interface was that the joystick interface could only handle joystick buttons and axes. And those were numbered with consecutive numbers - ie a joystick always had buttons 0..n and axes 0..m.

For more generic description of a device, instead of plain numbers (like m and n in the previous case), bitfields were introduced, describing what types of events a device can generate,

```
 EV_CNF /* Reconfiguration of the device, sync */
 EV_KEY /* Key/button press */
 EV_REL /* Relative valuators - mouse movement, mouse wheel */
 EV_ABS /* Asolute valuators - joystick position, tablet position */
 EV_MSC /* Miscellaneous values */
 EV_LED /* LEDs (output) */
```

```
EV_SND /* Bells and whistles */
EV_REP /* Key repeat control */
EV_FF  /* Force feedback */
```

and for each event type valid for the device, a list of codes that the device is capable of sending or receiving. Some examples:

- Keys and buttons: KEY_8 KEY_9 KEY_0 KEY_MINUS KEY_EQUAL KEY_BACKSPACE KEY_TAB KEY_Q KEY_W BTN_LEFT BTN_MIDDLE BTN_RIGHT BTN_TRIGGER BTN_THUMB BTN_TOP BTN_SELECT
- Relative valuators: REL_X REL_Y REL_Z REL_WHEEL
- Absolute valuators: ABS_X ABS_Y ABS_RZ ABS_THROTTLE ABS_RUDDER ABS_PRESSURE
- LEDs: LED_NUMLOCK LED_CAPSLOCK LED_SCROLLLOCK LED_COMPOSE LED_KANA
- Sounds: SND_BELL SND_BEEP
- Repeat: REP_DELAY REP_PERIOD
- Force Feedback: FF_SPRING FF_FRICTION FF_PERIODIC

These bitmaps, along with driver version, device vendor and product IDs, and other information about a device can be obtained via ioctl()s in userspace.

Kernel side architecture

To satisfy the requirements that kernel drivers be simple and userspace should get both the generic event stream carrying the complete information and also backward-compatible interfaces resembling a PS/2 mouse, and have bindings to the console layer, a three-level architecture was implemented:

```
[HARDWARE] <--> input driver <--> input core <--> input handler <--> [USERSPACE]
```

The input driver talks to the hardware, handles interrupts and/or polls the input device repeatedly if interrupts are not available and sends input events to the input core via a function call:

```
input_event(device, type, code, value);
```

Pre-cooked variants for keys, absolute and relative valuators do exist:

```
input_report_key(device, code, pressed);
input_report_rel(device, code, displacement);
input_report_abs(device, code, position);
```

The input_event function resides in the input core and dispatches the events to input handlers which registered to receive events at init time.

The handlers then either queue the events and implement a character device which enables the userspace to read them: event interface - evdev.c, emulated PS/2 mouse - mousedev.c, joystick interface - joydev.c, emulated touchscreen - tsdev.c). Or they send the events somewhere else into the kernel: keybdev.c in 2.4 or keyboard.c in 2.5 send the events to the console code, power.c tells the powermanagement code about sleep/wake/power key presses.

This way a single device can appear in different interfaces - any mouse will be available both as an emulated PS/2 mouse and at the same time the events generated from the driver will be available to userspace in raw form.

An input driver dissected

A very basic input driver will look like this:

```
#include <linux/input.h>
#include <linux/module.h>
#include <linux/init.h>

#include <asm/irq.h>
#include <asm/io.h>

static void button_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
input_regs(&button_dev, fp);
input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) & 1);
input_sync(&button_dev);
}

static int __init button_init(void)
{
if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL))
return -EBUSY;

button_dev.evbit[0] = BIT(EV_KEY);
button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);

input_register_device(&button_dev);
}

static void __exit button_exit(void)
{
input_unregister_device(&button_dev);
free_irq(BUTTON_IRQ, button_interrupt);
}

module_init(button_init);
module_exit(button_exit);
```

Now let’s take a look at what this does.

First it includes the linux/input.h file, which interfaces to the input subsystem. This provides all the definitions needed.

In the _init function, which is called either upon module load or when booting the kernel, it grabs the required resources - in this case only the device IRQ. If the devices are not available, or the device is not present, it exits with an error code.

For drivers written for a hotplugging subsystem (like in the case of USB, CardBus, or serio), this allocation will happen in the ->connect() or ->probe() callbacks.

Then it sets the input bitfields. This way the device driver tells the other parts of the input subsystem what it is - what events can be generated or accepted by this input device. Our example device can only generate EV_KEY type events, and from those only BTN_0 event code. Thus we only set these two bits. We could have used

```
set_bit(EV_KEY, button_dev.evbit);
```

```
set_bit(BTN_0, button_dev.keybit);
```

as well, but with more than single bits the first approach tends to be shorter, namely in the cases where we set more than one bit.

The macros LONG() and BIT() are defined in linux/input.h and return the index in a long array and a bitmask for a specific position in the bitfield.

Then the example driver registers the input device structure by calling

```
input_register_device(&button_dev);
```

This adds the button_dev structure to linked lists of the input driver and calls device handler modules ->connect() callbacks to tell them a new input device has appeared.

While in use, the only used function of the driver is

```
button_interrupt()
```

which upon every interrupt from the button checks its state and reports it via the

```
input_report_key(device, code, pressed);
```

call to the input system. There is no need to check whether the interrupt routine isn't reporting two same value events (press, press for example) to the input system, because the input_report_* functions check that themselves.

The

```
input_regs(device, registers)
```

function is used to tell the input layer about the state of registers when an interrupt caused by a keypress happens, so that the kernel can print them out or take some other action in case that the key is mapped to a 'print registers' or other function.

The

```
input_sync(device)
```

function signalizes the input layer (and thus also the userspace) that this is the end of the report from the device - it groups the events into blocks which belong together. A user would be quite disappointed if a mouse on the screen didn't move diagonally but instead moved first horizontally and only after that it finished the motion by a vertical move.

An input handler dissected

Let's take a look at an example handler now. In this case we create a module that registers itself as a handler and receives all keypresses and printk()s them to the system log.

```
#include <linux/config.h>
#include <linux/input.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/module.h>

static struct input_handler printdev_handler;

void printdev_event(struct input_handle *handle, unsigned int type,
unsigned int code, int down)
{
printk(KERN_INFO "device: %p, type: %d, code: %d, value: %d",
handle->dev, type, code, value);
}

static struct input_handle *printdev_connect(struct input_handler *handler,
struct input_dev *dev)
{
struct input_handle *handle;

if (!test_bit(EV_KEY, dev->evbit))
return NULL;

if (!(handle = kmalloc(sizeof(struct input_handle), GFP_KERNEL)))
return NULL;
memset(handle, 0, sizeof(struct input_handle));

handle->dev = dev;
handle->handler = handler;

input_open_device(handle);

return handle;
}

static void printdev_disconnect(struct input_handle *handle)
{
input_close_device(handle);
kfree(handle);
}

static struct input_handler printdev_handler = {
event:  printdev_event,
connect:  printdev_connect,
disconnect:  printdev_disconnect,
};

static int __init printdev_init(void)
{
input_register_handler(&printdev_handler);
return 0;
}

static void __exit printdev_exit(void)
{
input_unregister_handler(&printdev_handler);
}

module_init(printdev_init);
module_exit(printdev_exit);
```

At the beginning the module again includes linux/input.h to get access to input core functions and structure definitions.

In the _init and _exit functions it only registers and deregisters itself as a handler with the input core. Since the input core is a hotplug capable subsystem (devices can come and go at runtime), the handler needs two callbacks ->connect() and ->disconnect() to be notified about new devices being connected and existing devices being removed from the system.

When a new device is registered with the input core by its respective input device driver, the core calls the ->connect() callback of each handler (and thus our example handler as well), asking it if it wants to receive events from that device.

If the handler is interested - in our case we check for any devices that can generate key or button events - it has to create a struct input_handle instance, open the device using it (so that the device driver can enable interrupts or start polling the device) and return the handle pointer.

In case the handler is not interested it just returns a NULL.

In ->disconnect() the driver closes the device and tears down the allocated structures.

And the last discussed function in this example is the ->event() function. It is called by the core, from the previously discussed input_event() function in case there was a change in the device valuators. This means that when a device calls input_event(EV_KEY, ..., 1); twice, the ->event() callback will not be called for the second time, since no change happened.

And in this example in ->event() we print the event on the screen. A real handler would probably queue it and send the output to userspace or process it in some way.

Gameport, serio

To simplify development of input device drivers additional core modules were implemented. Only a few input devices reside directly on the host bus, most of them are connected via some kind of simple serial interface (RS232, KBD/AUX), or some other kind of bus (USB, gameport). For PCI and USB Linux kernel already had subsystem cores with hotplug capability. However there were no such supporting core drivers for serial i/o - most of serial i/o is built around TTY code, and there was absolutely no support for gameports - joystick drivers relied on direct i/o access.

For this, two abstractions or subsystem cores were built: gameport.c and serio.c.

serio.c registers port drivers and serial device drivers the same way the input core takes care of input device drivers and input handler drivers. For each registered port it tries to find a driver which would be able to communicate with a device connected to that port. Since all RS232 serial ports are handled by the TTY layer in Linux, a single rs232 to serio driver exists, named serport.c. It implements a line discipline which routes all communication over a serial port to the serio subsystem.

Since the AT and PS/2 AUX and KBD ports are nothing else than synchronous serial ports running at approximately 9600 baud, clocked by the device, the easiest solution is to route their data through the serio subsystem as well. Hence i8042.c exists, which is a driver for the i8042 keyboard chip found on many platforms, accompanied by a bunch of other aux/kbd port drivers.

Many joysticks are connected to the gameport, and while in the past the only way to communicate over the gameport was accessing the i/o port 0x201, nowadays with the advent of PCI sound cards, the old method doesn't work in all cases. The interface abstraction is handled by the gameport.c core module, which works in pretty much the same way as any oyher subsystem core. It registers gameport drivers and drivers for gameport devices and tries to match them together.

Example data flows

To illustrate the role of the various subsystems, a couple of typical examples are shown below:

A gameport joystick:

```
ns558.c -> gameport.c -> analog.c -> input.c -> joydev.c -> USERSPACE
```

An AT keyboard:

```
i8042.c -> serio.c -> atkbd.c -> input.c -> keyboard.c -> TTY code.
```

An USB keyboard:

```
ohci-hcd.c -> usbcore -> hid -> input.c -> keyboard.c -> TTY code.
```

A PS/2 mouse:

```
i8042.c -> serio.c -> psmouse.c -> input.c -> mousedev.c -> USERSPACE
```

Future

Future enhancements

One of the most itching spots regarding drivers in the Linux kernel is the autoconfiguration, autoloading and configuration of userspace based on what devices are present.

This is handled by the hotplug agent, "/sbin/hotplug", which is called by subsystem cores when new devices of their class appear. In 2.5, the input core has support for the hotplug agent, though the gameport and serio subsystems still lack it. Support for the hotplug agent is planned for gameport and serio in the future, so that when a PCI card with a gameport is found by the PCI subsystem and the soundcard driver is loaded by the hotplug agent, when the sound driver registers a gameport on the sound card, the gameport core would request a driver for a device attached to that gameport.

Another new technology evolving in the 2.5 kernel is 'sysfs'. This is a filesystem that reflects the driver structure in the kernel, allowing drivers to properly handle powermanagement, and allowing to configure the drivers at runtime. The input core as well as serio and gameport lack this functionality and it will have to be implemented in the future, too.

Collateral benefits

The transition to the input core doesn't bring only the expected beneficial results, like a single event protocol between userspace and the kernel that covers all existing input devices, is simple and extensible, but also some benefits that were not the goals of the projects.

When converting the keyboard drivers to the input core, it became apparent that most of the keyboard drivers found in the Linux kernel in various per-architecture directories are in fact the same driver again and again, copied from the i386 architecture and slightly modified to get it working on a different architecture. This resulted in a massive code duplication, which was possible to remove thanks to the input core.

More or less the same applies to mouse driver - each mouse driver in the past implemented its own character device to interface with the userspace (this was partially helped by the 'busmouse' subsystem), and the input core allowed to remove all of that code, since the userspace interface is done by the event handlers.

Since the TTY code now communicates with the input layer, and the input layer has a fixed set of codes for keys, now called the 'Linux keycodes', with the input core in place there only needs be one set of keymaps for all languages, not M*N keymaps for all languages on all different hardware architectures. This is very nice for distribution makers, because it means less work and less worries about the keymaps.

Conclusion

Despite the fact that, like every new code, the input core had numerous bugs in it when was first completely deployed in the 2.5 kernel - two of two of Linus' keyboards didn't work - thanks to the simple design those were quickly fixed and now it serves its purpose well.

The input core was well accepted by both kernel and userspace Linux programmers, even XFree86 people like it very much, because it lifts the burden of implementing input drivers in userspace from them.