Creating an input device driver

The simplest example

Here comes a very simple example of an input device driver. The device has just one button and the button is accessible at i/o port BUTTON_PORT. When pressed or released a BUTTON_IRQ happens. The driver could look like:

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

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

static struct input_dev *button_dev;

static irqreturn_t button_interrupt(int irq, void *dummy)
{
        input_report_key(button_dev, BTN_0, inb(BUTTON_PORT) & 1);
        input_sync(button_dev);
        return IRQ_HANDLED;
}

static int __init button_init(void)
{
        int error;

        if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
                printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
                return -EBUSY;
        }

        button_dev = input_allocate_device();
        if (!button_dev) {
                printk(KERN_ERR "button.c: Not enough memory\n");
                error = -ENOMEM;
                goto err_free_irq;
        }

        button_dev->evbit[0] = BIT_MASK(EV_KEY);
        button_dev->keybit[BIT_WORD(BTN_0)] = BIT_MASK(BTN_0);

        error = input_register_device(button_dev);
        if (error) {
                printk(KERN_ERR "button.c: Failed to register device\n");
                goto err_free_dev;
        }

        return 0;

err_free_dev:
        input_free_device(button_dev);
err_free_irq:
        free_irq(BUTTON_IRQ, button_interrupt);
        return error;
}

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);
```

What the example does

First it has to include 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 (it should also check for the presence of the device).

Then it allocates a new input device structure with input_allocate_device() and sets up input bitfields. This way the device driver tells the other parts of the input systems 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.

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 functions to tell them a new input device has appeared. input_register_device() may sleep and therefore must not be called from an interrupt or with a spinlock held.

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()
```

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.

Then there is the:

```
input_sync()
```

call to tell those who receive the events that we've sent a complete report. This doesn't seem important in the one button case, but is quite important for example for mouse movement, where you don't want the X and Y values to be interpreted separately, because that'd result in a different movement.

dev->open() and dev->close()

In case the driver has to repeatedly poll the device, because it doesn't have an interrupt coming from it and the polling is too expensive to be done all the time, or if the device uses a valuable resource (e.g. interrupt), it can use the open and close callback to know when it can stop polling or release the interrupt and when it must resume polling or grab the interrupt again. To do that, we would add this to our example driver:

```
static int button_open(struct input_dev *dev)
{
        if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
                printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
                return -EBUSY;
        }

        return 0;
}

static void button_close(struct input_dev *dev)
{
        free_irq(IRQ_AMIGA_VERTB, button_interrupt);
}

static int __init button_init(void)
{
        ...
        button_dev->open = button_open;
        button_dev->close = button_close;
        ...
}
```

Note that input core keeps track of number of users for the device and makes sure that dev->open() is called only when the first user connects to the device and that dev->close() is called when the very last user disconnects. Calls to both callbacks are serialized.

The open() callback should return a 0 in case of success or any non-zero value in case of failure. The close() callback (which is void) must always succeed.

Inhibiting input devices

Inhibiting a device means ignoring input events from it. As such it is about maintaining relationships with input handlers - either already existing relationships, or relationships to be established while the device is in inhibited state.

If a device is inhibited, no input handler will receive events from it.

The fact that nobody wants events from the device is exploited further, by calling device's close() (if there are users) and open() (if

there are users) on inhibit and uninhibit operations, respectively. Indeed, the meaning of close() is to stop providing events to the input core and that of open() is to start providing events to the input core.

Calling the device's close() method on inhibit (if there are users) allows the driver to save power. Either by directly powering down the device or by releasing the runtime-PM reference it got in open() when the driver is using runtime-PM.

Inhibiting and uninhibiting are orthogonal to opening and closing the device by input handlers. Userspace might want to inhibit a device in anticipation before any handler is positively matched against it.

Inhibiting and uninhibiting are orthogonal to device's being a wakeup source, too. Being a wakeup source plays a role when the system is sleeping, not when the system is operating. How drivers should program their interaction between inhibiting, sleeping and being a wakeup source is driver-specific.

Taking the analogy with the network devices - bringing a network interface down doesn't mean that it should be impossible be wake the system up on LAN through this interface. So, there may be input drivers which should be considered wakeup sources even when inhibited. Actually, in many I2C input devices their interrupt is declared a wakeup interrupt and its handling happens in driver's core, which is not aware of input-specific inhibit (nor should it be). Composite devices containing several interfaces can be inhibited on a per-interface basis and e.g. inhibiting one interface shouldn't affect the device's capability of being a wakeup source.

If a device is to be considered a wakeup source while inhibited, special care must be taken when programming its suspend(), as it might need to call device's open(). Depending on what close() means for the device in question, not opening() it before going to sleep might make it impossible to provide any wakeup events. The device is going to sleep anyway.

Basic event types

The most simple event type is EV_KEY, which is used for keys and buttons. It's reported to the input system via:

```
input_report_key(struct input_dev *dev, int code, int value)
```

See uapi/linux/input-event-codes.h for the allowable values of code (from 0 to KEY_MAX). Value is interpreted as a truth value, i.e. any non-zero value means key pressed, zero value means key released. The input code generates events only in case the value is different from before.

In addition to EV_KEY, there are two more basic event types: EV_REL and EV_ABS. They are used for relative and absolute values supplied by the device. A relative value may be for example a mouse movement in the X axis. The mouse reports it as a relative difference from the last position, because it doesn't have any absolute coordinate system to work in. Absolute events are namely for joysticks and digitizers - devices that do work in an absolute coordinate systems.

Having the device report EV_REL buttons is as simple as with EV_KEY; simply set the corresponding bits and call the:

```
input_report_rel(struct input_dev *dev, int code, int value)
```

function. Events are generated only for non-zero values.

However EV_ABS requires a little special care. Before calling input_register_device, you have to fill additional fields in the input_dev struct for each absolute axis your device has. If our button device had also the ABS_X axis:

```
button_dev.absmin[ABS_X] = 0;
button_dev.absmax[ABS_X] = 255;
button_dev.absfuzz[ABS_X] = 4;
button_dev.absflat[ABS_X] = 8;
```

Or, you can just say:

```
input_set_abs_params(button_dev, ABS_X, 0, 255, 4, 8);
```

This setting would be appropriate for a joystick X axis, with the minimum of 0, maximum of 255 (which the joystick *must* be able to reach, no problem if it sometimes reports more, but it must be able to always reach the min and max values), with noise in the data up to +- 4, and with a center flat position of size 8.

If you don't need absfuzz and absflat, you can set them to zero, which mean that the thing is precise and always returns to exactly the center position (if it has any).

BITS_TO_LONGS(), BIT_WORD(), BIT_MASK()

These three macros from bitops.h help some bitfield computations:

```
BITS_TO_LONGS(x) - returns the length of a bitfield array in longs for
                   x bits
BIT_WORD(x)      - returns the index in the array in longs for bit x
BIT_MASK(x)      - returns the index in a long for bit x
```

The id* and name fields

The dev->name should be set before registering the input device by the input device driver. It's a string like 'Generic button device'

containing a user friendly name of the device.

The id* fields contain the bus ID (PCI, USB, ...), vendor ID and device ID of the device. The bus IDs are defined in input.h. The vendor and device IDs are defined in pci_ids.h, usb_ids.h and similar include files. These fields should be set by the input device driver before registering it.

The idtype field can be used for specific information for the input device driver.

The id and name fields can be passed to userland via the evdev interface.

The keycode, keycodemax, keycodesize fields

These three fields should be used by input devices that have dense keymaps. The keycode is an array used to map from scancodes to input system keycodes. The keycode max should contain the size of the array and keycodesize the size of each entry in it (in bytes).

Userspace can query and alter current scancode to keycode mappings using EVIOCGKEYCODE and EVIOCSKEYCODE ioctls on corresponding evdev interface. When a device has all 3 aforementioned fields filled in, the driver may rely on kernel's default implementation of setting and querying keycode mappings.

dev->getkeycode() and dev->setkeycode()

getkeycode() and setkeycode() callbacks allow drivers to override default keycode/keycodesize/keycodemax mapping mechanism provided by input core and implement sparse keycode maps.

Key autorepeat

... is simple. It is handled by the input.c module. Hardware autorepeat is not used, because it's not present in many devices and even where it is present, it is broken sometimes (at keyboards: Toshiba notebooks). To enable autorepeat for your device, just set EV_REP in dev->evbit. All will be handled by the input system.

Other event types, handling output events

The other event types up to now are:

- EV_LED - used for the keyboard LEDs.
- EV_SND - used for keyboard beeps.

They are very similar to for example key events, but they go in the other direction - from the system to the input device driver. If your input device driver can handle these events, it has to set the respective bits in evbit, *and* also the callback routine:

```
button_dev->event = button_event;

int button_event(struct input_dev *dev, unsigned int type,
                 unsigned int code, int value)
{
        if (type == EV_SND && code == SND_BELL) {
                outb(value, BUTTON_BELL);
                return 0;
        }
        return -1;
}
```

This callback routine can be called from an interrupt or a BH (although that isn't a rule), and thus must not sleep, and must not take too long to finish.