Reprinted with permission of Linux Magazine

Writing an Input Module

by Alessandro Rubini

Last month I gave an overview of the USB kernel subsystem, but I hadn't the space needed to show real code at work. This month we'll fill the gap by looking at sample drivers implementing input devices in the USB framework. The code being introduced has been developed and tested on version 2.3.99-pre6 of the Linux kernel, running on a PC-class computer.

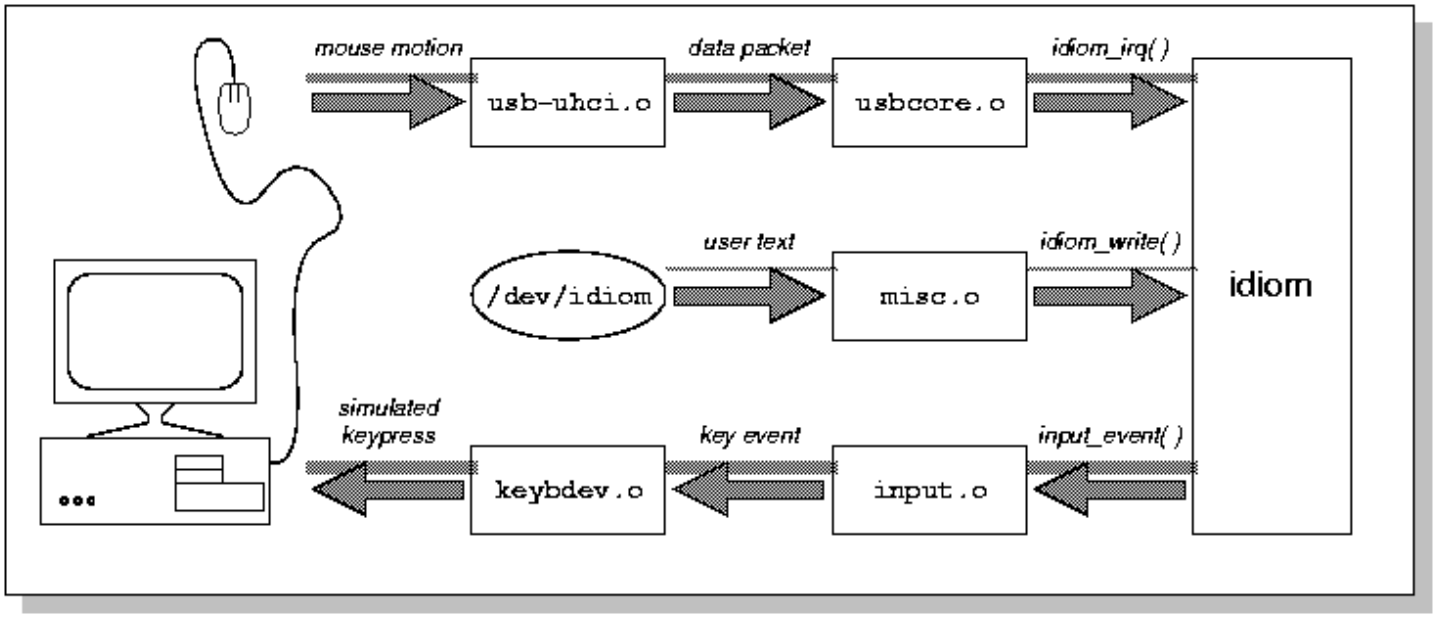
The sample module introduced here is called *idiom* (Input Device for Intercepting Output of Mice), and its well-commented source code is available for download from **here** . The sample module registers itself with the USB kernel subsystem as a mouse driver and with the input management subsystem as a keyboard driver. As a keyboard, *idiom* only reports arrow events, according to how the physical mouse is moved.

Since the module registers itself as an USB device, you won't be able to test its workings if you have no USB mouse to generate events. To partially fill the gap, *idiom* offers an additional entry point, `/dev/idiom`, where you can write text strings that will be converted to USB keyboard events.

Overview of the Idiom module

The overall design of the module is depicted in figure 1.

Data flow related to the idiom sample module



Donwload postscript: ***input.ps***

As outlined in the previous article, the USB input device driver must connect to two different infrastructures: the *usbcore* device driver that handles hardware events on the USB port, and the *input* module that collects and dispatches input events. In addition, *idiom* registers an entry point with the `misc` device driver. The figure shows how th device attaches to the three working environments using kernel facilities.

In order to test the module and assuming you compiled the USB kernel subsystem as modules, you'll need to invoke the following commands as root:

```
insmod usbcore
insmod usb-ohci

insmod input
insmod keybdev

insmod idiom
mknod /dev/idiom c 10 $(grep idiom /proc/misc | awk '{print $1}')
```

The first two commands load the USB machinery and an hardware driver for your host controller (not needed if you have no host controller on your computer and you plan to use `/dev/idiom`. The second pair of commands loads the input management machinery and the driver that consumes keyboard events. The last pair loads *idiom* itself and creates the associated misc device. You should remove `/dev/idiom` when you are done with these tests, or modify *idiom.c* to use `/proc` or *devfs* instead of the misc device.

Linking to the USB framework

At load time, the *idiom* driver (which is heavily based on the official *usbmouse* module) registers its own `usb_driver` data structure. The callbacks defined in the structure will be called whenever a new device is attached to the USB bus or an existing device is detached. Let's forget for a while that *idiom* registers with the misc device, and module initialization and shutdown turns out as simple as:

```
        static struct usb_driver idiom_usb_driver = {
                name:        "idiom",
                probe:       idiom_probe,
                disconnect:  idiom_disconnect,
        };

        int init_module(void)
        {
            usb_register(&idiom_usb_driver);
            return 0;
        }


        void cleanup_module(void)
        {
            usb_deregister(&idiom_usb_driver);
        }
```

Actual activation of the module happens within the *idiom_probe* function, called whenever a new device is plugged into a USB socket. The core of the *probe* function deals with detecting whether the new device is of the right kind or not, and I'm skimming over that part.

If device identification in the *probe* function goes well, a new device structure must be allocated -- you could use a static structure, but then your module won't support more than one device of the same type at the same time.

```
    /* allocate and zero a new data structure for the new device */
    idiom = kmalloc(sizeof(struct idiom_device), GFP_KERNEL);
    if (!idiom) return NULL; /* failure */
    memset(idiom, 0, sizeof(*idiom));
```

The new structure must include an ``USB Request Block'' structure, which needs to be initialized and submitted back to the USB subsystem:

```
    /* fill the URB data structure using the FILL_INT_URB macro */
    {
        static void idiom_irq(struct urb *urb); /* forward declaration */
        int pipe = usb_rcvintpipe(udev, endpoint->bEndpointAddress);
        int maxp = usb_maxpacket(udev, pipe, usb_pipeout(pipe));

        FILL_INT_URB(&idiom->urb, udev, pipe, idiom->data,
                     maxp > 3 ? 3 : maxp, idiom_irq,
                     idiom, endpoint->bInterval);
    }

    /* register the URB within the USB subsystem */
    if (usb_submit_urb(&idiom->urb)) {
        kfree(idiom);
        return NULL; /* failure */
    }
```

The macro `FILL_INT_URB` above is used to fill the URB data structure to describe ``interrupt transfers'' with the device. The USB specification defines four types of transfers (control, interrupt, bulk, isochronous), and the mouse device falls in the ``interrupt'' class. Interrupt transfers are not actually interrupt-based but rather happen as replies to polling performed by the host controller (driver).

The parameters of the macro call are used to initialize the field of the URB data structure, and the interested reader is urged to refer to `<linux/usb.h>` for the details. The interesting arguments here are `idiom_irq` and `maxp`. The former is the pointer to the *complete* handler, which eats data packets after completion of each transfer. The latter is the maximum length of the data buffer, which is set to three for the mouse driver -- although the protocol is actually made up of five-bytes packets, we only use the second and third byte; feel free to check *usbmouse.c* for differences in this initialization sequence.

During device operation, everything is accomplished by the *complete* handler, in this case *idiom_irq*. The function is handed a pointer to the `struct urb` and looks in the data buffer to retrieve data, after checking that no error has occurred:

```
static void idiom_irq(struct urb *urb)
{
    struct idiom_device *idiom = urb->context;
    signed char *data = idiom->data;
    struct input_dev *idev = &idiom->idev;

    if (urb->status != USB_ST_NOERROR) return;

    /* ignore data[0] which reports mouse buttons */
    idiom->x += data[1]; /* accumulate movements */
    idiom->y += data[2];
}
```

When the device is detached, the disconnect handler gets called. The handler must unregister the submitted URB and release memory allocated to the device:

```
    usb_unlink_urb(&idiom->urb);
    kfree(idiom);
```

These few lines of code are all that's needed for a device driver to interact to the USB framework. Even though things get slightly more complicated for different types of data transfers, the general rules outlined here apply, and the sources of real device driver will be a good reference for the curious reader.

Feeding the System with Input Data

We already know that reacting to USB data transfers is only half of the work of an USB device driver: the other half deals with communicating data to the external world. As far as input devices are concerned, the actual communication with the external world is performed by the *input* module, and the USB driver needs only to feed data to that module.

Communication with *input* is laid out in the usual steps: registration, communication, cleanup. The first step is performed by the same *probe* function that submits an URB to the USB framework; the driver must fill a `struct input_dev` to describe the input it might feed and register that structure. This is accomplished by the following lines:

```
    /* tell the features of this input device: fake only keys */
    idiom->idev.evbit[0] = BIT(EV_KEY);

    /* and tell which keys: only the arrows */
    set_bit(KEY_UP,    idiom->idev.keybit);
    set_bit(KEY_LEFT,  idiom->idev.keybit);
    set_bit(KEY_RIGHT, idiom->idev.keybit);
    set_bit(KEY_DOWN,  idiom->idev.keybit);

    /* and register the input device itself */
    input_register_device(&idiom->idev);
```

What's most apparent here is that everything is laid out in bitmasks: the driver must state that it only reports key events and also specify which keys will be generated (in this case, the four arrow keys). If you check `<linux/input.h>` you'll find several bit indexes defined, as other input channels are handled in the same way.

When the input device is registered with the input framework, the function *input_event* can be called by the driver whenever it has new data to feed. Within *idiom*, data is received by *idiom_irq*, and it's that very funtion that routes new data to its final destination.

Since *idiom* converts high-resolution mouse events to low-resolution keyboard events, I chose to only generate a keypress for every 10 pixel of mouse motion. Within *idiom_irq*, therefore, these lines are repeated four times (one for each possible direction):

```
    while (idiom->x < -10) {
        input_report_key(idev, KEY_LEFT, 1); /* keypress */
        input_report_key(idev, KEY_LEFT, 0); /* release */
        idiom->x += 10;
    }
```

These few lines of code turn an USB mouse to an arrow farm whenever you load *idiom* instead of *usbmouse*.

The function *input_report_key* is actually a wrapper to *input_event*: the first call showed above expands to ``input_event(idev, EV_KEY, KEY_LEFT, 1)''. While I personally would call `input_event`, it's more common practice to use the wrapper macro instead.

The last step in input management in unregistering. This is accomplished by a single line of code in *idiom_disconnect*. just before the `idiom` data structure is deallocated from memory:

```
    input_unregister_device(&idiom->idev);
```

Using /dev/idiom

To help people playing with *idiom* even in case no USB mouse or USB-capable computer is there, I added to the module support for a misc device. Use of the device is simple and straightforward: a new *idiom* device is created by the module whenever the device is opened, and a mouse movement is simulated whenever an uppercase or lowercase letter in the set ``u, d, l, r'' is printed to the device. For example, ```echo uuull ">" /dev/idiom`'' will generate three up-arrows and two left-arrows.

The implementation of the device entry point is confined to the final part of the source file for *idiom*, with the exception of the registering and unregistering calls, which are invoked at module initialization and shutdown.

Even though use of a ``simulated'' USB device cannot give the thrill of driving a real device, I hope this trick will help people with no USB hardware to begin lurking in this new arena of peripheral devices.

Be warned, anyway, that this discussion limits itself to the simplest available device, and driving a different device like a digital camera is quite a different kind of task, with new and interesting problems to face.

Alessandro is an independent consultant based in Italy, who suffers from high load and premature obsolescence. He reads email as `rubini-at-gnu-dot-org`.