Driving Me Nuts

The USB Serial Driver Layer

Understanding the USB-to-serial layer and how to get devices into it.

by Greg Kroah-Hartman

In my last column [see LJ December 2002], we covered the serial layer in the 2.5 (hopefully soon to be 2.6) kernel tree. We mentioned in passing that a USB-to-serial driver layer in the kernel helps out in working with those types of device drivers. This time we discuss that layer in more depth.

USB Serial Layer's History

A long time ago (in kernel development time, at least), a single USB-to-serial device driver was written and accepted into the kernel tree. It barely worked for one type of device and didn't work at all on SMP machines. As no standard USB-to-serial protocol existed, all devices used a custom protocol created by the individual vendors. The reason why there is no standard protocol is a long and sordid story; check the archives of the linux-usb-devel mailing list for the details. Soon a second type of USB-to-serial device was implemented within the first driver, sharing the reserved major and minor numbers. Over time, more and more devices were added to the driver until it was becoming an unwieldy mess. With the help of Peter Berger and Al Borchers, the original author of the driver rewrote the infrastructure and created what is now known as the USB-to-serial layer. This bit of code allows different USB-to-serial drivers to be written with a minimal amount of code, all sharing the same major and minor number range. It insulates the individual drivers from some of the complexity in the tty layer and the USB layer. It also allows the drivers to be compiled as individual modules and loaded only when they are needed.

In the 2.5 development cycle, the serial layer was created in order to provide an easier way to write serial port drivers, so as not to have to deal with the tty layer directly. Hopefully, someday the USB and serial layers will be merged. Both maintainers want to see this happen, but they do not have the time to do it. (They would gladly accept patches to accomplish this, if someone is looking for a project.)

In this article we cover the basics of the USB-to-serial layer, detailing how to register and unregister a driver and how to set up the main structures needed for a driver.

Registering and Unregistering a USB Serial Driver

All of the code and examples in this article are for the 2.5/2.6 kernel tree. The 2.4 and 2.2 kernel trees also support USB-to-serial drivers, but their interfaces are a bit different in places. For ease of use, we focus on only one kernel tree. If you have any problems porting a USB-to-serial driver to these older trees (once it is running on 2.5), please let me know.

To register a USB-to-serial driver with the kernel, the driver has to do two things: register with the USB-to-serial core and register with the USB core. Registering with the USB-to-serial core tells it to call your driver when new devices are found by the USB subsystem, and registering with the USB core is needed to tell it what kind of devices your driver can accept.

To register with the USB core, all you need is a list of USB devices that your driver will work for, in
traditional USB device ID format:

```c
static struct usb_device_id id_table[] = {
    {USB_DEVICE(MY_PRODUCT_ID, MY_DEVICE_ID),},
    /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, id_table);
```

This table is needed so the USB core knows what devices the driver can accept and the user-space hot-plug code knows what kind of devices are used. See my article `How the PCI Hot Plug Driver Filesystem Works', LJ May 2002, for more information about this table and how the hot-plug code uses it.

Then, a simple USB device-driver structure is created with this ID information:

```c
static struct usb_driver tiny_driver = {
    .name =         "tiny",
    .probe =        usb_serial_probe,
    .disconnect =   usb_serial_disconnect,
    .id_table =     id_table,
};
```

The .probe and .disconnect fields must be set to point to the USB serial core's functions because that type of logic is handled by it and not by your driver.

Then, a simple call registers this driver with the USB core:

```c
usb_register(&tiny_driver);
```

After this, the USB serial driver must be notified of the driver with a call to:

```c
usb_serial_register(&tiny_device);
```

This function takes a pointer to a struct usb_serial_driver_type, which will be explained in the following section.

To unregister a driver, the same steps have to happen, but in reverse order. First, we unregister with the USB serial core:

```c
usb_serial_unregister(&tiny_device);
```

Then, we unregister with the USB core:

```c
usb_unregister(&tiny_driver);
```

**struct usb_serial_device_type Explained**

To register with the USB serial core, the usb_serial_device_type structure must be filled. This structure can be found in drivers/usb/serial/usb-serial.h and is defined as the following:

```c
struct usb_serial_device_type {
    struct module *owner;
    char *name;
    const struct usb_device_id *id_table;
    char num_interrupt_in;
    char num_bulk_in;
    char num_bulk_out;
    char num_ports;
    struct list_head driver_list;
    int (*probe)(struct usb_serial *serial);
};
```
This is a rather large structure, but it's still smaller than either the tty layer's structure or the combination of the serial layer's structures, both of which are alternatives to using the USB serial layer.

Let us describe what all of these fields are used for and whether they are necessary. The owner field is a pointer to the module that owns this device. It should be set to the THIS_MODULE macro. When this is set, the module reference count logic is handled by the USB serial core, which is much safer than trying to do it on your own.

The name field is a pointer to a string that describes this device. This string is used in the syslog messages when a device is inserted or removed. It is also used in the /proc/tty/driver/usb-serial file to show what device is connected to what port.

The /proc/tty/driver/usb-serial File

The id_table field is a pointer to a list of usb_device_id structures that define all of the devices this structure can support. This field can be identical to the pointer that is passed to the USB core. If your driver needs to do different things for different types of devices, however, you can set up different structures describing these devices. An example of this is the Keyspan driver, which handles all of the Keyspan USB serial devices and needs different functions to be called for different devices.

The num_interrupt_in field is the expected number of interrupt in endpoints this device will have. An endpoint is a USB term, defined in the USB spec (www.usb.org). If you do not care about having the USB serial core check for this value (matching it up against any seen devices), use the NUM_DONT_CARE macro defined in usb-serial.h.

The num_bulk_in and num_bulk_out fields state the number of bulk in and bulk out endpoints this device will have. Again, the NUM_DONT_CARE macro can be used here if you do not want the core to pay attention to this value.
The `num_ports` field indicates the number of different ports this device will have. A single USB serial device can contain many different physical serial ports.

The `driver_list` field is used by the USB serial core to keep track of all the different drivers registered with it; it should not be used by the individual drivers.

The rest of the fields in the structure are all optional function pointers. If a field is not set, the generic USB serial driver's related function will be called. This allows a driver to be written with a minimal amount of code, if it happens to work the same way as the generic driver does. If not, almost all of these functions will need to be defined. These function pointers are divided into three groups: USB life-cycle pointers, tty life-cycle pointers and urb callback pointers.

**The Generic USB Serial Driver**

USB life-cycle function pointers consist of `probe`, `calc_num_ports`, `attach` and `shut down`. They are all called at different points in time as a USB device is initialized and shutdown. The `probe` function is called when a device matching one of the `id_table` devices is inserted into the system. This call happens before the device has been fully initialized by the USB serial core. It can be used to download any needed firmware to the device. In addition, any other early-initialize commands that the device needs can be sent at this time. If 0 is returned, the USB serial core continues on with the initialization sequence. Any other value will abort the call and notify the USB core that this device is not claimed by any drivers.

The `calc_num_ports` function is called to determine how many ports this device has. This should be used only by devices that can dynamically determine their ports. Any return value overrides the `num_ports` field in the `usb_serial_device_type` structure. It is called after the `probe` function is called but before the `attach` function is called.

The `attach` function is called when the struct `usb_serial` structure is fully set up. Any local initialization of the device or any private memory structure allocation can be done in this function. The `shutdown` function is called when the device has been removed from the system. Any local memory allocated for this device should be freed up at this time.

TTY layer function pointers consist of `open`, `close`, `write`, `write_room`, `ioctl`, `set_termios`, `break_ctl`, `chars_in_buffer`, `throttle` and `unthrottle`. If you recall the article on the tty layer [``The tty Layer'', *LJ* August 2002], these match up with the tty layer function call of the same name, with a few twists. First off, they all pass in a pointer to the specific `usb_serial_port` structure that is being operated on, and some of the functions are only called when something needs to happen.

The `open` function is called the first time `open()` is called on a port, but not for any subsequent calls to `open()`. Any urb submission the driver needs to do to start receiving data, or any device-specific messages that should be sent, should be done at this time. If any errors occur, they should be returned; otherwise, return 0 to signal success.

The `close` function is called for the last `close()` call, which is called from user space. Any running urbs should be shut down, and any device-specific commands that are needed should be sent now.

The `write` function is called exactly like the tty layer `write` function is called. The data passed to the function needs to be sent to the specified port. The number of bytes sent to the device should be returned. Remember, the device does not have to send all of the data that the user wants it to; a short write can happen, as long as the driver notifies user space that this has happened. This allows the driver logic to be much simpler. If an error happens, it should be returned as a negative number.
The write_room and chars_in_buffer functions are closely related. The write_room function is called by the tty layer to ask how many bytes the driver can accept to be written out at this time. The chars_in_buffer function is called to find out the number of bytes still left to be sent to the device.

The ioctl function is called with a wide range of tty ioctl values. If the driver cannot handle the specific value, -ENOIOCTLCMD should be returned. This will allow the tty layer to try to provide a default function. Some of the more common values asked for by user space are documented in the tty driver article previously mentioned.

The set_termios function is called to set terminal settings for a specific port. This includes baud rate, flow control, data bits and other line settings. The break_ctl function is called to set the BREAK value for the port. A state of -1 means that the BREAK status should be turned on, and a status of 0 means it should be turned off. The throttle and unthrottle functions are used to stop and resume data being received from the serial port.

### urb Callback Function Pointers

The read_int_callback, read_bulk_callback and write_bulk_callback function pointers are all used by the USB serial core to set up the initial callbacks for these kinds of USB endpoints. If the driver does not specify the read or write bulk callback functions, the generic callbacks are used. There is no generic read interrupt callback function, so if your device has an interrupt endpoint, you must provide this callback.

The operation of the generic read bulk callback adds the data received by the USB urb to the port's tty buffer, to be sent to user space when read() is called. It then resubmits the urb to the device. If your device does not need to interpret the data received in any way, I recommend using this function instead of writing a new one. The generic bulk write callback is much smaller and only wakes up the tty layer (in case it was sleeping, waiting for data to be transmitted to the device).

### Conclusion

In this article we have explained how to register and unregister a USB serial driver, as well as the basics of the main usb_serial_driver_type structure that all USB serial drivers must provide. Next time, we will go into the specifics of how the USB serial driver layer works and provide an example driver.

### Acknowledgements

I would like to thank all of the different programmers who have helped to create the USB serial layer over the years. Special thanks to Peter Berger and Al Borchers for their loadable module code offered back in July 2000, which is still in place today.

*Greg Kroah-Hartman is currently the Linux USB and PCI Hot Plug kernel maintainer. He works for IBM, doing various Linux kernel-related things and can be reached at greg@kroah.com.*