Driving Me Nuts

The Serial Driver Layer

Greg explains the interface to the new serial driver layer and how to register a serial driver and an individual serial port.

by Greg Kroah-Hartman

In the last two installments of Driving Me Nuts [LJ August 2002 and October 2002], we covered the tty layer, explaining how to create a minimal tty driver. We also explained some of the various ioctls and how to interpret the termios structure. Those articles are a great start if you have to implement a new tty-type device for your embedded system, such as a serial port. For every new system that is designed, the hardware engineers always like to place the serial port at a different address, or use a different UART, or sometimes just forget the serial port and use a USB port. So most developers want to create a whole new tty driver for their new devices in order for the hardware to work properly on Linux. Fortunately, there are some layers above the tty layer that help buffer its complexities and provide the developer with a lot of common functions that are needed for a serial driver and that fit the UART or USB model better. These layers are the serial and USB to serial driver layers. We cover the serial driver layer in this article and will cover the USB to serial layer in a future article.

Serial Mess

If you look at the code for the generic PC serial driver in the 2.2 and 2.4 kernels (available at drivers/char/serial.c), you will see a very complex driver that has numerous #ifdef lines, depending on the type of hardware you are using. This file has been copied numerous times in order to provide serial support for devices that do not fit the typical PC UART device (such as the serial_amba.c and serial_21285.c drivers). Thankfully, a number of developers, led by ARM Linux maintainer Russell King, restructured the serial driver into a generic serial core and a number of smaller, hardware-specific drivers. That code showed up in the main kernel tree sometime around the 2.5.28 release. The examples in this article are from version 2.5.35, so check that things have not changed for the kernel version you are using.

Registering a Serial Driver

The serial layer requires your driver to do two things: register the driver itself with the serial core and then register the individual serial ports as they are found in the system (through PCI enumeration or some other sort of device discovery mechanism). To register your driver, call uart_register_driver() with a pointer to a struct uart_driver structure. This function takes the information provided in the uart_driver structure and sets up the tty layer based on this.

The fields in the uart_driver structure that the serial driver needs to provide are the following:

```c
struct module           *owner;
const char              *driver_name;
const char              *dev_name;
int                      major;
int                      minor;
int                      nr; 
struct console          *cons; 
```
The owner field is a pointer to the module owning the serial driver. This is usually set to the macro THIS_MODULE.

The driver_name field is a pointer to the string that describes this driver, which is usually the same as the dev_name field. However, when describing the dev_name field, devfs needs to be taken into account, and the characters "%d" must be added to the end of this field if devfs is selected. This is because of the way devfs creates the device nodes. As an example, the amba.c driver sets the driver_name and dev_name fields as such:

```c
.driver_name            = "ttyAM",
#ifdef CONFIG_DEVFS_FS
   .dev_name               = "ttyAM%d",
#else
   .dev_name               = "ttyAM",
#endif
```

The major and minor fields specify the major number for the driver and the starting minor number, respectively.

The nr field specifies the maximum number of serial ports this driver supports.

The cons field is a pointer to the struct console structure that is used if this driver can support a serial console. If the driver does not support serial console mode, this field should be set to NULL.

### Registering a Serial Port

Now that the serial driver has been registered with the serial driver layer, each of the serial ports needs to be registered individually with a call to uart_add_one_port(). This function takes a pointer to the original uart_driver structure that was passed to uart_register_driver and a pointer to the uart_port structure. The uart_port structure is defined as the following:

```c
struct uart_port {
spinlock_t       lock;      /* port lock */
unsigned int     iobase;    /* in/out[bl] */
char             *membase;  /* read/write[bl] */
unsigned int     irq;       /* irq number */
unsigned int     uartclk;   /* base uart clock */
unsigned char    fifosize;  /* tx fifo size */
unsigned char    x_char;    /* xon/xoff char */
unsigned char    regshift;  /* reg offset shift */
unsigned char    iotype;    /* io access style */
unsigned int     read_status_mask;
   /* driver specific */
unsigned int     ignore_status_mask;
   /* driver specific */
struct uart_info *info;     /* pointer to parent info */
struct uart_icount icount; /* statistics */
struct console   *cons;     /* struct console, if any */
#ifdef CONFIG_SERIAL_CORE_CONSOLE
   unsigned long sysrq;     /* sysrq timeout */
#endif
unsigned int     flags;
unsigned int     mctrl;     /* current modem ctrl settings */
unsigned int     timeout;   /* character-based timeout */
unsigned int     type;      /* port type */
struct uart_ops  *ops;
unsigned int     line;      /* port index */
};
```
A majority of these fields are used by the individual serial drivers during their operation to define how the specific port is hooked up to the processor (through the hub6, iobase, membase, mapbase and iotype variables).

One of the more interesting variables in this structure is the struct uart_ops pointer, which defines a list of functions that the serial core uses to call back into the port-specific serial driver. This structure is defined as:

```c
struct uart_ops {
    unsigned int    (*tx_empty)(struct uart_port *);
    void   (*set_mctrl)(struct uart_port *,
                         unsigned int mctrl);
    unsigned int    (*get_mctrl)(struct uart_port *);
    void   (*stop_tx)(struct uart_port *,
                         unsigned int tty_stop);
    void   (*start_tx)(struct uart_port *,
                         unsigned int tty_start);
    void   (*send_xchar)(struct uart_port *, char ch);
    void   (*stop_rx)(struct uart_port *);
    void   (*enable_ms)(struct uart_port *);
    void   (*break_ctl)(struct uart_port *, int ctl);
    int    (*startup)(struct uart_port *);
    void   (*shutdown)(struct uart_port *);
    void   (*change_speed)(struct uart_port *,
                             unsigned int cflag,
                             unsigned int iflag,
                             unsigned int quot);
    void        (*pm)(struct uart_port *,
                         unsigned int state,
                         unsigned int oldstate);
    int        (*set_wake)(struct uart_port*,
                             unsigned int state);

    /*
     * Return a string describing the port type
     */
    const char *(*type)(struct uart_port *);

    /*
     * Release IO and memory resources used by
     * the port. This includes iounmap if necessary.
     */
    void   (*release_port)(struct uart_port *);

    /*
     * Request IO and memory resources used by the
     * port. This includes iomapping the port if
     * necessary.
     */
    int    (*request_port)(struct uart_port *);
    void   (*config_port)(struct uart_port *, int);
    int    (*verify_port)(struct uart_port *,
                             struct serial_struct *);
    int    (*ioctl)(struct uart_port *, unsigned int,
                             unsigned long);
};
```

This is a very large structure, with a lot of different function pointers, looking almost as bad as the tty_driver structure.
The startup function is called once each time the open(2) call happens. It is only called after the serial core has done a lot of resource checking and is sure the port needs to be opened. The serial driver usually does some hardware-specific setup to allow the port to be used in this function.

The shutdown function is the opposite of the startup function. It is called when the port is closed and all data has stopped flowing through it. This is where the hardware is told to stop, and any resources that were allocated in the startup function should be freed.

The request_port and release_port functions are used to reserve memory and other hardware resources that are related to the serial port. The config_port function is much like request_port, but it is called when the hardware can autoprobe for any serial ports connected to it and is also responsible for doing the same hardware reservations that request_port does.

The change_speed function is called whenever the port line settings need to be modified. The values passed to this function already have been cleaned up from the original termios structure that was passed through the tty layer, making the logic in the serial driver much simpler.

There are a number of functions that are used to get and set the serial line state and port status, these are:

- **set_mctrl**: sets a new value for the MCR UART register.
- **get_mctrl**: gets the current MCR UART register value.
- **stop_tx**: stops the port from sending data.
- **start_tx**: starts the port sending data.
- **tx_empty**: returns if the port transmitter is empty or not.
- **send_xchar**: tells the port to send the XOFF character to the host.
- **stop_rx**: stops receiving data.
- **break_ctl**: sends the BREAK value over the port.
- **enable_ms**: enables the modem status interrupts.

There are two functions related to power management issues with the serial port: pm and set_wake. If your hardware platform supports power management, use these functions to handle powering the hardware up and down.

verify_port is called to verify that the settings passed to it are valid settings for the specific serial port and is called when the user makes a TIOCSSESSERIAL ioctl(2) call on the port (see `"The tty Layer, Part II", LJ October 2002 for more on TIOCSSESSERIAL)

The serial driver layer handles many of the common serial ioctls, such as TIOCMGET, TIOCMBIS, TIOCMBIC, TIOCMSET, TIOCGSERIAL, TIOCSSERIAL, TIOCSRCONFIG, TIOCSERGETLSR, TIOCMWIWAIT, TIOCGICOUNT, TIOCSRGWILD and TIOCSERSWILD. If any other ioctl is called on the serial port, it is passed down to the specific port through the ioctl callback in the uart_ops structure.

The last remaining function is type, which is used to return a string describing the type of serial port. This is used in the proc file that resides in /proc/tty/driver/ and in the initial boot message when the port is discovered.
Where Is the Data?

You may have noticed there is no function in the struct uart_ops to send or receive data. The data that is sent from the user to the tty layer through a call to write(2) is placed in a circular buffer by the serial driver layer, and it is up to the specific UART driver to pick up this data and send it out of the port. Usually, every time the UART interrupt happens, the driver checks the circular buffer to see if any more data is present to be sent. The following example function shows one way of doing this:

```c
static void tiny_tx_chars(struct uart_port *port) {
    struct circ_buf *xmit = &port->info->xmit;
    int count;

    if (port->x_char) {
        /* send port->x_char out the port here */
        UART_SEND_DATA(port->x_char);
        port->icount.tx++;
        port->x_char = 0;
        return;
    }

    if (uart_circ_empty(xmit) ||
        uart_tx_stopped(port)) {
        tiny_stop_tx(port, 0);
        return;
    }

    count = port->fifosize >> 1;
    do {
        /* send xmit->buf[xmit->tail]
         * out the port here */
        UART_SEND_DATA(xmit->buf[xmit->tail]);
        xmit->tail = (xmit->tail + 1) &
                 (UART_XMIT_SIZE - 1);
        port->icount.tx++;
        if (uart_circ_empty(xmit))
            break;
    } while (--count > 0);

    if (uart_circ_chars_pending(xmit) <\n         WAKEUP_CHARS)
        uart_event(port, EVT_WRITE_WAKEUP);

    if (uart_circ_empty(xmit))
        tiny_stop_tx(port, 0);
}
```

The function starts out by looking to see if the x_char is specified to be sent out at this moment in time. If so, it sends it out and increments the number of characters sent out the port. If not, the circular buffer is checked to see if it has any data in it and if the port is not currently stopped by anything. If this is true, we start taking characters out of the circular buffer and send them to the UART. In this example, we send, at most, the size of the FIFO divided by two, which is a good average amount of characters to send. After the characters are sent, we see whether we have flushed out enough characters from the circular buffer to ask for more to be sent to us. If so, we call uart_event() with the EVT_WRITE_WAKEUP parameter, telling the serial core to notify user space that we can accept more data.

Any data received by the serial driver is passed to the tty layer, exactly like a normal tty driver would, with a call to tty_insert_flip_char(). This is usually done in the UART interrupt function.

Tiny Serial Example

Listing 1 [available on the LJ FTP site at ftp.ssc.com/pub/lj/listings/issue104/6331.tgz], shows an example of
how to register a serial driver with the serial driver layer and how to register a single serial port. This serial port is much like the previous tty example serial driver in that it acts like a character is received every two seconds as long as the port is opened. It also will print any characters that are sent to it with a call to write(2) to the kernel debug log.

**Conclusion**

In this article, the interface to the new serial driver layer has been explained, detailing how to register a serial driver and then an individual serial port. Thanks to this new driver layer being introduced in the 2.5 kernel, serial drivers can be much smaller, and creating a new one is a easier process, keeping the complexities of the tty layer away from the programmer.

*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](mailto:greg@kroah.com).