

Communicating with a Device using Linux and RS-232

Thomas Wa-jiw Casey

3/30/07

Abstract:

This document shows how to program Linux to communicate with a device using the RS-232 protocol. It explains how to create the device driver, how to install it on a Linux operated computer and how to access the driver through a user-space program.

Keywords:

Linux, Fedora, Red Hat, RS-232, Serial Communications, Device Driver

Introduction:

Serial data communications is the foundation for most forms of data communications used in modern computing. This guide assumes that you have configured your device to be able to send and receive serial data using RS-232 logic and have a basic knowledge of the C language.

If you need to know how to convert your data from TTL logic to RS-232 logic you should consult documentation on the MAX232 chip, which is very easy to use. A good website to learn how to use it is http://en.wikibooks.org/wiki/Serial_Programming:MAX232_Driver_Receiver If you want to learn more about the C language I recommend C Programming by David R. Brooks. You can view it for free on books.google.com.

Objective:

This article will be using the RS-232 based serial communication and will show how to implement it using the classic Unix C APIs for controlling serial devices. These APIs are the lowest level of abstraction besides kernel programming.

This guide will show how to implement communications using Fedora (Red Hat 7) using Linux kernel version 2.6. There are differences between this version of Linux with others so this guide would need to be modified if used with a different version.

Device Driver:

The device driver is split into five parts. The first are you have to decide what API libraries to use. Second you have to decide what type of file operations you need to use. Third you must implement the functions for those different file operations. Fourth you must create your initialize function which gets called when the driver is installed. Fifth and last you must create a cleanup function for when your driver is uninstalled.

1. API Libraries

There are certain Linux libraries you must include when creating any driver. These are:

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

#include <linux/kernel.h>

#include <linux/fs.h>

#include <asm/termios.h>

#include <asm/uaccess.h>
```

Using serial communications you must include this extra library:

```
#include <linux/ioport.h>
```

2. File Operations

When creating a device driver you must have open and close operations, but the rest are dependent on your device's capabilities. I will go over open, close, read and write. There are a few different types of ways to implement interrupts for your device so you may need to figure out the best way to implement them. Here is what a data structure would look like using the four operations I'm implementing, with comments where the other operations would go. The names of these inputs correspond to names of the functions that implement them.

```
struct file_operations rs_fops = {  
  
    NULL, /* lseek */  
  
    rs_read,  
  
    rs_write,  
  
    NULL, /* readdir */  
  
    NULL, /* poll */  
  
    NULL, /* ioctl */  
  
    NULL, /* mmap */  
  
    rs_open,  
  
    NULL, /* flush */  
  
    rs_write,  
  
    NULL, /* rs_fsync */  
  
    NULL, /* rs_fasync */  
  
    NULL  
  
};
```

3. File Operation Function Implementations

Here are examples of the four functions that you would have to implement for open, close, read and write. I've commented on each line what it does.

```
int rs_open(struct inode *in, struct file *filp){
```

```
    MOD_INC_USE_COUNT; // increments the number of drivers in the kernel
```

```
    minor = (MINOR(in->i_rdev)&0x0f); //gets the minor number of the device
```

```
    printk(KERN_INFO "rs:U OPEN %d\n" minor); //prints to kernel log that port is opened
```

```
    return 0;
```

```
}
```

```
int rs_close(struct inode *in, struct file *filp){
```

```
    MOD_DEC_USE_COUNT; //decrements the number of drivers in the kernel
```

```
}
```

```
int rs_read(struct file *filp, char *buf, size_t count, loff_t * t){
```

```

unsigned char x;

unsigned port = rs_base + minor;

x = inb(port + 1); //gets a byte from the port, + 1 for the offset

put_user(x, buf); //put the byte in user-space

printk(KERN_INFO "rs: READ %2.2x\n", x);

return 1;

}

```

```

int rs_write(struct file *filp, const char *buf, size_t count, loff_t * t){

unsigned char x;

unsigned port = rs_base + minor;

get_user(x, buf); //gets a byte from user-space

printk(KERN_INFO "rs: WRITE %2.2x\n", x);

outb(x, port); //sends byte to device

return 1;

}

```

4. Initialize Function

Here's an example of an initialize module for a serial driver:

```

int init_module(void){

int result = check_region(rs_base, 4);

```

```
#ifdef DEBUG

printk(KERN_INFO "rs: INIT_MOD\n");

#define DEBUG

if(result) // if getting region unsuccessful

{

return result;

}

if(!request_region(rs_base, 4, "rs")) // if port is busy

{

return 0;

}

result = register_chrdev(rs_major, "rs", &rs_fops); //register device

if(result < 0 ) // there was an error in registering

{

release_region(rs_base, 4);

return result;

}

if(rs_major == 0)

rs_major = result;
```

```
rs_buffer = __get_free_page(GFP_KERNEL); //  
rs_head = rs_tail = rs_buffer;  
}
```

5. Cleanup Module

Here's an example of a cleanup module for the previous driver.

```
void cleanup_module(void){  
unregister_chrdev(rs_major, "rs");  
release_region(rs_base, 4);  
if(rs_buffer)  
free_page(rs_buffer);  
}
```

Driver Install:

I will now cover how to install the driver onto a Linux system. This includes creating and running a makefile, and using system programs to install, and eventually remove your device.

1. Create Makefile

Here's an example makefile you could use if your driver file was called rs_example.c If you need to learn more about makefiles I suggest going to <http://www.opussoftware.com/tutorial/TutMakefile.htm>

```
all: t

CC = gcc

INCLUDEDIR = /usr/include

CFLAGS = -g -O2 -Wall -I$(INCLUDEDIR)

t: rs_example.o

$(CC) -o t rs_example.o

rs_example.o: rs_example.c

$(CC) -c rs_example.c $(CFLAGS)
```

2. Install Driver

Here's how to install your devices. You have to have root permissions to the computer though to be able to use them.

To install:

```
[ /folder_with_your_files]# /sbin/insmod -o rsport rs_example.o
```

where rsport is the name you want to use for the device

To remove:

```
[ /any_folder]# /sbin/rmmod rsport
```

where rsport is the device you want to uninstall

User-Space Access:

Now that you know how to install your device's driver you can access it in a user-space program by using:

```
FILE * fd = fopen("/dev/rsport", "r");
```

The previous would let you open your device for reading. Here is an example program for writing to a the previously installed device. What this does is test what happens when you connect the input pins of the DB-9 connector for the RS-232 port to itself. This is what I used to test that my basic driver was working.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
void loopTest(int data)
```

```
{
```

```
int fh, cnt;
```

```
char buf[2];
```

```
printf("Begin Loop Test %2.2x\n", buf[0], & 0xff);
```

```
fh = open("/dev/rsport", O_RDWR); // open device for reading and writing
```

```
if(fh)
{
write(fh, buf, 1); //write data to device

cnt = read(fh, buf, 1); // read data from device

printf("Status Port bits %2.2x\n", buf[0] & 0xff); //output what came through the input

close(fh); //close device

}

else

printf("Failed to open device\n");

}
```

```
int main()
{
int data;

printf("Press Ctrl C to exit\n");

while(1)
{

printf("Enter pattern in hex for data register");

scanf ("%x", &data);

loopTest(data);

}

}
```

Conclusion:

This guide was meant as a basis on communicating with a device using the RS-232 protocol and the Linux operating system. Every device has an individual way of sending and receiving data which is why I made this guide more general instead of specific to a certain device. You should be able to use this code with any device that uses RS-232 that constantly sends/receives data. Otherwise you will have to implement how your device handles interrupts etc. There is enormous amounts of data on the Internet on how to do this. The books that I used for references are a good beginning step to learn how to communicate with devices in Linux and help in deciding what protocol to use.

References:

1. Khan, Ashfaq A. Practical Linux Programming: Device Drivers, Embedded Systems, and the Internet. Hingham, Mass. Charles River Media, Inc. 2002
2. Corbet, Rubini, Kroah-Hartman. Linux Device Drivers - 3rd Edition. Sebastopol, CA. O'Reilly Media, Inc. 2005
3. Axelson, Jan, Serial Port Complete – Programming and Circuits for RS-232 and RS-485 Links and Networks. Madison, Wi. Lakeview Research LLC. 2000