Linux Device Drivers for the Radiometrix RPC Radio

Jeremy Elson
jelson@circlemud.org

Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292 USA

16 January 2001

This work was supported by DARPA under grant No. DABT63-99-1-0011 as part of the SCADDS project, and was also made possible in part due to support from Cisco Systems.

Contents

1 Introduction

This package contains a Linux device driver for the "RPC" (Radio Packet Controller) model of radio manufactured by Radiometrix[1]. The RPC is a fairly low-power, self-contained, short-range, plug-on radio. It requires only a simple antenna, 5V power supply, and interface to a byte-wide I/O port on a host microcontroller or bi-directional PC parallel port. The module provides all the RF circuits and processor intensive low level packet formatting and packet recovery functions required to inter-connect an number of microcontrollers in a radio network.

A data packet of 1 to 27 bytes downloaded by a host microcontroller into the RPC's packet buffer is transmitted by the RPC's transceiver and will "appear" in the receive buffer of all the RPC's within radio range.

This software package allows control of an RPC under Linux if the RPC is connected to a bi-directional PC parallel port. The driver is a Linux kernel module that implements two types of device drivers:

1. A character device, (e.g. `/dev/rpc`), allowing arbitrary data to be sent and received by user processes.

2. A network interface (`rpc0`), allowing peer-to-peer IP connectivity using the RPC as a slow but usable datalink network.

This Radiometrix driver was written by Jeremy Elson[2] (jelson@circlemud.org) while at the University of Southern California's Information Sciences Institute[3]. This work was supported by DARPA under grant No. DABT63-99-1-0011 as part of the SCADDS project, and was also made possible in part due to support from Cisco Systems. It is freely available under the GNU Public License (GPL). Up-to-date information, including the latest version of the software, is available via the SCADDS home page[4], or directly from the author's page[5].

2 The Physical Interface

The first step in using the Linux Radiometrix RPC Device Driver is establishing the physical connection between the RPC radio and your parallel port.

Note that this driver *does not use* the RPC development kit sold by Radiometrix. It assumes you have a *bare* RPC radio hooked directly up to the pins of the parallel port of your PC.

The pin mappings that this driver assumes exists between the RPC and the parallel port are shown below. However, if necessary, some of these can be modified by changing the `#define` statements in rpc_lowlevel.c. If you do change the pin mappings, be aware that they are constrained by the capabilities of the PC's parallel port, described in more detail in other documents[6].)

[1] http://www.radiometrix.co.uk/
[2] http://www.circlemud.org/jelson
[3] http://www.isi.edu/
[4] http://www.isi.edu/scadds/
[5] http://www.circlemud.org/jelson/software/radiometrix
[6] http://www.circlemud.org/jelson/software/parapin/docs/node2.html

Pin Label	RPC Pin	Direction	PC (Printer Port) Pin
GND	1	—	any of 18-25
D0	2	$\leftrightarrow$	2
D1	3	$\leftrightarrow$	3
D2	4	$\leftrightarrow$	4
D3	5	$\leftrightarrow$	5
TXR	6	$\leftarrow$	16
TXA	7	$\rightarrow$	12
RXR	8	$\rightarrow$	13
RXA	9	$\leftarrow$	14
RST	10	$\leftarrow$	17
5V	11	$\leftarrow$	+5 V, 20mA Supply
GND	12	$\leftarrow$	any of 18-25
Interrupt	See note	$\rightarrow$	10

2.1 Power and Grounding

The RPC needs a +5V supply in order to operate. According to Radiometrix tech support, the radio can actually handle inputs ranging from +4.5V to +5.5V.

It would be extremely convenient if you could simply supply the needed 5V from one of the output pins of the PC parallel port. However, the original PC parallel port was only spec'd to supply up to 2.5mA, and the RPC draws up to 20mA@5V. Using the parallel port for RPC power may work with modern parallel port controllers (if they exceed the official specs of their predecessors by a factor of 10); I've just never tried it. In our lab we typically use a battery connected to a voltage regulator, or a bench power supply.

Naturally, the PC and RPC must share a common ground. Take care if the RPC's power supply does not share a ground with the PC's power supply—make sure there is not a significant potential between the two grounds before plugging them together.

2.2 Interrupts

Using interrupts will significantly improve the performance of the radio but is not strictly required. The two state pins that are relevant for interrupt generation are

- **RXR**. This is the most important pin to monitor because it signals that a new packet has been received. The RPC only has a one-packet buffer, so this packet will be lost if it is not read before the next packet arrives over the air. For this reason, it is very important that RXR be serviced in a timely manner.

- **TXA**. This is the RPC's indication to a host that it is ready to transmit a packet. If TXA generates an interrupt, transmission performance will improve because packets move down the transmit pipeline more quickly.

Interrupts on the parallel port of the PC are signalled on the rising edge of pin 10. However, RXR and TXA are both normally high. Therefore, the simplest interrupt generator is simply an inverter between RXR and Pin 10. A better one is an XOR gate that combines RXR and TXA—then, when either one of these goes low, there is a low-to-high transition on the interrupt pin.

Note that if the RPC interface hardware generates interrupts, the PC parallel port itself must also be configured to generate interrupts; see Section 3.1 for details.

Important Note: If any transmitter in your testbed generates interrupts on TXA, make sure all receivers generate interrupts on RXR (or both RXR and TXA). This is required because a transmitter that uses TXA interrupts will transmit too quickly for a non-interrupt-enabled receiver.

3 Compiling and Installing the Driver

Unpacking the distribution and typing `make` should generate an object file called `krpc.o`. (Other files are also created while krpc.o is being built; these files can be ignored). A Linux kernel v2.2 or higher is required.

The `parapin` parallel port pin control library, which is required by the krpc module, is included in the RPC driver distribution. It can be used separately for other applications or drivers that use the PC parallel port as a generic digital I/O interface. For more details, see the Parapin home page[7].

After the module has been compiled, it can be installed much as any other Linux kernel module; e.g. `insmod krpc.o`. Note that it depends on the `parport` and `parport_pc` modules; these might have to be installed manually before you install `krpc`. For example:

```
insmod parport
insmod parport_pc io=0x378 irq=7
insmod krpc
```

The krpc module takes a number of options which can be set when the module is inserted by passing arguments to `insmod`; for example

```
insmod krpc.o lpt=1 krpc_debug_level=10
```

A complete description of legal module options can be found in Section 4.1.

3.1 Configuring the Parallel Port for Interrupts

If your RPC interface is capable of generating interrupts, special care must be taken to ensure that those interrupts are delivered all the way up to the RPC application. First, make sure the parallel port hardware is configured to generate interrupts. On modern motherboards with integrated parallel ports, this is selected from the BIOS setup screen. Older systems may use a DIP switch or jumper on the motherboard or the expansion board that drives the parallel port.

Next, the Linux kernel itself must be configured to handle parallel port interrupts. Unlike most other hardware devices, the kernel does not detect or claim the parallel port's interrupts by default. It is possible to manually enable kernel IRQ handling for the parallel port by writing the interrupt number into the special file `/proc/parport/`*n*`/irq`, where n is the parallel port number. For example, the following command tells the kernel that `parport0` is using IRQ 7:

```
echo 7 > /proc/parport/0/irq
```

If parallel port support is being provided to the kernel through modules, it is also possible to configure the IRQ number as an argument to the `parport_pc` module when it is loaded. For example:

```
insmod parport
insmod parport_pc io=0x378 irq=7
```

Note that both the `io` and `irq` arguments are required, even if the parallel port is using the default I/O base address of 0x378.

The actual interrupt number used by the kernel (7 in the examples above) must, of course, match the interrupt line being used by the hardware. The IRQ used by the parallel port hardware is usually configured in the BIOS setup screen on modern motherboards that have built-in parallel ports. Older motherboards or stand-alone ISA cards usually have jumpers or DIP switches for configuring the interrupt number. The typical assignment of interrupts to parallel ports is as follows:

Port	Interrupt
LPT1	7
LPT2	5

[7] http://www.circlemud.org/jelson/software/parapin

These are reasonable defaults if the actual hardware configuration is not known.

When the krpc module is inserted, it will report its status in the kernel log that appears on the console and/or in the system log files. Make sure these messages indicate the module is using interrupts. If the driver reports that it is in "polling mode", the Linux kernel is not properly configured as described above. **Note that the Linux kernel must be configured to use interrupts *before* the krpc module is inserted.**

Use the `irq_debug` module option (desribed below) to make sure that interrupts are actually being delivered. At least one "interrupt received" message should appear in the system log each time a packet is received (if your RPC interface generates RXR interrupts) or transmitted (if your RPC interface generates TXA interrupts).

4 Using the Device Driver

Once the krpc module has been successfully compiled and installed, it actually creates two different "personalities." The first is a character device (e.g., `/dev/rpc`), described in Section 5. The character device can be used to transmit arbitrary data from one station to another using a simple file interface; e.g. "`echo hello > /dev/rpc`" will transmit the string `hello`.

The second personality is a network interface (`rpc0`), described in Section 6. The network interface can be used to create a peer-to-peer IP network using the RPC radios. This interface can be configured in the normal way (i.e., using `ifconfig`).

Only one of these personalities can be used at a time. Multiple processes are allowed to access the device simultaneously; however, all accesses must be using the same personality at the same time. For example, if the network interface is configured and up, any attempt to use the character interface will return an error of `EBUSY` or `EAGAIN`.

4.1 Module Options and the `/proc` Interface

The krpc module takes a number of options. All of them can be set at the time the module is inserted by passing arguments to `insmod`; the syntax

```
insmod krpc.o lpt=1 krpc_debug_level=10
```

Some of the parameters can also be set and queried using the `proc` interface. All of the settable variables are in a directory called `/proc/krpc`. For example:

```
cat /proc/krpc/irq_debug                 queries current IRQ debug state
echo 4 > /proc/krpc/krpc_debug_level     sets the debug level to 4
```

The legal module parameters are:

Parameter	Default	/proc?	Description
lpt	0		Specifies the parallel port number to which the RPC is attached. This number refers to the parallel port number as assigned by the Linux kernel—type `ls /proc/parport` for a list. (Each subdirectory is a parallel port number.)
krpc_major_number	240		The major number registered by the RPC character device.
irq_debug	0	Yes	If 1, a message is printed each time an interrupt is received by the RPC driver. This is useful to see if interrupts are working.
krpc_debug_level	1	Yes	This specifies the level of debug messages that should be printed. 0 means completely silent operation. 1 gives typical status messages. 5 gives more verbose errors. Levels higher than 5 are useful primarily for developers of the driver. All krpc debug messages are prefaced by `krpc:` and the current UNIX time (seconds and microseconds since Jan 1, 1970).
ca_active	0	Yes	See Section 4.2
ca_holdoff	5	Yes	See Section 4.2
ca_holdoff_rand	5	Yes	See Section 4.2

4.2 Collision Avoidance Features

In environments where many RPC radios are trying to transmit simultaneously, collisions can become very likely. Collisions cause lost packets. Lost packets waste bandwidth but can also lead to more serious problems for algorithms that do not deal well with loss. Ideally, this should be fixed at the MAC layer by implementing RTS/CTS scheme, and/or MAC-layer retransmission. While the RPC driver currently does not incorporate such features, it does have a simple collision avoidance algorithm that has been reasonably effective for avoiding collisions.

With collision avoidance enabled, the RPC driver will not emit packets until a (randomized) quiet interval has passed—that is, an interval during which no packets have been received from other radios. We call this mandatory waiting time between receiving and sending the *holdoff period*.

Collision avoidance works because

- Packet arrivals are usually correlated due to fragmentation. In other words, if we just heard a packet it's likely that another one is coming.
- Multiple stations that are waiting to transmit will hopefully not collide due to the randomization of the holdoff period. As long as two holdoff periods don't expire simultaneously, the first to expire will cause other stations to restart their holdoff periods.

A larger number of peers in a collision domain requires a larger range of possible holdoff periods that can be picked by the randomizer. Squeezing too many nodes into too small a number of possible holdoff periods will make it much more likely that two or more holdoff periods will expire simultaneously, leading to a collision.

On the other hand, the downside to collision avoidance is that long holdoff periods lead to channel underutilization. In a network with few nodes but large holdoff periods, nodes will often sit around waiting in their holdoff periods when they could have been transmitting packets instead.

The module options that control collision avoidance are:

- **ca_active**—turns the collision avoidance algorithm on or off (0=off, 1=on). Default is 1.

- **ca_holdoff**—the *minimum* holdoff period, measured in *jiffies*. A jiffy is usually 10ms. (Its exact definition is 1/HZ, where HZ is defined in the platform-specific `/usr/src/linux/include/asm/param.h`).
- **ca_holdoff_rand**—the randomized value added to ca_holdoff; see below.

The RPC's CA algorithm can be summarized as follows: If ca_active is 1, and we want to send a packet, we do not send anything until X jiffies have passed without any packets being received—where X = ca_holdoff plus a random number between 0 and ca_holdoff_rand.

5 The Character Device

The character device personality of krpc provides simple access to the radio using a convenient file interface.

When krpc is installed, it registers a major device number for the RPC device. By default, the major number is 240 (this number is reserved for local/experimental use in the kernel). To change the major number to something different, use the `krpc_major_number` module parameter described in Section 4.1. Type `cat /proc/devices` to see the bindings of major numbers to device drivers.

The driver also has a number of different "sub-personalities" that can be selected using the minor number of the device file. Specifically:

Minor Number	Mode Description
0	Raw, direct access to the RPC radio. Messages of more than 27 bytes are not allowed due to the RPC hardware's message-size limitation.
1	Cooked radio interface; implements fragmentation so that larger packets, up to 8K, may be transmitted and/or received.
2	Packets are delivered to applications with the `rpc_augmented` structure prepended (i.e., before the data). This structure contains meta-data such as the time of reception of the packet. See the header file `krpc.h` for the definition of this structure.
3	Both 1 and 2.

Remember, *only one of these personalities may be used at a time.* If one process is reading from `/dev/rpc`, another process that attempts to open `/dev/rpcc` will receive `EAGAIN`.

5.1 Creating Device Files

After you decide on a major number to use for krpc, device files using that major number must be created. For example, using major number 240, the following commands would create the appropriate device files:

```
mknod /dev/rpc c   240 0  minor number 0: this creates the standard
                          interface, max of 27 bytes per write]

mknod /dev/rpcc c  240 1  minor number 1: this creates the "cooked"
                          interface, which does fragmentation, and
                          allows 8K writes

mknod /dev/rpca c  240 2  minor number 2: the "augmented" interface

mknod /dev/rpcca c 240 3  minor number 3: both 1 and 2
```

5.2 Using the Character Device

The RPC character device can be read and written with the normal `read(2)` and `write(2)` system calls, similar to other character device drivers.

However, there are important differences between this driver and other character devices. In particular, be aware that the sequence of data returned when you issue a read() acts more like a datagram interface. **All reads will deliver data starting from the beginning of a packet, even if the previous read did not consume a complete packet**. In other words, if a 20-byte packet comes in, and you only issue a read for 15 bytes, the remaining 5 bytes are lost forever. The next read does not start where the previous one left off; it gives you the beginning of the next complete packet. Therefore, it is important to issue reads with buffers large enough to consume entire packets.

In many ways, the semantics of the driver are the same as any other character device. For example:

- Multiple readers and writers are allowed. Writes are multiplexed correctly. Reads are a little more complicated; like with any other char device, data is only delivered to *one* reader when it comes in, not all of them. The reader that gets the data is undefined (whichever happens to wake up first). These semantics are the same as with any other char device.
- Blocking reads are supported.
- Nonblocking reads are supported; that is, if `O_NONBLOCK` is set using ioctl, a read will return EAGAIN if no data is immediately available.
- The `select(2)` and `poll(2)` system calls work.

5.3 RPC_MSG_PEEK: Reading a message without dequeuing it

Sometimes it is convenient to read the first few bytes of a message before reading the entire thing—for example, if the length of the complete message is encoded in its header. The traditional Berkeley datagram interface implements an option on `recv(2)` called `MSG_PEEK` to facilitate this.

The RPC implements a similar feature in its character device interface. When using the character interface (in any minor mode), requesting a read with the RPC_MSG_PEEK bit logical-or'd into the length will return the first message in the queue without actually dequeueing the message. A subsequent read call will return the same data.

In other words, if we have a file descriptor `fd` open to an RPC character device file, the call

```
read(fd, buf, 10 | RPC_MSG_PEEK);
```

will request the first 10 bytes from the message on the front of `fd`'s read queue, *without* removing that message from the queue.

5.4 Experimenting with the Character Device

It is possible to do quick debugging and experimentation using standard utilities such as `echo` and `cat`. For example, to wait for incoming packets on a receiver you can just type

```
cat /dev/rpc
```

Then, to send a packet containing "hello", on a transmitting side simply type

```
echo hello > /dev/rpc
```

Minor number 0 is a raw interface straight to the radio, so it will not accept writes longer than 27 bytes. This is due to the RPC packet controller's 27 byte message-size limitation.

To use the cooked interface that includes fragmentation, allowing you to send longer messages, use minor number 1:

```
cat /dev/rpcc  (on one side)

echo "this is a really long message that requires fragmentation" >
  /dev/rpcc    (on the other side)
```

Make sure to set up the receiver before sending; when you start receiving, the queue is cleared of old packets. Just like a socket, you can't receive data that was sent before you were listening for it.

6 The Network Device

When the krpc kernel module is loaded, it also creates a network device, `rpc0`. This lets you run IP over the RPC radio, similar to a wireless Ethernet card in "peer-to-peer" mode. All IP protocols (e.g. TCP, UDP, ICMP) work as usual—just a lot slower. In fact, the interface is so slow and unreliable that it is not especially useful for interactive applications such as telnet or X terminals. However, it may be useful for sending occasional debug or control information.

The rpc0 network interface is used just like any other interface. For example:

```
ifconfig rpc0 192.168.1.1  (on one machine)
ifconfig rpc0 192.168.1.2  (on a second machine)
```

The two hosts should then be able to ping each other, telnet to each other, etc.

7 Known Bugs and Limitations

Currently, the RPC driver can only control one RPC device at a time. Configurations with multiple RPC radios are not supported. (The underlying Parapin library currently has the same limitation.)

When using the fragmentation interface, the memory used by krpc will grow slowly over time (up to a certain limit). The driver currently does not implement garbage collection; it will in the future.

When providing an IP interface, the RPC driver really should do link-layer retransmissions. TCP performance is rather bad otherwise.

The implementation of fragmentation can be made somewhat more efficient; the first "introduction" fragment currently does not carry any data.

Please report other bugs and suggestions to the author, Jeremy Elson[8], at jelson@circlemud.org. I love getting feedback.

[8] http://www.circlemud.org/jelson