PCR-1000 radio for UN*X GUI Development and an OOP Library

Part II: The Comm

A Ghetto.Org Investigation
PolyWog and Javaman
17 November 1999

Objective:

Create a shared object (or static) library for facilitation of the Icom PCR-1000 UN*X interface. Create a GUI and command line interface for the Icom PCR-1000 all band, all mode receiver.

PCR-1000 Comm Port Specs:

Now that you have the protocol memorized,[1] we can move on to the second most important part: talking to your radio and listening to what your radio has to say. This part is devoted entirely to someone who is already familiar with serial port programming under UN*X environments.[2] It assumes the basic knowledge encapsulated in the Serial-Programming-HOWTO; for example, setting baudrates, opening UN*X file descriptors for character devices, and the various other info located in the `termios.h` header file.

Getting Started:

Making first contact is usually a very dangerous and delicate process. It involves use of the correct settings and functions. Some basic information that you will have to know (specific to your system) is what comm port the radio is attached to, and what device you would like to use in conjunction with that particular comm port. For example, on my laptop, I have the radio attached to comm one. On boot up, the IRQ's (etc) are set automagically by the Slackware `rc.serial` script. Most times this should be enough to start catting stuff to and from `/dev/ttyS0` or `/dev/cua0` (since comm one = 0, two = 1, three = 2, etc...)

But alas, no! For the script sets the comm port to 38600 baud (most distributions assume that you are going to use some fancy piece of hardware like a 14.4 modem where this would be desireable ;^) On power-up the PCR-1000 can only understand the universal 9600 baud speed rate. So what we have to do is tell the software to save the current comm port settings and alter them for the run of the program, then set it back when it's done playing with the port. Consider it *protection*... in case something gets foobar'd. So, let's start to play, shall we?

FD Haven:

First, we open and grab a file descriptor to our preferred device (for the purposes of modern technology, let's use `/dev/ttyS0`) This device should be opened with the flag options of read/write (O_RDWR), and not as a controlling tty (O_NOCTTY). We want to open it with the O_NOCTTY flag, because we do not want to get killed if some ghetto ass blow dryer (or even another radio itself) sends some line noise that the computer could interpret as a CTRL-C.[3]

---

[1]Don't worry, I have not left out the BandScope feature. This is a complex enough feature to merit its own section.

[2]If you are unfamiliar with serial port programming, I suggest you get a crash course in it by checking out the Serial Port Programming HOW-TO at http://www.linux-howto.com

[3]Note that throughout this whole explanation you should be doing some error checking to exit gracefully. Remember, `perror()` is your friend. He lives in `stdio.h` and `errno.h`. Be nice and use him as much as you want, you'll be better off. As an example, if you cannot open /dev/ttyS0 (which if this is your first time playing with the serial port, as a normal user... which you should be doing it as

Serial Etiquette:

Now, we save the current state of the serial port, so that we can reset it after we are done. You wouldn't want to leave any of your users surfing the net at 9600 baud, now would you ;^) The easiest way to do this is to create two `termios` objects. Get the current term attributes into one (labeled something like `old_foo`) and `bzero`() the other one. At the end of your program, or in one of your `SIGTERM`, `SIGBRK` handlers (yeah, you should be adding handlers, but if not, that'll be our secret) you set the attributes of the `old_foo` object back. After `bzero()`'ing the new one, you want to start playing with the new object's settings.

The Wind-Up:

First you should set the baud rate for the new object. There are several ways of doing this, but one way is to use `cfsetispeed`() and `cfsetospeed`() your `new_foo` to `B9600`. Now you want to alter the object's control modes (`c_cflag`). Let's think about the control modes that we will need. It's going to be a local connection, with no modem control. You will also need to enable receiving characters, as well as, disabling parity generation on output and parity checking on input. You will also want to set one stop bit rather than two, remove whatever character size mask it currently has, because you want to set it to eight bits. Finally the most important control mode, is to stop the hardware from lowering the modem control lines after the program exits. This is useful if you want to set a radio setting and keep the radio going after the program finishes (or hangs up the device.) In summation for the control flags, in the following order, you want to logic-or `CLOCAL` and `CREAD`, logic-and not-`HUPCL`, not-`PARENB`, not-`CSTOPB`, not-`CSIZE`, and finally logic-or `CS8`.

For the local modes (`c_lflag`), you want to enable canonical input, while disabling all echo functionality. In the following order you want to: logic-or `ICANON`, and logic-and not-(`ECHO` or `ECHOCTL`). For the output modes, you want to disable implementation-defined processing. A simple logic-and not-`OPOST` will take care of that. And that's pretty much it. You are ready to roll as far as calling `tcsetattr`() to `TCSANOW` your new termios object.[4] You may want to flush first (the port, that is). I don't, though.

The Catch:

Ok, so you are saying that it cannot be this easy? Well, actually, it is.... um, kind of. See there are some interesting little bugs in the PCR-1000 that could send you for a 3 hour loop, as it did to me when I first encountered it... and you are getting it here free of charge ;^)

1. In `G300` mode, (autoupdate off) you want to be wary of how many `write`()'s you do before you `read`(). If you tell the radio to go into `G300` mode, and then ask it to make you coffee, when you listen to what it has to say, you have to remember that it will always, blabber a new line. This new line is one char long, so you (should always check and) can *always* ignore `read`() counts that are equal to one byte.
2. If you send it more than one command before listening to a response, *always remember* that you will have to `read`() as many times as `write`()'s you sent it. The first `read`() will be for the first `write`() you sent it. The second `read`() for the second `write`() etc. etc. . . . Remember that asynchronous thing I was talking about? Well, that's where this would come in handy ;^)
3. If you are /<R4d 31337 enough to be using `read`() instead of some other lame ass call (ie: if

---

anyways, you'll run into permission problems) then you should check to see if the file descriptor is <0. If it is then catch that and use perror() thusly: `perror("shite! I couldnt open /dev/ttyS0");` this will tell you exactly, *why* you could not open `/dev/ttyS0.` Make sure you exit with an error code (for me, I just usually `exit(-1);` `exit`() can be found in `unistd.h`)

[4]You may feel so inclined to initialize your control characters, so that you can set the blocking timer, or minimum read. I haven't putzed with this yet, but you really don't need it since it is not *true* asynchronous I/O.

you did an `fdopen()` on the socket, and are using `fprintf()` or `fscanf()`) then remember to put a null after the last character of the buffer read in. You should be keeping track of the number of bytes read anyway, so doing a `foo_buf[bar]=0;` should be sufficient if `bar` is the number of bytes read in, and `foo_buf` is your buffer string.

WTF Are You Talking About:

If you didn't understand any of that above, RTFM!! Get the serial programming HOWTO and don't move until you understand what's going on!

Next Up: [Part III: The Notorious BandScope][5]

[5]Can't you just hear the sinister music ;^)