InDepth

Controlling Devices with Relays

Ring bells and more with a simple C program and inexpensive hardware.

by Jason Ellison

I recently was given the curious task of making noise on a factory floor at set intervals. The company wanted to use this noise to provide important auditory cues for specific events. These events included the time employees should clock in, clock out, take a break and return from break. The solution I implemented has been working well, and everyone involved has been pleased with the results. The gratitude I receive from employees that clock in and clock out is related directly to the inaccuracy of their own personal timepiece—those with fast watches appreciate the apparent extra time they have before clocking in. Later, though, they curse me for being delayed in clocking out when the watches on their wrists clearly show it is time to go.

The company I work for uses barcode clock terminals placed throughout the plant to record the time that employees clock in and clock out. In addition, the barcode clocks also record what job each individual employee is performing. When an employee clocks in or out, the employee scans his or her identification badge and a barcode representing the job the person is doing. The clock terminal copies this data to a buffer, along with a timestamp of when the items were scanned. CMINet, a data collection system running on two Microsoft Windows NT systems, is used to acquire the data from the clocks. The clocks are polled every minute or so to retrieve the data held in the clock terminal's buffer. During polling, the clocks also are set to the local time of the machine polling the clocks. The data acquired from the clocks is later imported into the enterprise resource planning (ERP) system for job costing and payroll.

The company's policy on tardiness states, “employees will be considered tardy if they clock in one minute or more after their assigned start time.” Due to the company policy concerning tardiness, the event bells needed to be synchronized perfectly with the barcode clocks and the machines running CMINet. This synchronization would, in theory, alleviate the discrepancies between the recorded clock-in time, recorded clock-out times and the true local time. Correct time synchronization is key, but it is not discussed thoroughly in this article.

Considering the one-minute-or-more-late-equals-tardy rule and the fact that employees invariably become time-challenged when returning from break, it was decided that scheduled bell soundings would be useful. Therefore, a mission-critical noise-making system was needed, and I was given the task of implementing it.

Implementation Concepts

Based on the issues at hand, this project had to meet the following requirements:

1. Provide auditory cues using a bell system. The bells need to be audible and annoying enough to be noticed even after prolonged exposure.

2. Provide a simple way of closing an electrical circuit over a computer.

3. Provide accurate time synchronization for all time-sensitive systems, including the machine that would

trigger the bells, the machines running CMINet and, indirectly, the barcode clocks.

With the requirements defined I began to formulate a plan. Five new 12VAC bells and a 12VAC power supply were purchased. These are the same type of bells commonly used in alarm systems or any sounding system. The bells physically hammer themselves in the same manner as an old alarm clock, and they pump out the decibels. The bells needed to be wired so that connecting two wires would ring all the bells in the plant.

To control the bells, we needed to close the bells' circuit temporarily. I was pointed toward a specific relay device, the AR-2, that can be controlled by a computer over a serial port. This device would be used to complete the bell circuit, causing them to ring. The AR-2 is sold by Electronic Energy Control, Inc. (EECI) for about $44 US. It has two relays on a circuit board, and both can be connected as normally open (NO) or normally closed (NC) relays. Only one relay wired as NO would be used for this project. An enclosure (EN-B) to house the board also was purchased. An emergency on/off switch was mounted on the enclosure for use in case something went terribly wrong. The AR-2 is documented well enough that I knew it would be possible to write a program in C to control it. This program could be scheduled using cron jobs.

For time synchronization, the NTP protocol was chosen. NTP clients and servers are unlikely to have interoperability problems due to a well-defined protocol that is widely adopted. In addition, NTP clients can handle network latency, daylight saving time and local time zones. Free clients are available for almost every major operating system in use today. Computers using these clients can readily obtain time sources accurate to at least the millisecond. Good accuracy is exactly what we need when trying to synchronize multiple systems.

Implementation Details

The five 12VAC bells were installed throughout the plant and in the designated break areas so that everyone would hear them. The bells were wired in mixed series and parallel with a 12VAC power supply supplied by the vendor. One lead from the power supply was left unconnected to provide a point at which the bells could be turned on and off.

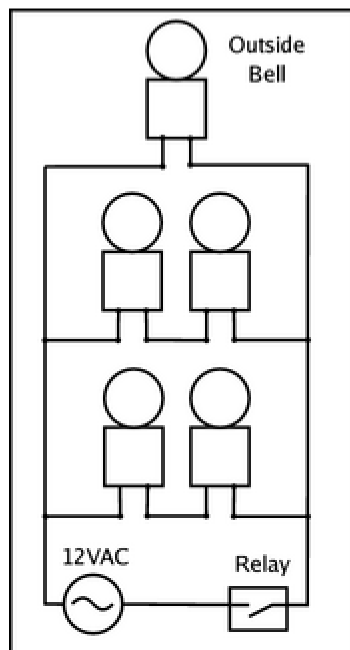

Figure 1. Schematic of the Bell System

I obtained one of our company's spare Dell PCs to use for this project, a basic PC with a Pentium III 750MHz processor, 128MB of SDRAM, 40GB IDE hard drive and an IDE CD-ROM. I used what was readily available, and really, the computer is a lot more powerful than necessary.

The AR-2 relay device attaches to the computer's serial port and is controlled by the DTR (data terminal ready) and RTS (ready to send) control lines. Setting RTS to high energizes the first relay and setting DTR to high energizes the second relay. Closing the bell circuit requires only one relay, so I needed to be able to toggle only the RTS signal. Example programs in GWBASIC and Turbo C were provided with the documentation. I wanted to write a program using GCC to control the relay but had little experience in C programming, so I searched the Web for some well-documented examples of C programs that made use of a serial port.

The first interesting program I found was upscheck.c from Harvey J. Stein's UPS HOWTO. upscheck, itself a modified version of Miquel van Smoorenburg's powerd.c, makes use of both the RTS signal and the DTR signal. The program originally was used to diagnose or examine UPS communications with a PC over a serial port. When starting the program, you indicate with parameters to which serial device the UPS is attached and whether to set RTS and/or DTR high. After the program opens the port and sets or clears the indicated serial control lines, the program would monitor another control line for feedback from the UPS. The program did everything I needed and more. All I had to do was remove the monitoring part, and I had my C program to control both relays on the AR-2 (Listing 1). The program is compiled with the following command:

```
[root@pluto ar-2]# gcc -o ar-2 ar-2.c

[root@pluto ar-2]# ./ar-2
Usage: ar-2 <device> <bits-to-set> <hold-time>
```

Listing 1. ar-2.c

```
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
/* Main program. */
int main(int argc, char **argv)
{
 int fd;
 int sleep_time;
 /* These TIOCM_* parameters are defined in
  *   <linux/termios.h>, which
  *  is indirectly included here.
  */
 int dtr_bit = TIOCM_DTR;
 int rts_bit = TIOCM_RTS;
 int set_bits;
 int flags;
 int status, oldstat = -1;
 int count = 0;
 int pc;
 if (argc < 3) {
  fprintf(stderr, "Usage: ar-2 <device> "
                  "<bits-to-set> <hold-time>\n");
  exit(1);
 }
```

```
 /* Open monitor device. */
 if ((fd = open(argv[1], O_RDWR | O_NDELAY)) < 0) {
  fprintf(stderr, "ar-2: %s: %s\n",
          argv[1], sys_errlist[errno]);
  exit(1);
 }
 /* Get the bits to set from the command line. */
 sscanf(argv[2], "%d", &set_bits);
 /* get delay time from command line. */
 sscanf(argv[3], "%d", &sleep_time);
 ioctl(fd, TIOCMSET, &set_bits);
 sleep(sleep_time);
 close(fd);
}
```

The program compiled with no errors, and better yet, it ran with no errors. Using the program is pretty straightforward, but let's briefly discuss it. AR-2 requires three parameters: device, bits-to-set and hold-time. Device refers to the special character device that addresses the serial device to which the AR-2 is attached. For example, if the device is attached to COM1, the device would be /dev/ttyS0. Bits-to-set is a little more complicated. It is a decimal number that must be converted to binary to be understood. Bit 2 controls RTS or relay one, and bit 3 controls DTR or relay two. So the decimal number 4, which in binary is 100, has the first bit set to zero, the second bit set to zero and the third bit set to one. Therefore, if the bit-to-set parameter is the decimal number 4, the DTR line would be placed high and the RTS line would remain low. The last parameter, Hold-time, simply is the number of seconds the modem control lines should be held in the state requested by the bits-to-set parameter.

I applied power to the AR-2 board and connected it to the serial port to test that the program worked as I had hoped. With the AR-2 connected to ttyS0 (COM1), I attempted to energize the first relay by placing RTS high by setting bit 2 with the command `./ar-2 /dev/ttyS0 2 5`. The program successfully placed RTS high for five seconds. After the program exited, it placed the RTS line in its original low state, de-energizing the relay. Next, I placed DTR high by setting bit 4 using the command `./ar-2 /dev/ttyS0 4 5`. Both RTS and DTR could be placed high by setting bits 4 and 2 at the same time, using the decimal number 6 for the bits-to-set parameter. The LEDs on the board are helpful in determining that both the AR-2 program and AR-2 board are operational.

Thanks to upscheck.c and its authors, I was able to place both the RTS and DTR control lines in a low or high state for a specified amount time. Without the source code listed in the UPS HOWTO, things would not have gone so smoothly. With a little effort, I was able to find C source code that was close to what I needed and modify it to meet my needs. I now can control two relays from a single serial port.

At this point, the bells could be attached to the first relay on the AR-2 using the NO contacts. Before connecting it all, though, I wanted to put the AR-2 in an enclosure, a nice little box to hold all this stuff so the electronics aren't exposed. I used the EN-B enclosure recommended and sold by EECI. It consists of two plastic pieces, a top and bottom, with black plastic inserts to cover openings on two sides. The enclosure is rather large for this job, but it works. I had to cut the black plastic inserts so that the external connectors would be accessible. Also, a hole was cut for the serial connector on the AR-2 that would be used for the connection to the computer's serial port. A hole was drilled for the female power connector that powers the AR-2 board. To connect the relay and the bell circuit, I chose a red and black banana clip, which can be found at most electronics stores. Two more holes were drilled for the banana clips. For added functionality, I added a power switch and power LED to the top of the enclosure so the power to the AR-2 board could be turned on and off. The LED lights up when the AR-2 board is turned on. With the electronics tested and enclosed, it was time to move on to the computer.

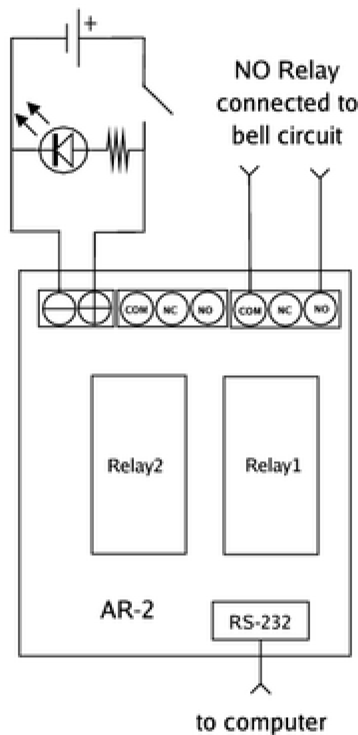

Figure 2. Connections to the AR-2 Board

I chose Red Hat 7.3 as the operating system. A fresh install was done, choosing Server as the installation type. Once the Red Hat installation was completed, I began customizing the box. I copied ar-2.c, the hacked-up version of upscheck.c, to the box and compiled it. After it successfully compiled, I copied the resulting binary ar-2 to the /usr/bin/ directory, because the AR-2 program runs from cron at specific times.

I wanted to allow remote administration of the cron jobs that ring the bells, so I really didn't feel comfortable with them running as root. I therefore decided to add a user for the specific task of ringing and gave it the user name bell with the command, `useradd bell`. The user bell has no need to log in locally or remotely, so it is a good idea to change the user's shell to a nonfunctioning one with the command `usermod -s /sbin/nologin bell`.

A problem with permissions arises here that needs attention. In Red Hat 7.3, the /dev/ttyS* devices have their permissions set so that only root and members of the group uucp have read and write access to them. User bell, however, needs to be able to write to the serial ports in order to control the attached relay device. By default, the user bell does not have read or write access to the serial devices. To solve this problem, I decided to make the user bell a member of the uucp group. As root, I added the user bell to the group uucp with the command `usermod -G uucp bell`. Now bell can use AR-2 on the /dev/ttyS0 device to manipulate the control lines of the serial port.

For remote administration of the bell's cron jobs, I choose a Web-based tool with which many Linux users are familiar, Usermin. I also downloaded Webmin for the purpose of configuring Usermin and installed them both. After installation, I logged in to Webmin and configured Usermin with the Usermin Configuration module. Because only user bell should be able to use Usermin, I used the Allowed Users and Groups module and selected “Only allow listed users”; bell, was of course, the only user I added to the list. Next, I used Available Modules to restrict all but the Scheduled Cron Jobs module. Last but not least, I went into the User Interface module and clicked Yes for the Go direct to module if user has only one option. Now bell is the only user that can log in using Usermin, and when user bell does log in, Usermin jumps straight to displaying Scheduled Cron Jobs. I also turned on mandatory SSL encryption. Webmin wasn't needed for anything else, so I stopped it with `service Webmin stop`. To make sure it does not start back up on a reboot, I entered

`chkconfig webmin off.`

Now we can log in to Usermin as the user bell and easily schedule the times at which the bells should ring and how long they should last by using the command `/usr/bin/ar-2 /dev/ttyS0 2 2`. Employees can feel comfortable knowing the bell always lets them know when to start running. Management can rest easy knowing they have yet another tool to help keep things running smoothly, and I can rest easy knowing that it's running on Linux, which means I don't have to worry about constantly babying it.

Further Possibilities

This was a simple exercise that illustrated the use of controlling devices using a relay over a serial interface. The low-cost relay used in this example is capable of controlling any number of electrical devices. Also available are relay boards that can be expanded to up to 128 relays per serial port, not to mention analog and digital input cards.

Resources

Electronic Energy Control, Inc.: www.eeci.com

Red Hat: www.redhat.com

UPS HOWTO: www.tldp.org/HOWTO/UPS-HOWTO.html

Usermin and Webmin: www.usermin.com and www.webmin.com

Jason Ellison is from Fairhope, Alabama, and currently is a network analyst for Delphi. He also maintains several Linux and AIX systems at Delphi.