Teaching Real-Time Control Using Free Systems Software

Kenneth H. Jacker
Computer Science Department
Appalachian State University
Boone, NC 28608 USA
khj@cs.appstate.edu

Abstract

The paper begins with a short description of the Department's real-time course and its required group projects. This is followed by an overview of the hardware and systems software used in a specific real-time control laboratory assignment. After a discussion of the analysis, design, and implementation of the project, the paper concludes with a brief look at future work.

1 Introduction

The theory and practice of real-time systems is diverse and interesting [9, 10, 11, 23]. Topics include scheduling (rate monotonic, earliest deadline first, priority inversion, etc.), quality-of-service (streaming audio and video), data acquisition and reduction, and digital signal processing (spectral analysis, adaptive digital filters, echo suppression, etc.).

Another particularly engaging area for university-level instruction is real-time control. Using computers to control traffic lights, oil refineries, video games, or the space shuttle requires specialized hardware and software in addition to advanced computer science knowledge.

1.1 CS4620: *Real-Time Systems*

A senior/graduate-level course (CS4620) within the Computer Science Department at Appalachian State University has been taught once a year since the early 1980s. Though some time is spent covering more traditional real-time topics, the majority of the semester concentrates on real-time data acquisition and control.

Other papers have discussed the evolution and structure of the course [7, 8]. Starting with a single low-performance personal computer (PC) and operating system (DOS), the course now uses three high-speed PCs and free systems software. The advantages of using commodity, off-the-shelf computers and free open source software cannot be over stressed.

1.2 Student Projects

CS4620 consists of three hours of lecture each week and a three hour laboratory. The lectures focus on more theoretical aspects (e.g., analog-to-digital converter design, POSIX threads, software modeling, and Fourier analysis) whereas the labs are devoted to the application of real-time data acquisition principles. A new programming assignment is given each week with students working in two- or three-person groups. In addition to printed hardcopy of all source files, each group must demonstrate its application in action.

The following sections present one such project: the "six segment display driver", *ssdd* or $(sd)^2$. After first describing the particular hardware and free systems software required by *ssdd*, a detailed description of the project follows.

2 Project Hardware

2.1 Data Acquisition Board

Although the application could use the PC's serial and/or parallel ports for its inputs and outputs, a more general approach is taken. Since other course assignments involve analog input and output, multi-function I/O boards were purchased from National Instruments (NI). Each PCI-MIO-16XE board contains three major sub-systems: analog-to-digital conversion, digital-to-analog conversion, and a group of eight bi-directional, transistor-transistor logic (TTL) compatible, digital ports. The direction of each digital port is set via software during *ssdd*'s initialization.

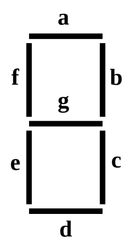

FIGURE 1: *Seven Segment Display*

2.2 Seven Segment Display

The most visible hardware component of the *ssdd* project is a single seven segment display (SSD). These electronic devices are common in digital watches, microwave ovens, alarm clocks and many other consumer products.

Segments In its simplest form, a SSD (see Figure 1) consists of seven "lines" or "segments" each of which is composed of multiple light-emitting diodes. Each segment is identified by a lower-case letter: a, b, c, . . . , g. A particular segment is enabled by setting its corresponding pin to a digital one and disabled by using a digital zero. Note that the state of each segment is independent of the others.

Digital Interface Only the first six segments (a-f) of the SSD are controlled by *ssdd*. The low-order six bits of the digital interface are configured for output and correspond to the segments as follows:

```
.----------------------------------.
| bit#    | 7  6  5  4  3  2  1  0 |
|----------------------------------|
| segment | -  -  f  e  d  c  b  a |
.----------------------------------.
```

Useful Segment Patterns Not all of the 2^7 total segment patterns are useful. In fact, most SSDs are driven by BCD-to-seven-segment decoders (e.g., the TI SN54LS49) which only allow decimal digits and a few other characters to be displayed. When an interface permits the independent enabling of all seven segments, some interesting pattern possibilities arise (e.g., "goose soup", "you lose, fella", "dollar", "philosophy is cheap", and even a few Chinese ideographs [15]).

2.3 Debounced Switches

An earlier lab used simple two-position mechanical switches. Students wrote programs that sampled the state of the switches at high-speed. They saw that a single depression of the switch resulted in multiple zero-one transitions. Thus, these unmodified switches produced false readings. To avoid this problem, *ssdd* uses two "debounced" switches [6] for input: `sw0` and `sw1`.

Digital Interface The two high-order bits in the NI digital sub-system are used to determine the state of the two switches. After being configured for input during initialization, the application is able to obtain the state of each switch via bits 7 and 6 as shown below:

```
.----------------------------------.
| bit#   |  7   6   5  4  3  2  1  0 |
|----------------------------------|
| switch | sw1 sw0  -  -  -  -  -  - |
.----------------------------------.
```

2.4 Keyboard

The final device needed by *ssdd* is the keyboard. As explained below, the program presents a *curses* interface to the user. Different options of the program are specified using unbuffered and "uncooked" character input obtained from the keyboard.

3 Free Systems Software

Prior course offerings used proprietary software environments [7]. The first version of the course used a PC and *ASYST* (an extended version of **Forth**). Later, with the help of a grant from the National Science Foundation, the course moved to a high-performance mini-computer manufactured by Concurrent Computer Corporation. Though the Real-Time Unix (RTU) operating system and real-time libraries were excellent, the hardware and software maintenance costs were too high for our Department.

Learning from the past, the course has returned to the PC platform. This time, however, the operating system and support software are free.

3.1 Linux and the FSF

The majority of the systems software used in the real-time course is based on the excellent GNU packages developed and distributed by the Free Software Foundation (FSF). In addition to *gcc*, *gas*, *ld*, *gdb* and related utilities, other freely-available programs (e.g., *gnuplot*, *scilab*, *tcm* and *xfig*) form the foundation for all software development and laboratory assignments in CS4620.

By far the most important software component is the Linux kernel itself. Even modern Linux kernels are not suited for real-time work. Though they do support soft real-time (using the `SCHED_FIFO` scheduling mode), they do not include hard real-time schedulers that can guarantee task scheduling deadlines. The exciting "preemptible kernel patches" [12] can reduce average latencies from hundreds of milliseconds (ms) to the 1–2 ms range. For slow data acquisition sampling rates, these approaches are fine. However, when the rates approach 5 kHz or greater, a hard real-time system is needed. See [10] for a discussion of hard, firm, and soft real-time systems.

3.2 RTLinux

Many real-time operating systems (RTOSs) are available for Intel, PowerPC, SPARC, and Alpha processors – both free and commercial. Examples of the latter include Compaq's RSX, Wind River's VxWorks, and QNX [5]. Like Concurrent's RTU, these are excellent, full-featured and mature systems. Unfortunately, there are two disadvantages in using these products: cost and lack of source code.

Examples of free RTOSs are KURT [20], RED-Linux [24], RTAI [3, 13, 14], and RTLinux [1, 2, 25]. Only RED-Linux, RTAI and RTLinux contain true hard real-time schedulers. For the past four years, the real-time course has used RTLinux originally developed by Victor Yodaiken and Michael Barabanov while at the New Mexico Institute of Mining and Technology.

RTLinux treats the entire conventional Linux kernel as the lowest priority real-time thread. Typical applications are divided into two cooperating entities: a single non-real-time Linux process, and multiple real-time threads. All time-critical portions of the application reside on the real-time "side". The remaining elements (user interface, data display, file input and output, networking, etc.) are placed within the standard Linux process.

Naturally, the real-time and non-real-time components must be able to communicate with each other. RTLinux provides this capability with real-time FIFOs (first-in, first-out) and shared memory (see Figure 2). Since the FIFOs are synchronous, no special mechanism is needed to ensure reliable inter-process communication. Shared memory, however, requires the use of some type of synchronization facility.

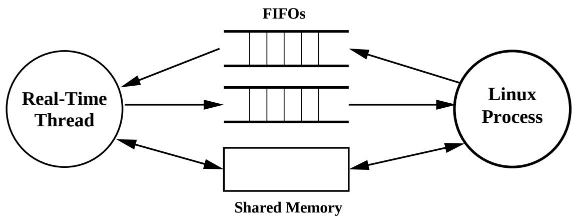

FIGURE 2: *FIFOs and Shared Memory*

3.3 COMEDI

Modern data acquisition systems need software to access their digital and analog hardware. Though students could be asked to write a custom device driver for the NI board [18], the task would exceed both their background and available course time. Instead, CS4620 uses the *Linux Control and Measurement Device Interface* (COMEDI) [19] created by David Schleef while at the Lawrence Berkeley National Laboratory.

This software consists of two major components: device drivers (implemented as loadable Linux modules) and libraries. Both user space (*comedilib*) and kernel space (*kcomedilib*) libraries are available. *Kcomedilib* was created specifically for use within real-time Linux threads.

Besides providing a robust application program interface (API), COMEDI function calls are device independent. This is achieved through an extra software layer between the library functions and specific data acquisition device drivers. Some of the common boards supported by COMEDI include those manufactured by Analog Devices, National Instruments, and Data Translation.

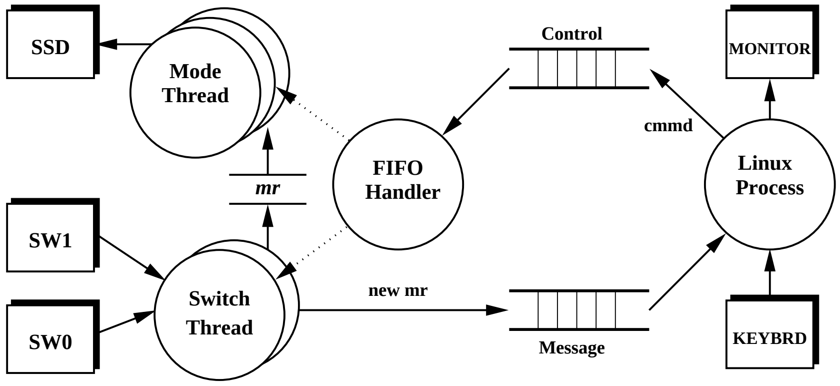

FIGURE 3: *The ssdd Application*

4 Analysis: What?

The main behavior of the *ssdd* application (see Figure 3) is to create different dynamic, cyclic patterns using the first six segments of the SSD. The frequency at which patterns repeat is determined by the "metabolic rate" (see below) which has a default value of 1 cycle per second.

Upon program startup, a "splash screen" is momentarily displayed followed by a top-level menu showing a list of various available "modes". Once a mode is selected, the SSD immediately displays the corresponding pattern and a mode-specific screen (see Figure 4) is shown on the monitor. In addition, all screens display the current time and date which are updated each second. Note that a given mode continues until another is selected or the program terminates.

```
.=========================================.
| Mode: Clockwise           02 Dec 6 12:00 |
|                                          |
|              mr:   1.00 Hz               |
|                                          |
| sw1 ===> mr++             sw0 ===> mr-- |
.=========================================.
```

FIGURE 4: *ssdd Mode-Specific Screen*

4.1 Modes

The project requires a total of seven modes. One of the modes is called the "mystery mode" and is created by the group members. Following is a short description of three typical modes.

Test In order to verify that the SSD interfacing hardware (supplied by the instructor) has been wired correctly and that the SSD itself is not defective, a *test* mode is run automatically during program initialization. This simple mode enables all six segments and disables all six segments three times at a rate of 1 cycle (i.e., all segments "on" and all segments "off") per second.

Alternating Us This mode consists of two patterns: an "upper U" (segments a, b and f) and a "lower U" (segments c, d and e). The two "Us" bounce up and down at the current metabolic rate.

Clockwise Here, the pattern consists of an illuminated segment going around the perimeter of the SSD in a clockwise direction. Each of the segments a, b, c, d, e, and f are successively enabled and then disabled.

4.2 Metabolic Rate

As mentioned earlier, the metabolic rate (*mr*) determines the frequency of the dynamic patterns being displayed on the SSD. It begins at one cycle per second, but can be increased or decreased by depressing (single clicking) the left (*sw1*) or right (*sw0*) switch respectively.

The maximum *mr* is limited to 30 Hz. Changing patterns on the SSD appear as a "blur" at frequencies greater than about 30 Hz. The center of each mode screen contains the current *mr* which, like the date and time, is updated dynamically.

5 Design: How?

The *ssdd* application uses both real-time and non-real-time components. The primary responsibility of the real-time components is monitoring the state of the two switches and ensuring that the proper SSD segments are enabled at the metabolic rate. The standard Linux process (running on the non-real-time side) manages the screen, routes keyboard inputs/commands to the real-time threads, and receives messages from the real-time side indicating changes in the metabolic rate. As shown in Figure 3, real-time FIFOs are used to pass commands and updated metabolic rates between the Linux process and the real-time threads.

5.1 Threads

Most students use two groups of RTLinux threads. Two threads are created each of which has the purpose of monitoring one of the switches. The work of sequencing and cycling the segments for each mode is given to separate "mode threads". Note that all threads in *ssdd* are periodic.

Before each thread can begin executing, it must first be created and scheduled. An internal data structure is instantiated which includes the thread's priority, period and other information. Once the thread exists, its execution is requested. When the thread actually begins running depends on its activation time and priority as specified in a node within RTLinux's "run queue".

Part of the difficulty of this project lies in accurately detecting single- and double-clicks of the switches. The problem is simplified by using debounced switches. The logic, however, is still complicated. Each group is required to model the switch behavior using a state transition diagram (STD). The switch-monitoring threads, therefore, are finite state machines. An example of such a model is shown in Figure 5.

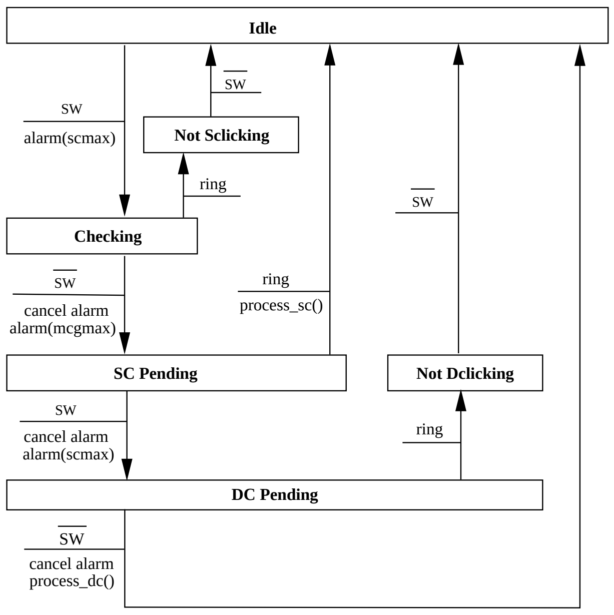

FIGURE 5: *STD: Switch Behavior*

5.2 RT FIFOS and FIFO Handlers

As requests are made by the user via the keyboard (change of mode or program termination), they are validated and then written to one of the real-time FIFOs, the "control FIFO". A second FIFO, the "message FIFO", is used to advise the Linux process of changes in the metabolic rate. Two very different methods are used to obtain the data being placed within these FIFOs.

In order to avoid unnecessary processor overhead by the real-time threads, a FIFO handler is created during program startup. Instead of constantly monitoring the control FIFO for new input, the handler (a C function) is invoked asynchronously any time new data arrives. Once the handler gets control, it reads the new control information and performs the requested action: stop the current mode thread, terminate all threads, or change the current mode.

A different problem appears on the non-real-time side. Since the Linux process must update the displayed date and time each second, it cannot just issue a Unix `read()` on the message FIFO. The process would "block" if no new *mr* was in the FIFO. One solution to this problem is to use `select()` [22] to monitor two file descriptors. One descriptor corresponds to the message FIFO and the second to "standard input" (for keyboard input).

Another approach is to use a standard Linux thread within the non-real-time process which updates the date and time asynchronously. This, in effect, would make the process completely "input driven". In other words, `select()` would use a "timeout" of zero, thus suspending the process until input became available from either the message FIFO or keyboard.

6 Implementation

Space does not permit a detailed discussion of a typical implementation. However, a few comments can be made.

Much of the code on the non-real-time side concerns itself with displaying text on the screen and menu management (also modeled as a STD). The former uses *curses* [4, 21], a simple, character-based interface first introduced in an early Berkeley Software Distribution (BSD) release. Most students seem to quickly understand how to effectively use the package through a class lecture and sample code.

RTLinux began using POSIX threads [16] in version 2.0. Though some non-portable extensions are included (e.g., `pthread_make_periodic_np()`), most routines adhere to the POSIX standard. Many sample programs are included in the RTLinux distribution to help learn how to use real-time threads and FIFOs. In particular, one fun program is *frank* which repeatedly writes "Frank" and "Zappa" to two FIFOs using two threads running at different priorities. A standard Linux process reads the FIFOs (using `select()`) and copies their contents to the screen.

COMEDI does a good job of providing access to most of the more advanced features (e.g., direct memory access) of a wide range of data acquisition boards while still retaining device independence. Overall, using the digital interface with COMEDI is quite easy. For example, once the direction of the digital lines is specified with `comedi_dio_config()`, reading from and writing to the digital sub-device is done via the `comedi_dio_bitfield()` function.

Analog input and output, however, is much more complicated. The most recent version of COMEDI uses "commands" instead of a multitude of functions with complex calling sequences. Though CS4620 does not currently use the newest version, it appears that creating analog applications will be much easier with the new API.

7 Future Work

7.1 *Tcl/Tk* Instead of *curses*

The use of *curses* is awkward and dated. Most of the students would much prefer using a graphical user interface instead of the simple character interface provided by *curses*. In the future, students will write the entire non-real-time portion in *Tcl/Tk* [17]. Though time must be spent teaching the fundamentals of this popular interpreted language, it is easy to learn and will provide a much more modern and attractive interface to the real-time applications.

7.2 Using All Segments via Encoding

The *ssdd* application should be able to control all seven of the segments. Even though it will require more complicated interfacing between the NI outputs and the SSD, each of the seven segments can be uniquely specified by encoding its "segment number" (e.g., 001/a, 010/b, ..., 111/g) in the lower three bits of the digital output. Another approach would associate each of the 2^6 possible digital output values with more complicated, multi-segment patterns (e.g., 1001/"upper U", 1010/"lower U", etc.).

7.3 RTAI

As mentioned in Section 3.2, RTAI (Real Time Application Interface) also supports hard real-time scheduling. Developed by Paolo Mantegazza in the Aerospace Department at the Politecnico di Milano, this alternative to RTLinux uses a very similar programming model (e.g., the splitting of an application into real-time and non-real-time sides, using FIFOs and/or shared memory for inter-process communication, etc.). Efforts will be made to use RTAI in future course offerings.

8 Summary

This paper has described the use of free systems software in teaching a university course in real-time data acquisition and control. Though *Real-Time Systems* requires significant prerequisites, most students enjoy the technical nature of the course and the challenges in implementing the lab projects. Some of the difficulties of using RTLinux and COMEDI will hopefully be reduced in upcoming offerings of the course.

References

[1] Barabanov, M. Introducing Real-Time Linux. *Linux Journal 34* (February 1997).

[2] Barabanov, M. A Linux-based Real-Time Operating System. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[3] Cloutier, Pierre, Mantegazza, Paolo, Papacharalambous, Steve, and Yaghmour, Karim DIAPM-RTAI Position Paper. In *Real Time Linux Workshop* (2000), Thinking Nerds.

[4] Goodheart, Berny *Unix Curses Explained.* Prentice Hall, 1991.

[5] Hildebrand, D. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (April 1992), USENIX.

[6] Horowitz, P., and Hill, W. *The Art of Electronics*, second ed. Cambridge University Press, 1989.

[7] Jacker, K. H. Real-Time Instructional Technology: Experiences with Multi-User Real-Time Systems. In *Real-Time Systems Education* (1996), IEEE Computer Society Press.

[8] Jacker, K. H. Teaching Simple Sound Synthesis: Real-Time, Numeric and Symbolic Computation. In *Real-Time Systems Education* (1997), IEEE Computer Society Press.

[9] Krishna, C. M., and Shin, K. G. *Real-Time Systems*. McGraw-Hill, 1996.

[10] Laplante, P. *Real-Time Systems Design and Analysis: An Engineer's Handbook*, 2nd ed. IEEE Press/IEEE CS Press, 1997.

[11] Liu, W. S. J. *Real-Time Systems*. Prentice Hall, 2000.

[12] Love, Robert Lowering Latency in Linux: Introducing a Preemptible Kernel. *Linux Journal 97* (May 2002).

[13] Mantegazza, P., Bianchi, E., Dozio, L., and Papacharalambous, S. RTAI: Real time application interface. *Linux Journal 72* (April 2000).

[14] Mantegazza, Paolo DIAPM-RTAI for Linux: WHYs, WHATs and HOWs. In *Real Time Linux Workshop* (1999), Vienna University of Technology.

[15] Neumann, P. G. DisPlay's the Thing. *Software Engineering Notes, V14#1* (January 1989).

[16] Nichols, B., Butlar, D., and Farrell, J. *Pthreads Programming*. O'Reilly and Associates, 1996.

[17] Ousterhout, J. K. *Tcl and the Tk Toolkit*, fourth ed. Addison Wesley, 1994.

[18] Rubini, A. *Linux Device Drivers*, second ed. O'Reilly & Associates, 2001.

[19] Schleff, David *Control and Measurement Device Interface*. `http://stm.lbl.gov/comedi`.

[20] Srinivasan, B., Pather, S., et al. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium* (June 1998).

[21] Strang, John, Mui, Linda, and O'Reilly, Tim *Termcap & Terminfo*. O'Reilly and Associates, 1996.

[22] Stevens, W. R. *UNIX Network Programming*, second ed., vol. 1. Prentice Hall, 1998.

[23] Vandoren, A. H. *Data Acquisition Systems*. Reston, 1992.

[24] Wang, Y.-C., and Lin, K.-J. Providing Real-Time Support in the Linux Kernel. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium* (June 1999).

[25] Yodaiken, V., and Barabanov, M. A Real-Time Linux. In *Proceedings of the USENIX Annual Technical Conference* (1997), USENIX.