Publications of the Astronomical Society of the Pacific
**105:** 616–624, 1993 June

# The Wyoming Infrared Observatory Telescope Software System

EARL J. SPILLAR
University of Wyoming Infrared Observatory, Department of Physics and Astronomy, University of Wyoming, Box 3905 University Station, Laramie, Wyoming 82071
Electronic mail: spillar@corral.uwyo.edu

DANIEL DUMBRILL
IOTC Incorporated, 211 South Third Street, Laramie, Wyoming 82070
Electronic mail: dumbrill@corral.uwyo.edu

G. L. GRASDALEN
G-Star Enterprises, 286 North Pennsylvania, Denver, Colorado 80209
Electronic mail: garyg@corral.uwyo.edu

R. R. HOWELL
University of Wyoming Infrared Observatory, Department of Physics and Astronomy, University of Wyoming, Box 3905 University Station, Laramie, Wyoming 82071
Electronic mail: rhowell@corral.uwyo.edu

*Received 1992 November 16; accepted 1993 February 26*

**ABSTRACT.** We describe the University of Wyoming telescope control and data-acquisition software system. The software was designed to be maintainable, portable, and inexpensive. Moreover, the software was designed to allow rapid communication between the hardware, the data-acquisition processes, and the tracking processes, while leaving each distinct. We show how the new real-time features embodied in the POSIX.4 standard and implemented in the Unix compatible LynxOS operating system allow us to perform all of our tasks on a single 80486 machine with a standard Unix-like environment, with outstanding real-time performance. We discuss our telescope pointing model, which allows us to point with a root-mean-square error of less than 5 arcsec over the sky with the 2.3-m telescope. For more detailed investigation and use, we will make the software available through anonymous FTP.

## 1. HISTORY AND INTRODUCTION

In 1986, the University of Wyoming Infrared Observatory was faced with an aging PDP-11 computer system and a floppy disk-based Forth operating system that were becoming difficult to maintain. New infrared array instruments were difficult to control with such limited hardware, and the Forth code appeared difficult to move from the PDP-11 to other architectures. Since we guessed that computer replacement every few years would become a way of life, we elected to write new, portable telescope tracking and data acquisition software in C on a real-time UNIX system. The new system was first implemented on a Masscomp 5500 system.

In 1991, we found the Masscomp, in turn, becoming obsolete and expensive to maintain. Moreover, the Masscomp had both a main processor and a data-acquisition coprocessor, which we found difficult to coordinate for the purposes of data acquisition. We ported the software to a new platform, a single IBM compatible with a 33 MHz 80486 processor running the LynxOS operating system, using the new POSIX.4 real-time software standards.

Through this migration process, we have developed a relatively machine independent telescope software package written in C. In this paper we describe our software and the rationale for the choices that we made, emphasizing the communication between processes and tasks abetted by the POSIX standard, in the hope that our experiences will be of use to others.

In Sec. 2 we list our desiderata, and explain our rationale for using a single machine to control both telescope and instruments. In Sec. 3 we discuss the required real-time features of the operating system. In Sec. 4 we describe the system itself. In Sec. 5 we summarize our pointing model. In Sec. 6 we discuss the ease of porting of the system, and in Sec. 7 we discuss future directions.

## 2. DESIDERATA

Tight coupling between data-acquisition software, data-analysis software, and telescope-tracking software is essential. For example, consider the acquisition of far-infrared data with a single element detector (Grasdalen et al. 1984). The chopping secondary changes position many times a second to sample a different portion of the sky. The whole telescope is moved on the sky ("nodded") between beams, or scanned to take a raster image of the sky. The data-acquisition hardware must be synchronized with both motions, and the data tagged with telescope and secondary status as they are acquired. Meanwhile, in order to make the system truly "friendly" to the user, the data ought to be demodulated and displayed as they are acquired, while the user brings up old data and perhaps modifies the data acquisition strategy in real time. Real-time data processing is also required for real-time data strategies, such as algorithms to center the telescope on bright sources based on incoming data. For each of these of interacting tasks, relative time delays of more than a few milliseconds would be undesirable, if not disastrous. It follows that the telescope system needs to be built on a true real-time operating system with strong interprocess communication.

Another important consideration is the need to add new instruments and functions to existing software with a minimum of difficulty. This work is done by various academics programming part-time, adding dollops of code to drive instruments, fix problems, or improve functionality, at random times. A simple and comprehensible system architecture eases the addition of new instruments and features.

A key question is whether to use a net of computers, or a single machine to accomplish as many tasks as possible. Since instruments are frequently developed by diverse investigators on different platforms, integrating these platforms is advantageous. Using several DOS machines also allows us to use a wide range of cheap hardware without writing device drivers. Simple operating systems, such as DOS or OS/2, can also be used on such machines. Many telescope systems are currently operated in this fashion.

Nevertheless, we chose to implement as many functions as possible on a single machine; although not the only solution, several arguments make such an approach attractive. Tight coupling between telescope control, data-acquisition, and data-analysis processes is easiest when they all reside on a single machine. Since there are very few true network operating systems, shared memory, messaging, or semaphores on a single machine are the simplest and most direct means of interprocess communication. One of the principle objections to using a single machine in the past was that it could not be done without using unusual and expensive software and hardware. Today, inexpensive, Unix-compatible real-time operating systems which run on inexpensive hardware are capable of performing all the required tasks at once. Furthermore, maintaining several machines is expensive. Each machine requires its own separate version of the software, which must be maintained, and all must be connected over a network, which in turn needs to be maintained. Each new CPU adds to system complexity and expense.

Our experience with the Masscomp strengthened our conviction that a single CPU is the best solution when practical. Masscomp 5500 had a 68020 main processor and a tightly coupled custom data acquisition processor which allowed very high data rates. Although presumably an excellent solution for some sort of data acquisition, we found synchronization between the processors quite difficult and cumbersome. Although the data-acquisition processor responded very quickly and reliably to interrupts, the complexity and difficulty of chaining the interrupts to the main processor and interactively changing the data acquisition processor's orders made complex coordinated operations very difficult.

In practice, many of our instruments still use their own host machines. Nevertheless, we feel emphasizing a central machine has simplified our programming task, and reduced the clutter in our control room.

## 3. THE CHOICE OF OPERATING SYSTEM

After considerable reflection, we chose real-time Unix as our operating system. The adoption of the POSIX.4 real-time standard allowed us to create a real-time system on a single machine in the comfortable and familiar Unix development environment.

### 3.1 The Operating System

In light of the desiderata listed in the last section, five requirements guided our choice of operating system. First, we require real-time response, since the telescope and data-acquisition system cannot wait while the system formats a floppy disk. Second, we require the system support standards so that the system will be portable, and programmers need not face a steep learning curve in approaching the system. Third, we require robust support for multiple users on multiple terminals. This allows terminals in the dome, in the control room, and remotely over phone lines, and simple serial line interfacing for new devices. Fourth, we require robust multitasking, since a user may be browsing a catalog, formatting a floppy disk, using kermit to access a remote computer, and running a data-acquisition routine, all while the telescope tracking software is running. Finally, we require quick and robust interprocess communication to coordinate data acquisition and telescope control. True multitasking, standardization, and support for many terminals strongly suggested that we choose a version of Unix.

The most stringent requirement is real-time performance. Most versions of the Unix kernel do not support real-time applications. The principle obstacles are built into the kernel itself. First, there are kernel routines which *cannot* be interrupted (preempted) before completion, and so have no guaranteed completion time. Second, the standard "fair" task scheduling algorithm will dynamically lower the priority of high-priority tasks. The delay before a high-priority interrupt is serviced or a high-priority task runs may not be visible to a user typing at a keyboard, but they may reach many milliseconds. Such delays are simply unacceptable for real-time applications.

Several Unix-like operating systems with rewritten kernels are available. We chose the LynxOS system (Singh and Bunnell 1990), from Lynx Incorporated, for several reasons. First, LynxOS is POSIX.1 compatible and implements the latest draft of POSIX.4 (see below). Second, the interrupt response time of the system is determinate under any load. Third, the system is available for many architectures, including the 80386, 80486, i860, 68030, 68040, 88110, Sparc, and PA-RISC. Sun Microsystems, Hewlett

Packard, and the Space Station Freedom, among others, are all using or recommending LynxOS for real-time applications.

Despite the current popularity of LynxOS, it would still be a risky choice were it not for the POSIX.4 standard. Although various vendors have independently arrived at similar solutions for real-time problems, most use different interfaces. For example, the system calls for starting threads vary widely from system to system. The POSIX.4 standard, now being adopted by many of these real-time operating systems, creates a common interface for most real-time programming tools, assuring their portability.

### 3.2 Real-Time Operating System Facilities

The fundamental goal of a real-time operating system is to simulate the operation of several interacting tasks running concurrently, despite the necessity to share processor cycles on a single CPU, and to facilitate speedy and reliable communication between these processes and the physical world. In order to fulfill these goals, the LynxOS kernel (and other real-time kernels) differ from standard Unix in several ways. In this section we describe several of these features, in the next section we will illustrate their use in the code itself. More detailed discussions of the features of real-time operating systems can be found in Fugelso and Michnovicz (1991) and Bunnell and Bunnell (1989).

One of the most important features of a system designed to take data is fast and consistent response to interrupts from hardware used to trigger data transfer. Maximum interrupt response time can be defined as the maximum time delay before the system will respond to the highest priority interrupt pending. Most operating systems, including standard Unix systems, *do not place any upper bound on interrupt response time.* In Unix, this is because most kernel service routines must run to completion once started, and have no well-defined maximum execution time. The LynxOS kernel is completely preemptable at any time; hence a maximum interrupt response time to the highest-priority interrupts can be defined. Of course, there are still some limits to the responsiveness of the machine: if there are too many interrupts of the highest priority, problems will occur. Nevertheless, we find LynxOS interrupt responses more than adequate. On our 80486 tracking and data-acquisition computer, we have reliably run data acquisition routines with intervals less than 80 $\mu$s between interrupts, while simultaneously running the tracking system, and compiling code. While low-priority user tasks run noticeably more slowly when the machine is under such a load, *no cycles of data acquisition or tracking are lost.*

Three logical levels of functionality appear in our code. At the lowest level, device drivers respond to interrupts coming from the hardware. Since the data are fleeting, the device driver must respond very quickly. Since other critical processes may be blocked while interrupts are being serviced, our interrupt drivers do the minimum possible work and then exit.

At a higher level, important data handling and storage tasks are handled by high-priority tasks running in background. Specific tasks (discussed in the next section) drive the telescope and control instruments.

At the highest level, user tasks set telescope and instrument control structures, while gathering and analyzing data. Since these tasks are not as time critical as control tasks, they run at the lowest priority.

Once the hardware has been serviced by an interrupt routine, lower-priority resident user processes handle the data. Since this is often conveniently handled by several closely communicating tasks running at various priorities, many operating systems are capable of running several *threads* of execution within one traditional program. Each thread can be regarded as a complete running program, separately scheduled by the kernel, except that several threads can share the single address space of what would be a single traditional process. Since all the threads share the same address space, communication and synchronization between the threads is more easily handled than if each thread were a completely separate traditional process. The same technology which enables user threads has been added to the LynxOS kernel, allowing the fast interrupt response times discussed in the last paragraph.

For synchronization between threads and processes, mutexs (a contraction of *mutual exclusion*) are implemented in the kernel. Their use can be illustrated by considering two tasks which, respectively, write and read from a single memory array. If the writing thread were interrupted by the reading thread in the midst of writing the array, the reading thread might read a garbled copy of both new and old data. The mutex is a memory "flag" which can be claimed by the first process before manipulating the array, and released upon completion. The second thread will block (stop executing) when it tries to claim the flag before reading the array, since only one thread can own the flag. When the first process finishes and relinquishes the flag, the second thread in turn claims the flag and reads the array, certain that no other thread can change the data. Since this type of control is intimately bound with the scheduling, mutexs are best implemented in the kernel. In cases where threads are implemented in "libraries," as in some versions of the Sun operating system, scheduling among threads in different processes is compromised.

Communication between threads in separate tasks can be accomplished in many cases by shared memory. In some cases, however, as continuous stream of data must be transferred, as through a traditional Unix pipe. Unfortunately, standard Unix pipes must be connected to both producer and consumer when both are started, making it impossible to change the consuming task. Changing consumers is extremely desirable, for example, when changing data-acquisition strategies. POSIX.1 adds FIFOs or "named pipes." These pipes can be created at any time, and are given a name in the file system. Both producer and consumer open the pipe by name and read and write from it as if it were a standard file. Therefore the consumer process can be dynamically attached at run time, and replaced as needed.

A major shortcoming of traditional Unix systems for real-time operation is the "fair" algorithm with which they allocate process or time to process. The algorithm (Bach 1986) guarantees that all tasks, including those of the lowest priority, will receive a slice of processor time occasionally. In order to achieve this, the kernel will dynamically lower the priority of even the highest-priority tasks to free processor time. While this is salutary to tasks running in the background, a data-acquisition task cannot afford to be interrupted. The LynxOS scheduler never changes priorities: the highest priority task runs until it is blocked by the need for a resource, or it puts itself to sleep. Thus the user is assured that when the tracking code needs to point the telescope, it will not be preempted by another task, such as a user asking for a graphic analysis of his data. The drawback is that low-priority tasks, such as text editors, may experience delays. We have found user delays to be acceptable in all circumstances. An inconvenience is the new ability to create uninterruptible high-priority rogue processes running in an infinite loop, for which there is no solution but rebooting the machine. In practice we have found these arise only when debugging new code.

The greatest disadvantage of using a non-MSDOS system on the PC platform is that the DOS BIOS routines and handlers shipped with many cards will not run inside the alien Lynx environment. This means that new device drivers need to be written for some devices. We must confess that writing device drivers is not one of our favorite pastimes. However, most real-time systems, including LynxOS, supply sample drivers (including source code) for most standard devices, including displays, disk drives, ethernet cards, serial port cards, and so on. Moreover, device drivers can be dynamically loaded into the system while it is running. This means new versions of a driver can be tested without rebuilding the kernel, as is necessary in most Unix systems.

One difficulty is that the standard PC-AT architecture does not provide enough slots, or enough hardware interrupts (IRQ's). This limits the number of data-acquisition cards that can be conveniently added to the system.

Although we have discussed these features as present in LynxOS, other systems share them. The C interfaces to these features are defined in the POSIX.4 draft, so that we expect to be able to transport our code to new operating systems if the need arises.

### 3.3 The Language

For a real-time system, once the operating system is chosen, the language is a foregone conclusion. For almost any version of Unix, C is the language of choice for system programming. C's popularity has other advantages. A huge collection of software is available bundled with most Unix systems, on internet file servers, and on such services as compuserve. Some examples are the standard Unix math library, the curses screen control routines, graphics programs such as GNUPLOT, and ephemeris programs such as AA. Since the source is available, these easily available tools can be integrated into the telescope software system.

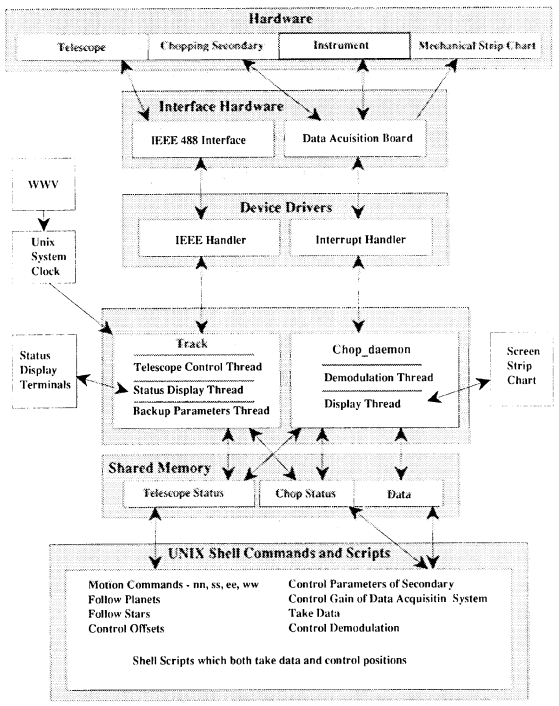

FIG. 1—The software architecture of the tracking system. Blocks indicate independent programs or hardware; lines indicate control or information transfer.

### 3.4 The Application Programming Interface: POSIX.1

C differs from some other languages in that early C left most I/O and control statements completely unspecified. Even statements to do things as simple as printing to the screen were left deliberately undefined in order to make the core language as portable as possible. The ad hoc definitions in the "standard library" were all that was available, and many run time systems differed in subtle ways. The several flavors of Unix have introduced several incompatible interfaces.

The ANSI POSIX.1 standard (not to be confused with the POSIX.4 real-time standard) defines a set of standard interfaces. POSIX.1 covers such system services as serial lines, file systems, time and date, task priorities, and interprocess communication. The standard has become a requirement for fulfilling many federal government contracts; this means that most major vendors are shipping POSIX.1 compliant systems. Not only will most Unix systems soon have POSIX.1 system calls available, but so will VMS, Windows NT, and many IBM systems. Most of our system calls are now POSIX.1 calls.

## 4. THE SOFTWARE

In this section we survey the architecture of the software. Figure 1 presents the overall architecture of the tracking and data-acquisition system. On the left-hand side, the structures for telescope control are presented. On the right-hand side, chopped photometry is presented. The logic of real-time operation dictates similar layers for both systems, which are arranged vertically in the figure.

## 4.1 The Tracking System

The main tracking routine "Track," compiles from approximately 1500 lines of C code. It tracks the telescope, keeps the dome aligned, displays telescope status on display terminals, and archives parameters for later use.

The interface to the telescope hardware consists of several parallel ports which read the position of the telescope and dome via encoders, report on status switches and hand paddles, and send rate commands to the telescope hardware. Because of the huge inertia of the telescope, new rates need only be commanded at a rate of roughly 10 Hz, so that the encoders are read at a relatively slow rate. We use commercial parallel interfaces controlled through an IEEE 488 interface. The IEEE interface is controlled through a fast serial port; higher data rates for instrument control could be supported by a bus card. Currently the IEEE handler is situated inside the tracking code.

"Track" itself runs three separate threads of execution once started. The highest-priority thread monitors the encoders on the telescope, calculates the desired position, and commands telescope rates. It puts itself to sleep for roughly a tenth of a second until the next position check.

A second thread, running at lower priority, runs the status display screens. These can be commanded to display on an arbitrary set of terminals when the "Track" routine is started.

The third thread, running at lower-priority still, records a snapshot of shared memory to disk every few seconds. This image is loaded when the system is restarted: thus all important pointing constants are recovered, even after an accidental power outage.

The POSIX threads facility makes this separation easy. In Listing 1, we show a code fragment from the "Track" routine which starts three threads. (Note that all listings have been edited, leaving only the few lines of code relevant to the discussion.) The first set of commands create structures which hold the attributes of the threads. After setting their priorities, the threads are launched by specifying the functions to be performed (trackloop(), do_screens(), and do_store()) and calling pthread_create. The functions are normal C functions. After each pthread_create(), the main program proceeds to the next line while the launched thread runs independently. When the thread finishes or dies, it informs the kernel. The command pthread_join() pauses until the thread referred to by its first argument has finished running.

Because of the high priority we have assigned it, the highest-priority task trackloop() will always execute at assigned intervals, regardless of other tasks on the system. The deterministic scheduling of the LynxOS kernel means no lower priority task can ever preempt it. Some even more time critical data acquisition tasks are assigned similar or higher priorities; if they require too many CPU cycles, the lower-priority display threads will stop running completely before tracking is affected at all.

Listing 1
Track, the Main Telescope Tracking Program

```
/* track.c - the main telescope tracking program */
main()
{
/* Create thread attributes - handles by which threads are manipulated */
     pthread_attr_create(&thatScreen);
     pthread_attr_create(&thatTrack);
     pthread_attr_create(&thatStore);

 /* Set priorities for the threads*/
     pthread_attr_setprio(&thatScreen,SCREENPRIORITY);
     pthread_attr_setprio(&thatTrack,TRACKPRIORITY);

/* Launch threads - each command returns immediately,
    while the named routine  runs independently */
     pthread_create(&thTrack, thatTrack,trackloop,&totask);
     pthread_create(&thScreen, thatScreen, do_screens, &totask);
     pthread_create(&thStore, thatStore, do_store, &totask);

/* Wait for the threads to exit before before quitting*/
     pthread_join( &thTrack,  stat);
     pthread_join( &thScreen, stat);
     pthread_join( &thStore, stat);
}


/************************** Main tracking loop ***************************/


void trackloop(void)
{
/* Set a timer to restart the thread periodically */
     setitimer(ITIMER_REAL,&interval&oldinterval);

/* The loop itself */
     while(tinfo->keep_tracking != TRACK_STOP)  {
/* Wait for the timer to tick */
          sigwait(mask, &v);

           do_encoders();            /* read telescope encoders */
           do_times( );              /* read and set time variables */
           do_angles( );             /* Calculate telescope angles */
           do_corrections( );        /* Correct for flexure etc. */
           do_tracking( );           /* Calculate object position */
           do_rates( );              /* calculate desired rates */

           gpib_wr(drive_adr, (lpchar) &(drives.dome), 5);
   /* Write instructions to the IEEE488 interface */
      }
}
```

In order to make it easy for user supplied code to access the system, "Track" communicates through shared memory (see Fig. 1). In Listing 2 we show a fragment of a program which offsets the telescope. Once a user function has opened shared memory, manipulating the telescope is simply a matter of changing variables. As another example, the code which directs the telescope to follow an object gets the coordinates from a data file, processes and nutates them, performs other corrections, and places the topocentric coordinates in shared memory.

Accurate timekeeping is essential when tracking a telescope. The LynxOS system clock ticks every millisecond, as opposed to the roughly 57-ms period of the standard

Listing 2
User Commands and Manipulating Shared Memory

```
/* Telescope variables are spelled  with capital letters. */
/* They are aliased to variables in the shared memory structure */
/* tinfo defined in wiro.h; e.g.  #define OFFSET_HA  tinfo->offset_ha */

struct  wiro_memory *tinfo;

main( argc, argv )
{
/*First, we open shared memory, so we can directly access
    tracking parameters */
      tinfo = (struct wiro_memory *)
            smem_get("WIRO_MEMORY",sizeof(*tinfo),
            SM_READ | SM_WRITE ) )

      if ( strcmp(argv[0],"zero") ==0 ) {
   /* User command:  set offsets to zero */
                  OFFSET_DEC = 0.0;
                  OFFSET_HA = 0.0;
                  exit(0);
      }

      if ( argc == 2) {
            sscanf( argv[ 1 ], "%lf", &val );
            if (strcmp(argv[0],"nn") == 0) {
/*  User command:  move telescope north  */
                  OFFSET_DEC += val;
            exit(0);
      }
}
```

DOS clock. This is accurate enough for most astronomical purposes, so we use the standard unix clock through system calls in the tracking code. Since PC hardware clocks can drift several seconds over a night, synchronization with a time source such as WWV is crucial. To accomplish this, a separate process monitors a hardware clock synchronized to WWV, and in turn synchronizes the PC hardware clock every few seconds.

### 4.2 Chopped Data Acquisition

Infrared photometry is difficult because the signal from the object is superimposed on a much larger, variable, sky background. In order to accurately subtract the sky background, a chopping secondary mirror switches the beam between the object and sky several times a second. The flux from the source is determined by demodulating the resulting alternating signal (Grasdalen et al. 1984). Meanwhile, the telescope is periodically "wobbled" so that the source is not always in the same beam. Doing all of this requires coordination between the data-acquisition system, the telescope system, and the chopping secondary controller.

Our software to accomplish this task is presented in the right half of Fig. 1. The signal from the instrument is conditioned by a preamplifier and filter which are not shown in the figure. An inexpensive multifunction board from Metrabyte digitizes data from the instrument, and controls the secondary and the mechanical strip chart. Timing is controlled through a periodic interrupt generated by timers on the board. The interrupt triggers three actions by our software interrupt handler. First, the A/D converter is read and the data are stored in a data buffer for handling by the chop_daemon. Second, a new position for the secondary chopping is read from an array and written to one of the D/A converters. Third, a second value which was stored by the chop_daemon is transferred to the second D/A converter. This is normally a demodulated signal for the strip chart.

Since we wrote the device driver, we were able to define the read, write, ioctl, and other system calls in a somewhat nonstandard way, which nonetheless fits the logic of the situation better than some more traditional approaches (see listing 3 for an example of these calls). Like any device driver, ours performs as few functions as possible very quickly. In order to prevent the loss of any data, it is normally run at the highest priority. All heavy calculations are performed in due time by the background task chop_daemon.

An important goal was to have a real-time data display of demodulated data running continuously, even when data were not being recorded, such as while peaking up on a source. Therefore data demodulation and display are handled by a background task named chop_daemon (see listing 3). chop_daemon is launched at the beginning of an evening, and in turn launches the device driver. Commands for chop_daemon are placed in shared memory by user commands. chop_daemon reads and writes data from the interrupt handler through read and write commands, which block when data are not yet available. The data are

Listing 3
The Data-Acquisition Daemon

```
/* chop_daemon.c */
main( argc, argv )
{
/* Install the dynamically loadable device driver to control
    the I/O card
*/
     install_driver();

/* Open the I/O card for read and write access
    through the driver just installed
*/
     ciofile=open("/dev/cio",O_RDWR);

/* Open a FIFO, or named pipe.  This is a data stream which
   can be opened later by any data acquisition
   program on the system.
*/
     chopFIFO = fopen( "/usr/local/wiro/files/chopFIFO", "w" );

/* Attach to the data ready semaphore.   This is set by
    the device driver when data has been stored in the
    data buffers.
*/
     sem_num = sem_get("DATA_READY",0) )

/* Configure the data acquisition card:
    Set the period of the chop cycle,
    load an array of offsets to define the waveform,
    and set the number of data channels to be read.
*/
     ioctl(ciofile, PERIOD, &period);
     ioctl(ciofile,WAVEFORM, &waveform);
     ioctl(ciofile, CHANNELS, &CHOP_CHANNELS);

/* The data collection loop
*/
     while (LOOP) {
/* We are about to acquire and process data - lock out
    reading processes
*/
         pthread_mutex_lock( &data_ready_mutex );
/* Read data to the array b */
        read(ciofile, data, n_points);
/* Demodulate the data here - code omitted */
/* Write a value to the strip chart */
          write(ciofile, b, 1);

/*  A data collection program acquires data by opening the FIFO for
     reading,  and setting CHOP_FIFO_LENGTH to the number of data
     desired.
*/
          if ( CHOP_FIFO_LENGTH > 0 )  {
               fwrite( &STRIP_CHANNEL[0], sizeof( STRIP_CHANNEL[0]), \
               CHOP_CHANNELS, chopFIFO );
               fflush(chopFIFO);

               CHOP_FIFO_LENGTH--;
          }

/* Unlock the mutex protecting the data structures */
          pthread_mutex_unlock( &data_ready_mutex );
     }
/* close the file representing the I/O card and remove its device driver */
     close(ciofile);
     remove_driver();
}
```

demodulated, for example, by convolving it with a sine function. The deconvolved signal is immediately written to the strip chart and screen display, whether data are being consumed by any other task.

When data are being used by a consumer task, or being used by a task which implements an observing strategy, the task places the number of samples it wants in shared memory and opens a FIFO. chop_daemon opens the FIFO and writes the desired number of samples to it. Thus, data-acquisition tasks are dynamically patched into the data stream as needed.

User programs are available to set variables in shared memory, such as the shape of the demodulating function and the position of the secondary as a function of time. The data structures are protected by a mutex when the program is executing its main loop to prevent control tasks from changing control variables while data are being processed.

### 4.3 Controlling Cameras

Infrared cameras (Spillar et al. 1990; LeVan 1990) were run with the Masscomp system, and are currently being ported to the LynxOS system. Although many instruments are based on private host computers, we find it advantageous to base instruments on the tracking system. Both tracking and data-acquisition commands can be executed from a single terminal, through a single shell script, without needing to communicate between machines. A single host can easily produce a single log of all system events, and has easy access to the telescope positioning system.

## 5. THE FORMULAS USED IN TELESCOPE TRACKING

Most of the formulas needed to track a telescope are available in the literature, see, for example, the *Astronomical Almanac* (1992), Montenbruck (1989), Meuus (1982), and *The Explanatory Supplement to the American Ephemeris and Nautical Almanac* (ed. Seidelman, 1992). An overview of the process of driving telescopes may be found in Trueblood and Genet (1985). Therefore, we only briefly refer to the equations, stating the approximations we have adopted, and list their order of application. We discuss our pointing corrections in greater detail.

### 5.1 Applying the Formulas

When a new object is acquired, its topocentric position is calculated by a user program run from the shell. The topocentric position is the position of the object on the celestial sphere as observed from the surface of the Earth in the absence of an atmosphere. These coordinates are obtained by applying steps 1 through 4 below. The coordinates, including the time of calculation and any motion against the sky (in the case of planets) are stored in shared memory, and the program terminates.

Every 10th of a second, the main tracking loop wakes up. It calculates the LST and the encoders for the telescope position are read. Step 5 is applied to arrive at the topocentric coordinates at which the telescope is pointed. The desired coordinates are calculated from the data stored in shared memory by the user program, taking into account elapsed time since the object was acquired. Step 6 calculates the desired telescope rates, which are sent to the telescope, and the thread puts itself to sleep until the next cycle.

The steps are:

(1) Calculate the local sidereal time and Julian date. We use the Unix system time through a standard system call to calculate the LST and JD. We neglect the correction between TDB and TDT, which is under 2 ms, and the difference between GMST and GAST, which may be up to a second, but changes slowly. Such errors will be absorbed in the pointing correction constants for the telescope (see below). An excellent overview of astronomical time may be found in the *Explanatory Supplement to the American Ephemeris and Nautical Almanac.*

(2) Precess the coordinates. Although formulas for precession are given in many references, the conversion from the B1950 system to the J2000 system is not often covered. Smith et al. (1989) and Yallop et al. (1989) discuss the theory and practice of converting between these systems, as well as precession.

TABLE 1
The Terms in the Pointing Model

| Term Name | Value (arcseconds) | Description |
|---|---|---|
| $C^{\alpha}_{Refraction}$ | 66.66 | Refraction term, Hour Angle |
| $C^{\delta}_{Refraction}$ | 39.81 | Refraction term, Declination |
| $C^{\alpha}_{Flexure}$ | 120.72 | Flexure term, Hour Angle |
| $C^{\delta}_{Flexure}$ | -2.31 | Flexure term, Declination |
| $C^{\alpha}_{M.Az.}$ | 9.68 | Misalignment of the Polar Axis in Azimuth |
| $C^{\delta}_{M.Az.}$ | 9.68 | Misalignment of the Polar Axis in Azimuth |
| $C^{\alpha}_{M.El.}$ | 143.15 | Misalignment of the Polar Axis in Elevation |
| $C^{\delta}_{M.El.}$ | 133.14 | Misalignment of the Polar Axis in Elevation |
| $C^{\alpha}_{NP}$ | -17.21 | Non-perpendicularity of the Axes |
| $C_{E1}$ | -48.89 | empirical term 1 |
| $C_{E2}$ | -5.11 | empirical term 2 |

(3) Calculate the constants of nutation, and apply the correction. These are derived in the standard references listed above.

(4) Compensate for annual aberration. We are fond of the procedures presented in Trueblood and Genet (1985) and Smart (1979).

(5) Apply the pointing corrections to the measured telescope position. These are discussed in the next section.

(6) Calculate the desired telescope velocity. After all of the above corrections, we have calculated where the telescope is pointing, and where it should be pointing. The WIRO telescope servo system works by reading position, comparing it to a desired position, and commanding a new telescope velocity. If $T$ is the true coordinate value and $D$ is the desired value, the commanded rate $R$ is given by

$$R = A(D-T)/\mathrm{ABS}(D-T) + B. \tag{1}$$

$A$ and $B$ are system-dependent constants. The same algorithm is applied in both right ascension and declination. At this point the dc motors which drive the telescope are commanded to turn at the desired rate, and the thread sleeps until the next cycle.

### 5.2 The Pointing Model

Our pointing model includes terms to account for flexure, refraction, misalignment of the polar axes, and the fact that the right ascension and declination axes are not exactly perpendicular. (The WIRO 2.3-m telescope uses an equatorial mount.) To determine the coefficient of each term, observations of approximately 100 stars are obtained during a few hours one night. The errors in position are fit using the pointing model, and used for future observations. In this section we discuss the terms in our physical model. We give the current coefficients in Table 1.

Note that whereas right ascension ($\alpha$) increases to the east on the sky, the hour angle ($h$) increases to the west. Declination is denoted $\delta$, altitude is denoted ALT, and the latitude of the observatory LAT.

#### *5.2.1 Refraction*

First, consider refraction. Refraction moves an image of a star higher in the sky. For typical pressure and temperature conditions, deflection is 50″ at an altitude of 45° (see, for example, *Explanatory Supplement to the American*

*Ephemeris and Nautical Almanac*, pp. 140–145). In fact, we allow for differing constants in right ascension and declination, which are fit empirically (we would like to include a temperature dependent term as well). Elementary calculations show an image of the star moves by

$$\Delta h = C^{\alpha}_{\text{Refraction}}(\sin h \cos \text{LAT})/\sin \text{ALT} \cos \delta,$$
$$\Delta\delta = C^{\delta}_{\text{Refraction}} \frac{(\sin \text{LAT} - \sin \text{ALT} \sin \delta)}{\cos \delta \sin \text{ALT}}. \tag{2}$$

### *5.2.2 Tube Flexure*

For a rotationally symmetric telescope tube, flexure would also displace an image purely in altitude. However, because the tube is not symmetric, the actual equations use different coefficients for $H$ and for $\delta$. The sign of the altitude correction must be determined empirically, since it is not obvious whether the end of the tube or the mirror cell will flex more, displacing the image up or down. The corrections are given by

$$\Delta h = C^{\alpha}_{\text{Flexure}} \sin h \cos \text{LAT}/\cos \delta,$$
$$\Delta\delta = -C^{\delta}_{\text{Flexure}}(\sin \text{LAT} - \sin \text{ALT} \sin \delta)/\cos \delta. \tag{3}$$

### *5.2.3 Misalignment of the Polar Axis*

Next, we correct for slight misalignments of the polar axis of the telescope. First, there are two terms due to the misalignment of the telescope in azimuth

$$\Delta h = -C^{\alpha}_{M.Al.} \cos h \sin \delta/\cos \delta,$$
$$\Delta\delta = C^{\delta}_{M.Al.} \sin h. \tag{4}$$

Next, there are two terms due to the misalignment of the polar axis in elevation

$$\Delta h = C^{\alpha}_{M.El.} \sin h \sin \delta/\cos \delta,$$
$$\Delta\delta = C^{\delta}_{M.El.} \cos h. \tag{5}$$

### *5.2.4 Skewness of the Axes*

There is a term due to the fact that the two axes are not exactly perpendicular to each other

$$\Delta h = -C^{\alpha}_{NP} \sin \delta/\cos \delta. \tag{6}$$

We also include two empirical terms whose physical origin is uncertain

$$\Delta h = -C_{E1} \cos h - C_{E2} \cos 4h. \tag{7}$$

Both terms might be due to irregularities in the gear which drives the telescope in right ascension.

### *5.2.5 Worm Gear Errors*

Since there are small errors in our worm gears, we include an empirical correction $\Delta h$ depending only on the current rotation angle of the worm gear that drives the spur gear which rotates the telescope in HA. The values are obtained from a look-up table in the software.

### *5.2.6 User Definable Terms*

Four other terms with user definable parameters are included. Collimation errors,

$$\Delta h = K^{\alpha}_{c},$$
$$\Delta\delta = K^{\delta}_{c}, \tag{8}$$

account for the fact that the detector may not be centered in the focal plane. Encoder errors,

$$\Delta h = K^{\alpha}_{e}/\cos \delta,$$
$$\Delta\delta = K^{\delta}_{e}, \tag{9}$$

account for small errors in the time base and the zero points of the encoders. Typically, these four constants will be set the first night of a run and remain relatively unchanged until a new instrument is mounted. Small adjustments to these parameters may be made to optimize the pointing over a small section of the sky.

## 6. EFFORT REQUIRED FOR PORTING

In order to judge how well we have succeeded in creating a portable system, consider the latest move of the software from the Masscomp 5500 to the 486/33. The main tracking routines were moved to the 486 in a month during the fall of 1991 by one of us working roughly 2/3 time. At the same time, most of the current POSIX system calls were installed. In the next two months, the chopped data-acquisition system was implemented, the greatest effort going toward writing new device drivers for new data-acquisition boards and additional serial ports. Over the next few months, 2–6 h a week were spent fixing bugs, writing documentation, and installing new instruments. At the beginning of the project, the programmer was familiar with C and Unix at a user level, but fairly unfamiliar with most system calls, and totally unfamiliar with writing device drivers.

Moving to new hardware should be easier. First, many unportable parts of the first version of the code written on the Masscomp, which was not POSIX compliant, have been rewritten. Second, perhaps 1/3 of the effort was devoted to learning to write and writing device drivers; however drivers for many usable devices are now available directly from Lynx. Therefore, we expect that most of the system can be moved to a POSIX compliant system with suitable device drivers in two months.

## 7. FUTURE DIRECTIONS

The WIRO software will never be complete; new instruments and capabilities will continue to be added far into the future. Two improvements are especially important.

First, we wish to improve the maintainability of the current code. The major difficulty in adding new features or repairing the existing code is understanding which pieces of existing code will be affected, and how to interact with that code. These problems are addressed by the technique of object-oriented programming (OOP). The goals of object-oriented design are to improve the portability, maintainability, reusability, comprehensibility, and extensibility of software. This is accomplished by breaking the code into "objects" which encapsulate the software's data and functions, isolating changes in different modules from each other. New object-oriented languages, while not strictly necessary for object-oriented programming, encourage and abet the isolation of factors within the objects (Cox and Novobilski 1991).

The logical object oriented route from C is to C++ or Objective-C. Among the attractive features of C++ are the availability of good freely available compilers such as the GNU C++ compiler from the Free Software Foundation, which can compile the existing C code directly, and the fact that its current popularity in the programming community is approaching that of C. We expect we will slowly migrate the current system software to C++.

A second difficulty is the nonintuitive nature of the Unix shell. It is too easy for an occasional user to forget the exact syntax for a command. Graphical user interfaces have gained prominence because they are easy to comprehend, remember, and manipulate. Our principle considerations in choosing a GUI are the ability to work over a network (for remote observing), portability, and ease of use. X windows was designed for network operation, and has been ported to many operating systems. Although the raw programmer's interface is complex, "widget sets" are available which encapsulate much of the difficulty. We are currently installing an X-windows interface to the system.

## 8. CONCLUSIONS

In the past, writing telescope systems required the use of arcane real-time operating systems. In order to address the hardware and run quickly enough, much of the code was often written in assembly language. The results were likely to be tied to a single machine's architecture. Adding new instruments often meant adding a new computer to the control room. The code was not portable, and was difficult to maintain and debug. Telescope systems were rarely reused; they were rewritten.

The development of real-time Unix kernels and the promulgation of the POSIX.4 real-time standard provide stable, machine-independent standards on which to build telescope systems. The arrival of very fast IBM compatible systems provide an inexpensive hardware platform powerful enough to track a telescope and take data at the same time, even when programmed in a high level language. In the near future, object-oriented programming techniques will make it easier to add new software modules without needing to extensively change existing code.

We have developed a portable telescope control system built on these standards. The tracking module has already proven itself to be easily portable between machines. In the coming years, as new hardware systems replace current ones, we will be able to spend our time improving the system and adding features, rather than rewriting the existing system.

## REFERENCES

Astronomical Almanac, 1984 (Washington, U.S. Government Printing Office, and London, Her Majesty's Stationary Office)

Bach, M. J. 1986, The Design of the Unix Operating System (Englewood Cliffs, NJ, Prentice-Hall)

Bunnell, M., and Bunnell, M. 1989, Dr. Dobbs Journal, 152, 36

Cox, B. J., and Novobilski, A. J. 1991, Object-Oriented Programming—An Evolutionary Approach (Reading, Addison-Wesley)

Fugelso, D., and Michnovicz, M. 1991, The C Users Journal, May, 48

Grasdalen, G. L., Hackwell, J. A., and Gehrz, R. D. 1984, PASP, 96, 1017

LeVan, P. D. 1990, PASP, 102, 190

Meuss, J. 1982, Astronomical Formulas for Calculators (Richmond, Willmann-Bell)

Montenbruck, O. 1989, Practical Ephemeris Calculations (New York, Springer)

Seidelmann, P. K., ed. 1992, Explanatory Supplement to the American Ephemeris and Nautical Almanac (Washington, U.S. Government Printing Office, and London, Her Majesty's Stationary Office)

Singh, I. M., and Bunnell, M. 1990, Seventh IEEE Workshop on Real-time Operating Systems and Software

Smart, W. M. 1979, Textbook on Spherical Astronomy (Cambridge, Cambridge University Press)

Smith, C. A., Kaplan, G. H., Hughes, J. A., Seidelmann, P. K., Yallop, B. D., and Hohenkerk, C. Y. 1989, AJ, 97, 265

Spillar, E., Johnson, P. E., Wenz, M., and Warren, D. 1990, in Instrumentation in Astronomy VII, Soc. Photo-Opt. Instrum. Eng., 1233, 63

Trueblood, M., and Genet, R. 1985, Microcomputer Control of Telescopes (Richmond, Willmann-Bell)

Yallop, B. D., Hohenkerk, C. Y., Smith, C. A., Kaplan, G. H., Hughes, J. A., and Seidelmann, P. K. 1989, AJ, 97, 274