THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Microcomputer Control of Apparatus, Machinery, and Experiments

By B. C. WONSIEWICZ, A. R. STORM, and J. D. SIEBER
(Manuscript received January 27, 1978)

Microcomputers, operating as satellite processors in a UNIX system, are at work in our laboratory collecting data, controlling apparatus and machinery, and analyzing results. The system combines the benefits of low-cost hardware and sophisticated UNIX software. Software tools have been developed that accomplish timing and synchronization; data acquisition, storage, and archiving; command signal generation; and on-line interaction with the operator. Mechanical testing, magnetic measurements, and collecting and analyzing data from low-temperature convective studies are now routine. The system configurations used and the benefits derived are discussed.*

The vision of an automated laboratory has promise: computers control equipment, collect data, and analyze and display results. The experimenter, freed from tedium, devotes more energy to creative pursuits, presumably research and development. Unfortunately, the vision has proved to be a mirage for more than one experimenter who, after a year of learning the mysteries of hardware and software, finds the control of experiments as far away as ever.

This paper describes a system for laboratory automation using the UNIX time-sharing system that has permitted experiments to be automated in hours rather than years. This is possible because the

* UNIX is a trademark of Bell Laboratories.

UNIX system makes programming easy, standardized hardware solves many interfacing problems, and a library of programming tools performs many common experimental tasks. The complex task of automating an experiment reduces to the simpler tasks of assembling hardware modules, selecting the appropriate combination of software tools, and writing the program. The experimenter need not know the details of how signals are passed from one hardware module to another.

The benefits of automation are illustrated here by examples taken from the laboratory. Among these are very precise data logging, simplified operation of complex machinery or experiments, quick display of results to the operator, easy interfacing to data analysis tools or graphics, and ease of cross-correlating among experiments.

I. APPROACHES TO AUTOMATION

A variety of approaches have been made to the problem of laboratory automation. Systems can be designed for the job at hand, or they can be designed to be multipurpose. A computer may be devoted to a single experiment, or one computer may be shared among several experiments.

1.1 Job-specific automation

One approach to automation is to tailor a system to a specific problem, either by developing dedicated hardware or by developing job-specific software. A modern digital multimeter affords a good example. Some multimeters employ specially designed digital circuitry to accomplish a myriad of operations (ac-dc voltage and current readings, resistance, autoranging or preset scales, sampling times, etc.), while others incorporate a microprocessor with specific software to accomplish the same functions.

A drawback is that the design can be too specific. Changes in the operation of the device can be made only by rewiring the circuit or by rewriting the program. Since the operation of a digital multimeter changes slowly, the job-specific development is practical. If enough instruments are sold to recover the high costs of specific design, the approach is economical.

Larger examples of job-specific design are to be found in the automation of widely used scientific apparatus such as gas chromatographs and x-ray fluorescence machines, where the high cost of software and hardware can be amortized over many units and where

the function of the apparatus changes slowly. Unfortunately, there are other examples where the automation never quite worked properly or could not be changed to meet changing needs, and still others where the high cost of developing the job-specific design was never recovered.

1.2 Multipurpose automation

In fact, many cases of automation in research and on the production line are antithetical to the conditions for job-specific development. The tasks are varied, they change rapidly, and the number of identical installations is small. Such applications are best served by a system that is versatile and easily changed.

A system can be multipurpose only if the hardware and the software are multipurpose. Flexibility in hardware at the module level is illustrated by the multimeter, which performs a variety of functions by virtue of the specific hardware and software it contains. Hardware versatility at the system level is achieved by the ease of interconnecting or substituting various modules, meters, timers, generators, switches, and the like. Recently, standards for interfacing instruments have been gaining acceptance. Many modern instruments offer the IEEE instrument bus;[1] nuclear instrumentation follows the CAMAC standard[2] which permits higher data rates and larger numbers of instruments than the IEEE bus.

One might think that because typing is easier than soldering, it should be easier to change software than to change hardware. However, the ease of changing software depends on the language at hand, the quality of the editor, the file system structure, etc. The very features of the UNIX time-sharing system that make it suitable for system development[3] also make it suitable for automating a laboratory or a production line. Most of the work in automation is in the software, and UNIX makes developing the software easy.

1.3 Multipurpose software

Software is changed more readily if it is well designed and cleanly implemented. Quite small programs can meet a large number of needs if they fit Kernighan and Plauger's description of a software tool:[4]

> it uses the machine; it solves a general problem not a special case; and it's so easy to use that people will use it rather than building their own.

Naturally, such tools will not meet every need, but they will serve for most tasks. They are usually easily modified to meet special needs.

1.4 Stand-alone systems

An experiment can either have a computer devoted exclusively to it or it can share the computer with other experiments. The stand-alone approach has several virtues; chiefly, no one else can preempt the computer at a crucial instant (real-time response) and no one else's errors can cause the system to crash (independence). Real-time response and independence must be traded off against the high cost of hardware and software. While the cost of the central processor and memory have declined significantly in recent years, the cost of terminals, graphics, and mass storage devices has remained high.

It should be remembered that most of the cost of automation is in software development. The software development tools on most stand-alone systems are primitive by UNIX standards and result in substantially higher total development costs.

1.5 Shared systems

Connecting several experiments to a well-equipped central computer shares the costs of expensive peripherals and may provide a reasonable programming environment. Simple experiments which do not require real-time response can be interfaced to a time-sharing system directly. UNIX time-sharing can be used for this purpose if the data rates are slow enough or if time delays can be tolerated. For example, an x-ray diffractometer might be interfaced as an ordinary time-sharing user since data are normally taken every five seconds or so. If there is some delay due to the load on the time-sharing system in positioning the diffractometer for the next reading, nothing but time is lost.

Some central computers do provide a system which will guarantee real-time response. The MERT system is an example of one which provides a very sophisticated real-time environment aimed at users capable of writing system level programs.[5] Nevertheless, writing programs for real-time response on a shared system is a complex task that must be done with care because a single experiment can bring down the entire system. In the past, big shared central computers have been unsuccessful in controlling many experiments at once, although recent reports indicate some success.[6] In general,

such a system is forced to rely on a small amount of very reliable code to shield the users from one another and from the system.

1.6 Satellite processors

The real-time response and independence of a stand-alone system can be combined with the lower costs and superior software of a shared system if inexpensive, minimally equipped microcomputers are connected to a large, well-equipped central processor. The microcomputer or satellite processor (SP) provides real-time response and independence when the experiment is in progress; the central processor (CP) provides data storage, an excellent programming environment, and data analysis tools. Not only is the cost of such a system significantly less than the cost of an equivalent number of stand-alone systems, but the CP also provides time-sharing services for data analysis and reduction, document preparation, and general-purpose computing.[7,8]

II. SYSTEM DESCRIPTION

In our laboratory, we have automated several experiments with the distributed processing system shown schematically in Fig. 1. The CP provides UNIX time-sharing service to 16 incoming phone lines and supports up to 16 SPs for experimental control or as ordinary time-sharing users.

2.1 The central computer

A Digital Equipment Corporation (DEC) PDP-11/45 with 124K of core storage, cache, 80M bytes of secondary disk storage, and a tape drive serves as the central computer. Graphics are provided by a high quality terminal (Tektronix 4014) and a hard-copy printer plotter (Versatec 1200A). Data can be transmitted to other computer centers with an automatic calling unit and a 2000-baud synchronous phone link.

The central facility is similar to an ordinary UNIX installation, offering a wide range of time-sharing services. Like other UNIX systems, many jobs are program development or document preparation. An unusually large number of jobs are numerical analysis run in Fortran or Ratfor[9] or graphical displays using the UNIX **graph** command. Fortran is used because of the availability of many

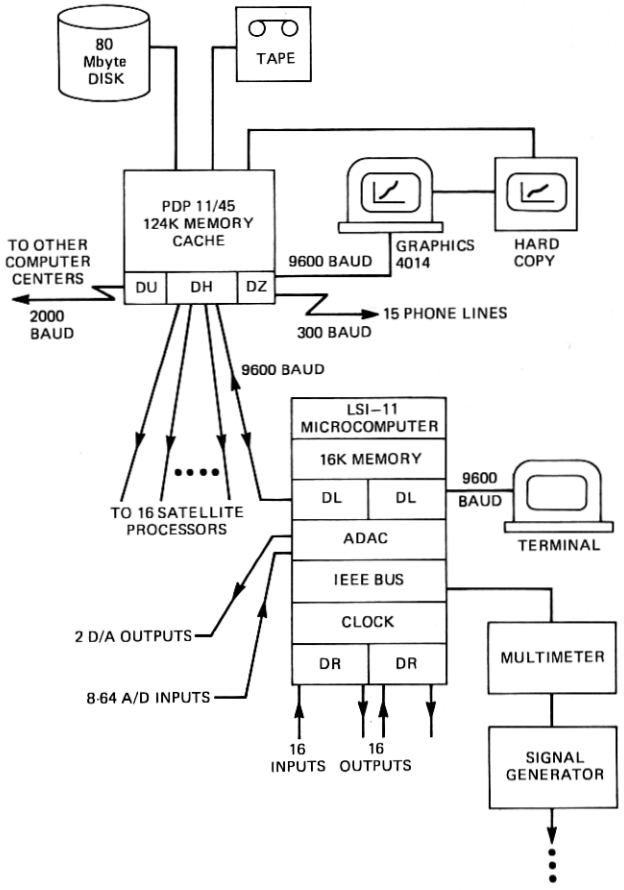

Fig. 1—Satellite processor system used for control of experiments.

mathematical subroutines, for example, the PORT library,[10] or because of inertia on the part of experimenters.

2.2 The satellite processor

Eight SPs are presently connected to this central facility, each consisting typically of an LSI-11 microcomputer with extended arithmetic, a 9600-baud CRT terminal, two serial line interfaces (one for the CP and one for the terminal), a PROM/ROM board containing a communications package, and from 8 to 28K of semiconductor

memory. Because memory is inexpensive, most users choose more than minimum storage.

2.3 The CP-SP interface

The SPs are linked to the CP by means of serial line interfaces (see DL and DH in Fig. 1), operating over two twisted pairs of wires, often 300m long.

The SP does not have the resources either in memory or secondary storage to run the UNIX system directly. For a UNIX system that will run on a microprocessor equipped with a floppy disk, see Ref. 11. However, with the cooperation of the CP, the SP can emulate the UNIX environment during execution of a program, using the Satellite Processing System (SPS).[12]

SPS prepares a program for execution in the SP, transmits it, and monitors it during execution. If the program in the SP executes a system call, it is sent to the CP for execution. In our experience, sending all system calls to the CP proved burdensome, so we modified SPS so that the SP would handle system calls itself if possible, referring only those system calls to the CP which the CP alone could handle. For example, reading and writing the local terminal is best handled by the SP itself, whereas reading or writing a remote file can only be done through the CP. Certain other system calls, **fork**, for example, are simply not appropriate in the present distributed processing framework and are presently treated as errors. When no program is executing in the SP, the local terminal behaves exactly as if it were connected directly to the CP under UNIX timesharing. Further revisions to the CP-SP communication scheme are under way that should permit the SP to run a long-term program acquiring data while the local terminal is used for ordinary timesharing.

Although SPS was designed to accommodate a variety of computers, cost considerations have led us to use LSI-11 microcomputers exclusively. If future needs dictate a large computer, the capacity is there, although the future may well bring bigger and faster satellite machines.

2.4 Interface to the experiment

A surprising variety of experiments can be automated with the interfaces shown schematically in Fig. 1.

(*i*) The CRT terminal operating at 9600 baud provides interaction with the operator at a speed high enough to permit messages as explicit as needed without slowing the experiment down.

(*ii*) Voltage signals are read or generated by means of an ADAC (Analog-to-Digitial Digital-to-Analog Converter). Connections are made through a standard multipin connector. From 8 to 64 lines can be used on input and two lines on output. The usual range of voltage is 10 volts with a precision of 5 mV. The programmable gain option allows the precision to be increased to 0.5 mV over a correspondingly smaller voltage range.

(*iii*) Signals that are binary in nature can be interfaced with the LSI DRV-11 parallel interface, which provides 16 input and 16 output lines. On output, pulses can be generated to step motors, set relays, or trigger devices that respond to TTL signals. On input, switch closure, TTL signals, and interrupts can be monitored. The DR also provides a way to interface to numeric displays or inputs. If the number of binary inputs is large, several DRs can be employed.

(*iv*) Sophisticated instruments, such as multimeters, frequency synthesizers, transient recorders, and the like, can be interfaced through the IEEE instrument bus.[1] More than a hundred instruments are available with the IEEE bus, and the numbers have been increasing rapidly.

(*v*) Timing during the experiment can be accomplished with the internal 60-Hz line time clock of the LSI-11 or by a more precise programmable clock, the KW-11.

III. INTERFACE TO THE EXPERIMENTER

Our goal was to create an interface between the user and the experiment which used the machine to do the dirty work, met a variety of needs, and was so easy to use that people wouldn't try to reinvent it—in short, to develop what Kernighan and Plauger describe as tools, as opposed to special-purpose programs.[4]

Each interface described above is handled with one or more tools summarized in Table I. Each is a function or series of functions written in the C programming language[13,14] which can be called from a C program. The C functions can also be called directly from programs compiled with the FORTRAN 77 compiler which is now operational on UNIX.[15] A tutorial discussion of the use of these

Table I—Tools for experimental control

```
Interaction with the experimenter
      rsvp(); number();
Data acquisition
      getvolt(); setgain(); zero(); parin(); parint();
Experimental control
      outchan(); putvolt(); ramp(); sine(); parout();
Timing and synchronization
      time(); delay(); setsync(); sync();
Dynamic data storage
      bput(); bget(); bfree();
Sending data to the central computer
      bsend(); bname();
```

tools is available[16] which has served as the text for a 16-hour course in computer automation.

3.1 Interaction with the terminal

The programmer can use two tools to ask a question of the operator and read the response: **number** which returns a floating point number and **rsvp** which matches the response to a list of expected replies. For example,

```
velocity = number("ram velocity", "mm/sec");
```

will print the message

```
ram velocity, mm/sec?
```

on the terminal, analyze the response, and return a floating-point number. If the input is unrecognizable, **number** repeats the message. If the reply is "1 ft/min," a conversion will be made to mm/sec, the units specified by the optional second argument. If the units are unknown, e.g., furlongs/fortnight, an error message will be printed and **number** will try again. It is possible for the user to supply an alternate table of conversion factors for **number**.

A second tool is provided to ask the experimenter a question and analyze the reply. The simplest use of **rsvp** is to use the terminal to get a cue, as in:

```
rsvp("Hit RETURN to start the test.");
```

which prints the message on the terminal and waits for the carriage return to be typed. **rsvp** will analyze responses, as in:

```
reply = rsvp("stress or strain?", "stress", "strain");
```

which prints the first argument on the terminal as a prompt,

analyzes the response, and sets reply to 1 if **stress** was typed, 2 if **strain** was typed, and 0 if anything else was typed.

Because **number** and **rsvp** are reasonably intelligent and cautious about accepting illegal input, they can be used to create other programs which are similarly gifted. For example, many experiments require a knowledge of the area of the specimen under study. A simple function can be written in a dozen or so lines which will calculate the area from information supplied at the terminal for either a circular or a rectangular specimen. The user can supply dimensions in units ranging from yards to microns.[16]

Because **rsvp** and **number** use the standard I/O routines, they run on the 11/45 as well as the LSI-11. Developing them was also easy, because the hard part was done by UNIX library routines like **atof**.

3.2 Interfacing with analog signals

Once the leads from the experiment are connected to the ADAC, the voltage on the **n**th channel can be read by calling

```
getvolt(n);
```

The gain on channel **n** can be set to 10X by calling

```
setgain(n, 10);
```

If this call is issued and the ADAC lacks programmable gain, or if the desired gain is not available, **setgain** will print an error message. In some experiments, it is convenient to take an arbitrary reference voltage as a zero reference. Calling

```
zero(n);
```

will take the current voltage reading on channel **n** as the zero reference for all subsequent readings on channel **n**. All data returned by **getvolt** will be in consistent internal units, so that the gain or zero can be set independently by different routines.

Voltages can be generated with the function:

```
putvolt(v);
```

If there is more than one D→A, the function **outchan** can be called to specify the channel for output. The following code causes zero volts to appear on the two output channels.

```
outchan(0);
putvolt(0);
outchan(1);
putvolt(0);
```

putvolt can be used in combination with the timing tools to create a variety of signal generators or command signals. If the response to the voltage command is measured with **getvolt** and fed back to the command, closed loop control is possible. Examples of such control are given in a later section.

3.3 Parallel interfacing

The 16 output lines of the parallel interface can be written simultaneously by writing a 16-bit word with the call:

```
parout(n, word);
```

where **n** refers to the number of the DR interface board. The binary representation of the word will determine which lines are on or off. Since the C language provides a number of operators for bit manipulation, the common operations of setting, clearing, or inverting the ith bit can be accomplished easily.[14]

Similarly, the state of the 16 input lines on the *n*th DR can be read simultaneously with the function:

```
parin(n);
```

which returns a 16-bit integer. The C bit operators are then used to mask off the bits which correspond to the signals of interest.

The interrupt inputs on the DR can be used to signal the processor as described in a later section.

3.4 The IEEE instrument bus

The IEEE bus interface tools are still under development with the goal of writing high-level commands that will interface to many electronic instruments. At present, the instruments can be controlled by the standard technique of writing or reading an ASCII stream on the bus with reads and writes or the high level routines **scanf** or **printf**. A command analogous to **stty** called **buscmd()** sets the state of the bus driver.

3.5 Timing

Events during the experiment can be timed by means of the internal line time clock in the LSI or by a higher precision programmable clock. In either case, the tools for timing remain the same; the precision simply changes. The simplest use of the clock is to measure elapsed time with the variable **time** which is incremented each clock tick. Events can be synchronized with a clock by the function **sync()**. A function **setsync(t)** is provided which sets the period of the sync signal to *t* ticks of the clock. Subsequent calls to **sync()** will not return until a sync signal is generated. In this way, data can be obtained at regular intervals, or pulses of a specified frequency can be generated. A third function **delay(n)** causes the program to wait for **n** sync signals. The accuracy of the timing functions is determined by the accuracy of the clock and ultimately by the speed of the LSI-11, which seems to limit practical timing to frequencies of less than about 10 kHz. In the future, faster processors may extend this limit.

3.6 Data storage and transmission

Many applications of computers to experiments are primarily data acquisition and recording. The tools for storing data on the SP and transmitting it to remote files on the CP are important. Therefore, considerable effort has been spent on devising a set of commands for storing data in buffers on the SP and transmitting the buffers to named files on the CP.

A group of buffers are provided in which data can be stored without regard to type (**int**, **long**, **float**, or **double**). The buffering routines keep track of the type automatically. This is useful in storing data for an xy graph; *n* pairs of integers which form the data for the graph could be stored in the same buffer as an initial pair of double precision scaling factors which convert the integer data into useful units. The entire buffer can be transmitted to the CP as a unit. The buffers are dynamically allocated; that is, depending on how the data are written, one buffer could consume all the buffer space, or all the buffers could share the space. The command **bfree(n)** releases the **n**th buffer whose space is then made available for data storage by any of the other buffers. The command **bsend(n)** sends the contents of buffer **n** to the CP. The function **bname()** is used to specify the name of the file.

The buffering commands can be used in combination to form a sophisticated data logging system. For example, **bsend** can be called periodically during data logging to update a file on the CP. The transmission will be incremental; that is, only the data added to the buffer since the last **bsend** will be transmitted. For applications where more data are taken than will fit in the SP, large files can be built on the CP by doing a **bsend** when the buffer fills, followed by a **bfree**. Once the files are saved on the CP, the UNIX shell and programs can be used to manipulate or forward the data to yet another computer facility for analysis on a higher speed computer.

3.7 Interrupt and signal handling

In the control of real-time experiments, it is sometimes necessary for the experiment to interrupt the computer, as for example when data collection is complete or an event of some urgency has occurred. The UNIX system's signal and interrupt mechanisms are rudimentary, as such events are rare in the fully buffered I/O environment. The single-level interrupt structure of the LSI-11 further complicates the problem of handling interrupts generated by the experiment. Our solution has been to implement a simple control primitive for doing all this, the EXECUTE command. EXECUTE(**routine, priority**) allows the user to specify a routine to be run when the software priority level of the program drops below the given level. Because things normally considered atomic in nature (floating-point arithmetic and system calls) may be extremely slow when simulated on the LSI-11, it is advantageous to make them interruptable at a high priority rather than atomic. To guarantee no side effects, the interrupting routines must take care not to execute non-reentrant code.

The user would not normally use EXECUTE directly but would use **parint(n, routine, priority)** which handles interrupts coming in on the *n*th DR interface. **parint()** can be used to turn off all interrupts or name a routine to be EXECUTEd at the specified priority when an interrupt occurs. For example, in the case of finding peaks in a diffraction spectrum, it is advantageous to compute while data are being collected by the counters, and be interrupted when the count is completed. Such an interrupt could be executed at a relatively low priority. If, on the other hand, an interrupt is received which indicates that a disaster has occurred, it should be handled at the highest priority.

3.8 Notes on implementation

We have tried to design a software interface to experiments which follows the UNIX unified approach to input-output (I/O), that is, I/O should take place without regard to the type of device being read or written. For example, the local terminal, remote files, or the IEEE bus can all be read or written with standard I/O.

Using standard I/O provides additional benefits. Routines for the IEEE bus were developed on the 11/45 and run on the LSI-11 without change. Development was much easier on the 11/45 where pipes and a bus simulator were used to test the program before running on the LSI-11, without the added set of problems caused by running on the unprotected LSI-11. The IEEE bus interface could easily be used as a model for a UNIX device driver if the need arises.

I/O is handled in the modified SPS framework by keeping a list of device descriptors that represent local devices. Any system call for a local device is handled without CP intervention; all others are passed to the CP. The local terminal driver includes fully buffered input and output, echoing, erase and kill processing, and interrupt and quit handling.

A different approach is necessary for the devices on which a single word or byte represents the entire transmission (DR or the ADAC). These devices can be faster than the LSI-11 itself, so we have kept the interface as low level as possible, thereby incurring less overhead than would be the case if the standard I/O system were used. We have tried to make the I/O as independent of the device as possible, so that the user will not need to understand the detailed operation of each device and so that similar devices can be interchanged without changing the user programs.

Each SP has a unique combination of hardware which requires a unique library of software tools to make it work. We are able to compile such a library by specifying options to the C preprocessor at compile time, so that the libraries can be prepared without changing the programs or including redundant information.

IV. EXAMPLES

The measure of the system and the tools is to be found in their application to actual experiments. The following examples are experiments which are now running, using the system and tools described above.

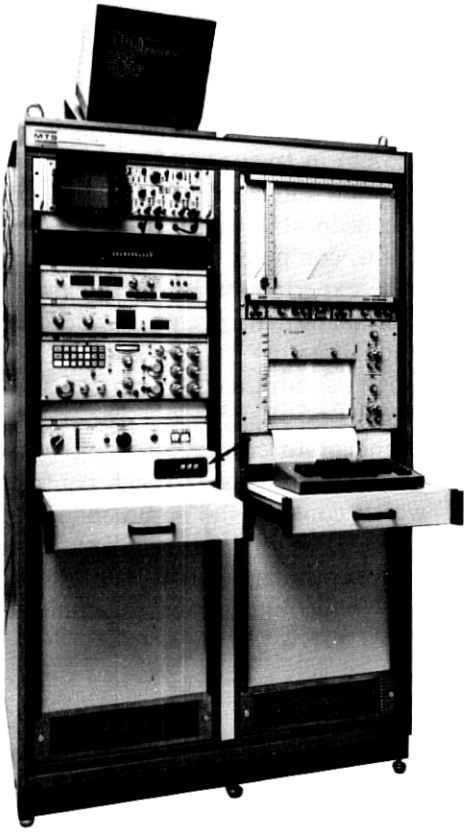

Fig. 2—Servo-hydraulic control panel. The satellite processor is the small module indicated by the arrow.

4.1 Mechanical testing

One function of computer automation is to simplify the operation of a machine and shield the inexperienced user from its intricacy. The complexity of the control module of a modern servo-hydraulic machine is apparent in Fig. 2. The microcomputer, which controls the machine, is the small module indicated by the arrow. As a simple example of the complexity of running an experiment without computer assistance, the units of the stress strain curve being displayed on the *xy* recorder in Fig. 2 are determined by the settings of four potentiometers, the sensitivity of two transducers, and the dimensions of the sample. Under computer control, the results are

presented directly in the appropriate units. The computer program also provides complete step-by-step instructions for the operator during the test.

A second function of computer automation is to provide safe control of a machine capable of generating forces of 50,000 kg and displacing a piston 25 cm in fractions of a second. Largely because of the simplified operation and safety of computer control, inexperienced operators have been able to use this machine with minimal instruction. In addition, computer control permits very precise data acquisition, control of experiments not possible by hand, detailed data analysis, and automatic data archival which forms the beginnings of an automated laboratory notebook.

Some details follow of how an experiment was automated to determine the beginnings of plastic flow. The machine can be controlled by selecting one of three servo modes: stroke, which controls the position of the piston; load, which controls the load generated; or strain, which controls the strain induced in a specimen. The servo mode (load, stroke, or strain) is set by calling **outchan(STROKE)**. The command signal is then generated by a call to **putvolt** or **ramp()** which generates a ramping voltage to the desired level.

```
outchan (LOAD);
zero(LOAD);
ramp((int)100.0*KG);
```

sets the servo mode to load control, takes the current load reading as the reference zero, and increases the load to 100 kg.

Figure 3 illustrates the results of a test to detect the elastic limit of a material. The specimen is deformed under strain control to *A*, then unloaded under load control to *A'*. The sample is cycled to progressively larger strains *(B, C, D, ...)* until a given amount of residual strain on unloading is reached, *G'*. Data are taken during the test with **getvolt** and stored with **bput**, as in:

```
bput(YS, getvolt(STRAIN));
bput(YS, getvolt(LOAD));
```

which stores a stress-strain pair in the buffer, *YS*. Strain control is necessary because the shape of the curve would not permit equal load increments.

Once this part of the test is completed, the results are analyzed and the yield stresses displayed for the operator. Testing then resumes under stroke control until failure of the specimen is

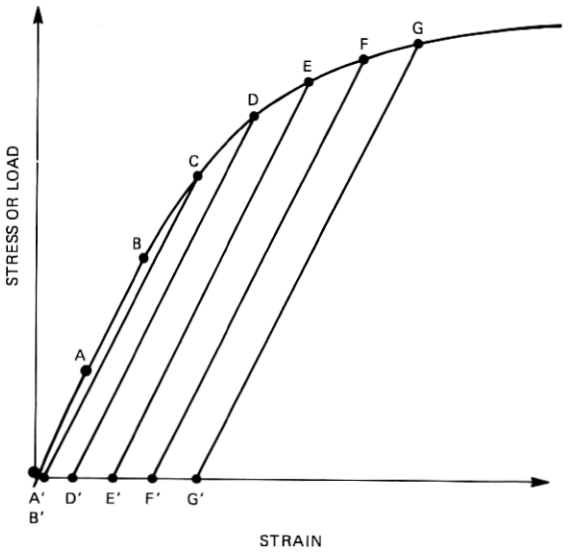

Fig. 3—Load-unload test. Testing begins at the origin and proceeds along the path given by AA′ BB′ CC′ DD′ EE′ FF′ GG′.

detected by a load drop. The uniform and total elongation and the tensile strength are then calculated and displayed. At the conclusion of each test, the results and stress-strain curves are transmitted by **bsend** to the CP to files bearing the name of the specimen. Finally the buffer space is freed using **bfree** and the program requests information for the next sample.

Thanks to the UNIX file system, the files are marked with the exact date and time of their creation, so that the file system itself serves as an automated laboratory notebook. Commonly, after a group of specimens is tested, the experimenter uses a graphics terminal under time-sharing to examine the data with the **graph|tek** command. The combination of the UNIX file system, the shell, and the graphics makes cross comparison among specimens easy. When the analysis is finished, the UNIX archive command **ar** can be used to store the data in a compact form.

Other programs have been written that accomplish static and load-unload tensile testing as well as fatigue and stress-relaxation testing. Benefits go beyond automation of existing tests, for the load-unload program outlined above would be impractical under hand control on this machine.

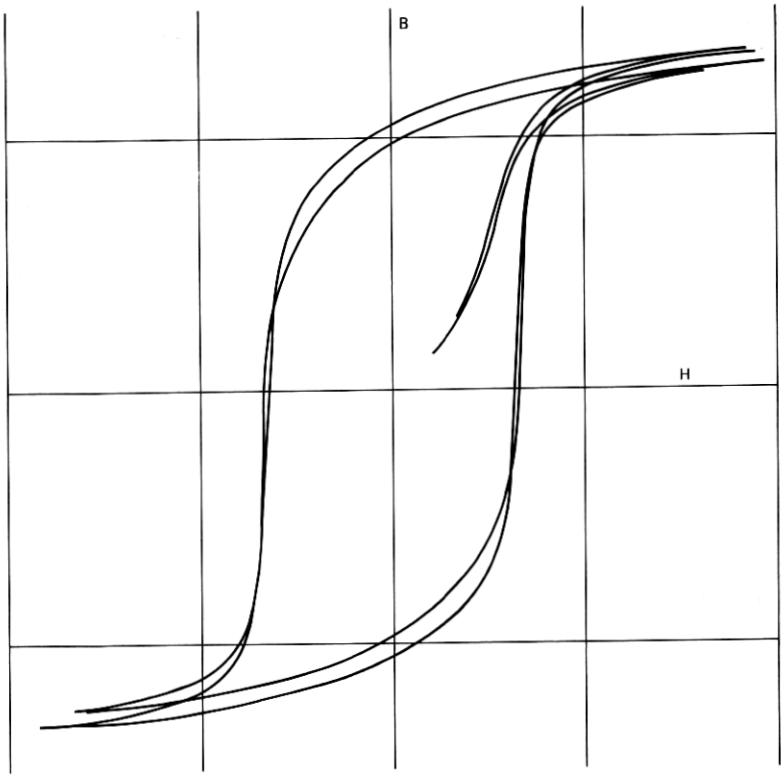

Fig. 4—Magnetic hysteresis loops. *BH* loops are shown for two materials given slightly different heat treatments. Note the difference in the II and IV quadrants, which show up in a lower energy product, *BH*.

4.2 Magnetic testing

Using the tools developed for the preceding example, we automated a magnetic hysteresis graph in about 5 man-hours from start of typing and soldering to the first analyzed data. The application monitors the magnetic field intensity H and the magnetic induction B of a sample to determine its hysteresis loop (*BH* loop). The output is a *BH* loop (see Fig. 4) which can be graphed on the CP, as well as a table of the cardinal points obtained from the loop: coercive force, remnant magnetization, saturation magnetization, etc. The program also provides the energy product, *BH*, at specific load lines and the maximum energy product.

Even though the application is little more than data acquisition and analysis, much time is saved during testing. The ability to

rapidly retrieve data and superpose *BH* loops of various materials, illustrated in Fig. 4, has been a valuable byproduct of automation. Future plans are to use the computer to control the field during testing, which would speed up the test and make possible the collection of recoil permeability data at various points in the loop.

4.3 Other mechanical tests

A single satellite processor has been used to control both the above experiments as well as acquire data from two screw-driven, mechanical test machines located in the same room and automate a rather specialized device called the *WT*-bend tester.[17] The bend tester determines the elastic modulus and the onset of plastic deformation by vibrating a small strip sample in bending. The test is normally controlled by dedicated digital hardware and mechanical feedback. Using the LSI-11 we were able to control the bend tester and acquire data more extensive and precise than had ever been obtained before. The operation was a severe test of the timing abilities of the LSI-11, because a precision of 100 microseconds was required, even though the natural frequency of the vibration was a few hertz. In this application, the normal timing tools were inadequate but served as the starting point for a routine that exploits the LSI-11 very close to its limits. This is possible because the C language permits programming at a level very close to the machine level, thereby taking full advantage of the hardware. If, on the other hand, the tools were written in Basic or Fortran language, they would not have lent themselves to a natural extension to the limits of the machine.

4.4 Low temperature fluid flow

Finally, to demonstrate the versatility of tools developed for a mechanical testing machine, we cite the work of Behringer and Ahlers who have studied the instabilities of fluid flow in liquid He at 2.2K.[18] The first use of their satellite processor was to acquire data over long periods of time ranging from an hour to several days. The ability to run continuously for weeks at a stretch is a measure of UNIX reliability.

More recently, Ahlers has been using the LSI-11 to control his experiments as well as take data. putvolt() is used to generate a sinusoidal power input at one plate of the convection cell, while at

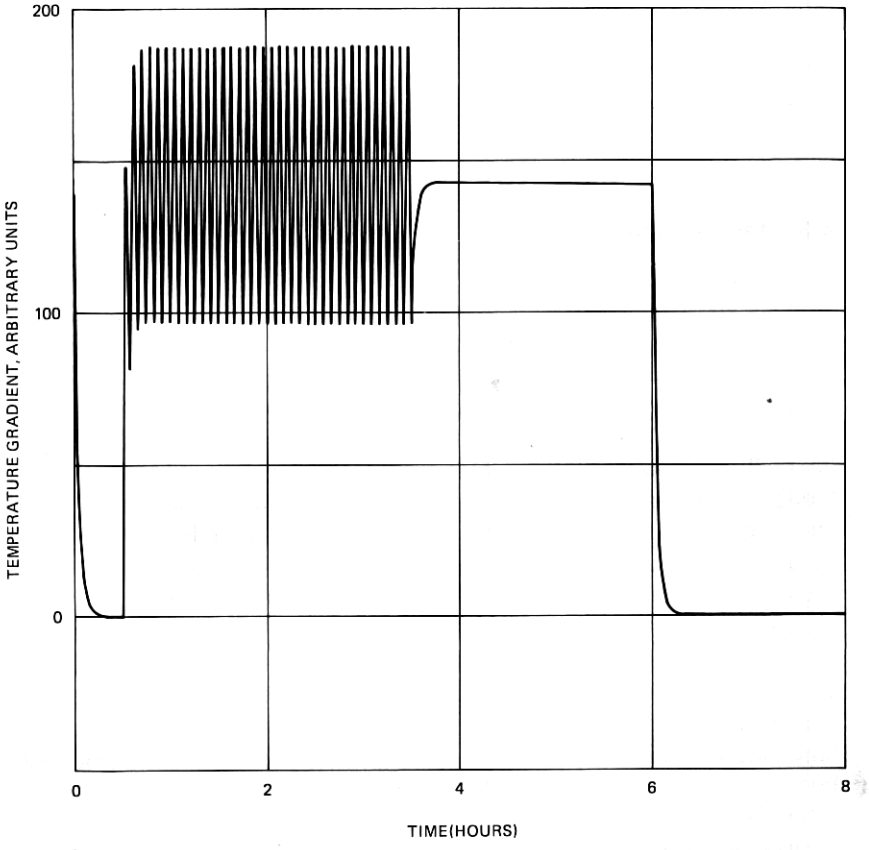

Fig. 5—Temperature gradient across a cell as a function of time. The gradient is produced by applying a power input to one plate of a cell filled with liquid He at 2.2 K. The power is varied sinusoidally at first, then is held constant at the rms power level of the sine, and then is shut off. See Fig. 6 for a magnified view of how the mean temperature approaches steady state and Fig. 7 for the power spectrum of the response.

the same time the response (the temperature gradient across the cell) is recorded with **getvolt** and stored with **bput**. The resulting gradient is displayed in Fig. 5. After a time the sinusoidal input is replaced by a constant power input of the same mean power, and the gradient is again monitored. The power is then shut off for an interval and the process is repeated with a slightly higher power density.

Figure 6 shows how the mean temperature gradient approaches steady state for the sinusoidal input and the constant input. It

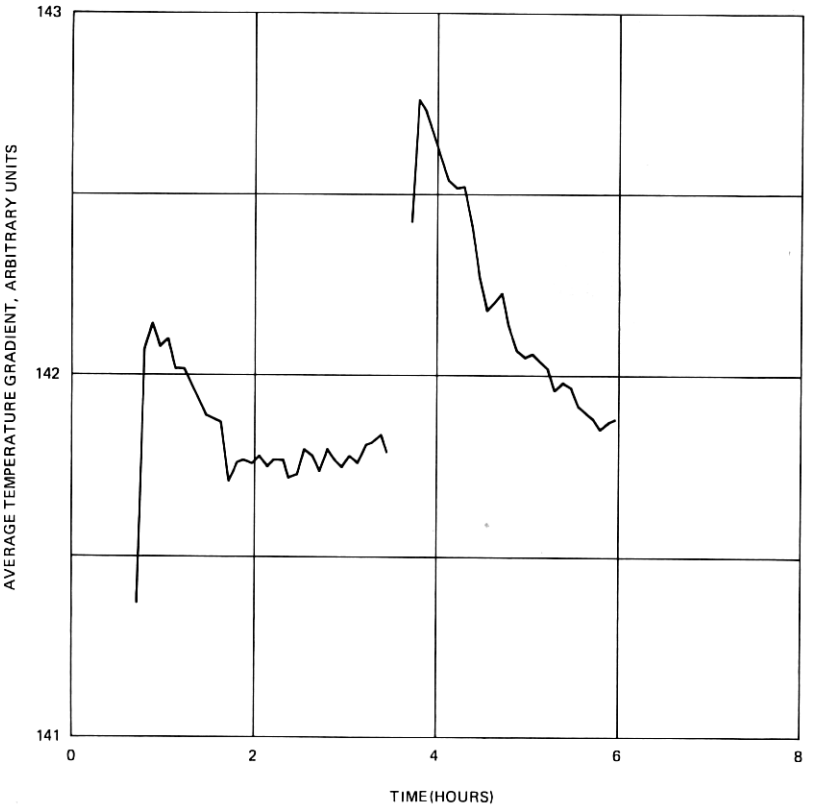

Fig. 6—Mean temperature as a function of time. The data from Fig. 5 have been averaged and magnified to show that the approach to steady-state conditions depends on whether the input is oscillating or constant.

shows that the stability of the system depends on whether the input is oscillating or steady. Still using data obtained with the LSI-11, Ahlers applies fast Fourier transforms to reveal the details of the instability (Fig. 7). The ability to write programs, control experiments, analyze data with sophisticated mathematical library functions, display it graphically, and write it up for publication on a single system is a significant advantage.

Other applications under way or in progress include x-ray diffraction, pole figure determination, scanning calorimetry, phonon echo spectroscopy, thin-film reliability studies, and semiconductor capacitance spectroscopy.

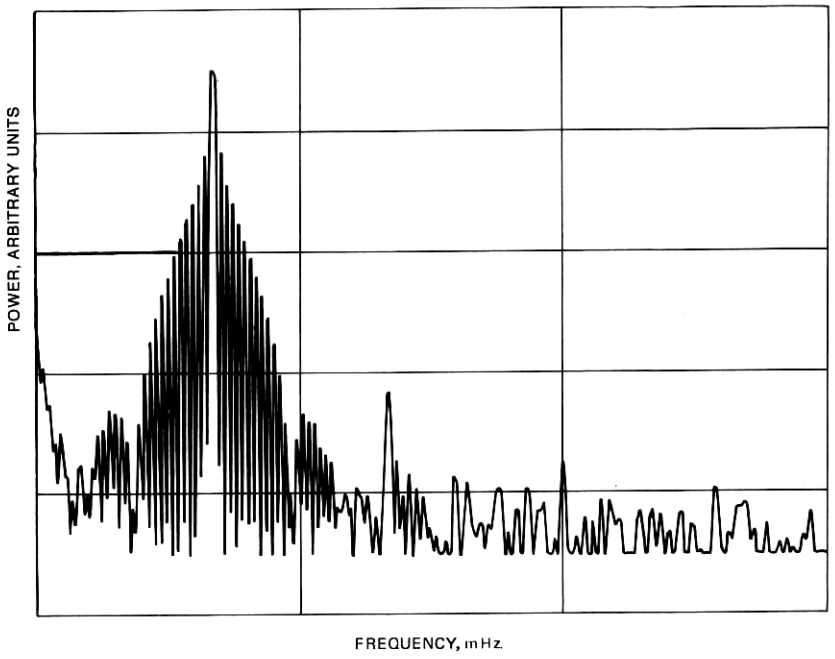

Fig. 7—Power spectrum of the temperature gradient vs. time function. The response to the sinusoidal power input is Fourier-transformed, and the corresponding power spectrum is displayed. Note the peaks at the second and third harmonic, which were not present in the power input.

V. CONCLUDING REMARKS

We have in operation a system for controlling laboratory experiments which is powerful, easy to use, and reasonably general. It combines the isolation and real-time response of a stand-alone system with the shared cost and better hardware/software facilites of a large time-shared system. It is powerful because the UNIX operating system and the C language provide facilities for file manipulation and the direct control of devices. It is easy to use because tools have been written which shield the novice from many of the interfacing details. It is general because the tools were written with general, rather than specific, applications in mind. Where great speed or specialization is necessary, the tools form a model that can easily be modified to meet the needs.

For the future, we expect bigger and faster satellite microprocessors which will add further to the attractiveness of the satellite processing scheme. As microprocessors become incorporated in test

equipment, we see a trend toward more intelligent instruments which, on command from the SP, can execute fast, complicated procedures without the SP's intervention. The test instrument should be intelligent about performing its essential functions and should provide an interface (eg. the IEEE bus) to a more general-purpose machine which controls other tests, coordinates instruments, and analyzes the results. Trends toward building full-blown software systems including file systems into large test equipment seem counterproductive.

As to the future of the software discussed here, we plan to revise the CP-SP communications interface to take advantage of new UNIX features to improve reliability, versatility, and speed. Further work remains to be done on a set of higher level tools for interfacing with instruments on the IEEE bus.

VI. ACKNOWLEDGMENTS

The authors wish to thank R. A. Laudise, T. D. Schlabach, and J. H. Wernick for support and encouragement during this project; H. Lycklama, C. Christensen, and D. L. Bayer for discussions and aid; and P. D. Lazay, P. L. Key, and G. Ahlers for numerous contributions and a critical reading of the manuscript.

REFERENCES

1. "Standard Digital Interface for Programmable Instrumentation and Related System Components," IEEE Std. 488-1975, IEEE, New York. See also D. C. Loughry, "Digital Bus Simplifies Instrument System Communication," EDN Magazine (Sept. 1972).
2. CAMAC (Computer Automated Measurement and Control) Standard, Washington, D.C.: U. S. Government Printing Office, documents TID-25875, TID-25876, TID-25877, and TID-26488. See also John Bond, "Computer-Controlled Instruments Challange Designer's Ingenuity," EDN Magazine (March 1974).
3. S. C. Johnson and M. E. Lesk, "UNIX Time-Sharing System: Language Development Tools," B.S.T.J., this issue, pp. 2155-2175.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Reading, Mass.: Addison-Wesley, 1976.
5. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., this issue, pp. 2049-2086.
6. J. V. V. Kasper and L. H. Levine, "Alternatives for Laboratory Automation," Research/Development Magazine, *28* (March 1977), p. 40.
7. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
8. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
9. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran," Software — Practice and Experience, *5* (1975), pp. 395-406.
10. P. A. Fox, A. D. Hall, and N. L. Schryer, "The PORT Mathematical Subroutine Library," ACM Trans. Math. Soft. (1978), to appear.

11. H. Lycklama, "UNIX Time-Sharing System: UNIX on a Microprocessor," B.S.T.J., this issue, pp. 2087-2101.
12. H. Lycklama and C. Christensen, "UNIX Time-Sharing System: A Minicomputer Satellite Processor System," B.S.T.J., this issue, pp. 2103-2113.
13. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, N.J.: Prentice-Hall, 1978.
15. The UNIX Fortran 77 compiler was written by S. I. Feldman (1978).
16. B. C. Wonsiewicz, J. D. Sieber, and A. R. Storm, unpublished work (1977).
17. G. F. Weissmann, B. C. Wonsiewicz, and Characterization of the Mechanical Properties of Spring Materials, Trans. of the ASME, J. of Science and Industry (1974), p. 839.
18. R. P. Behringer and Guenter Ahlers, "Heat Transport and Critical Slowing Down near the Rayleigh-Benard Instability in Cylindrical Containers," Phys. Lett., *62A* (1977), p. 329.