A satellite system for controlling real-time experiments

M. A. DERR, C. E. WRIGHT, and A. S. CORIELL

Bell Telephone Laboratories, Murray Hill, New Jersey 07974

In this paper, we describe a computer system for controlling real-time psychology experiments. We begin by considering the kinds of experiments that are performed in our laboratory and examining the hardware and software requirements of these experiments. We then review various systems along a continuum ranging from general-purpose timesharing systems to stand-alone dedicated processors. The capabilities of these systems are described in comparison with our requirements. The features of the PARASITE-FS system are then described. This host-satellite system includes a local file system and monitor program, real-time device drivers with flexible interrupt processing capabilities, user-level routines for controlling the real-time devices, and a package for controlling the timing of experimental events.

Psychologists in our laboratory and in other parts of Bell Laboratories conduct a variety of research in human information processing. In most of this research, computers are used as primary tools for data collection and analysis. In this paper, we describe how we developed a system that meets our particular computing needs.

SYSTEM DESIGN CONSIDERATIONS

In designing our system for controlling real-time experiments, we considered not only the kinds of experiments currently performed, but also the demands that future experiments might place on either the hardware or the software of a system.

Data Rate

As in most computer applications, the data-rate demands that our experiments place on the available hardware and software range from minimal to excessive. On the one hand, there are experiments for which timing to fractions of 1 sec or even 1 sec is adequate, and stimuli can be presented and data collected by a general-purpose timesharing computer using a standard terminal. An example is a text-editing experiment in which every character a subject types on the keyboard is recorded and the time measure of interest, the time to compose and enter commands, need only be measured in seconds.

For a more typical application, the stimuli and responses are still simple, but their timing is important and must be made with millisecond accuracy. An example of this is a word recognition experiment in which a character string, followed by a mask, is presented on a visual display scope for durations measured in milli-

seconds. The subject's time to press a response button is also measured in milliseconds. Here, an appropriate configuration might be a small microprocessor, with simple software, running either as a stand-alone system with a small disk or as a satellite of some larger host computer. If multiple subjects are run simultaneously, the system software needs to be more complex, but the basic hardware is still adequate. It is tempting, however, considering the relative costs of the hardware for such a system and software development, simply to replicate the hardware for each subject, rather than complicate the software.

On the other end of the spectrum, we now run experiments that 10 years ago would not have been considered feasible. These experiments use complex visual, auditory, and tactile stimuli presented with special hardware requiring control-signal rates up to tens of kilobits per second. The timing of these stimuli and the subjects' responses must usually be made with millisecond accuracy. The responses we collect have evolved from simple buttonpresses or keypresses to movement trajectories and spoken utterances that require data rates over 100 kbits/sec. Not only must a data sample be stored for later fine-grained analysis, but, often, it is used as the basis of real-time decisions with immediate effects on the progress of a trial.

As an example, consider a reading experiment in which the contents of a visual display are changed during a trial as the subject's eyes move. Eye movement information is obtained through multiple channels of an analog-to-digital (A/D) converter from an eye-tracking device. The program that controls the experiment stores this information for later analysis and also uses it in real-time to determine where the eyes are pointed and when a saccade is being made.

To be controlled properly, these experiments require a dedicated high-performance microprocessor or mini-computer, even if only for an instant during each trial. In addition, storage of complex stimuli or data require at least a medium-sized disk drive.

Mel Melchner provided invaluable help during the initial stages of this work. Steve Hartwell transported the portable C compiler, loader, and other utilities to the VAX. Walt Kropf designed the hardware interfaces to the visual display processors.

Devices

Because of the range of experiments performed in our laboratory, a system must be able to support many different kinds of peripheral devices for presenting stimuli and collecting and storing responses. Some of these devices and their uses are summarized below: (1) Terminals and other serial-line devices that operate at rates up to 9,600 baud. (2) Raster-vector display processors. (3) Programmable clocks for general timing and for timing the refresh rate for the visual display. (4) Digital input interface for collecting responses (e.g., a subject's keyboard, response keys, touch-sensitive response panels, or energy-sensitive hardware voice keys). (5) Digital output interface for controlling devices (e.g., programmable slide projector, signal lights, electronic shutters, audio-tape deck, or relays). (6) Other input or output devices, such as a line printer or a digital mini-tape drive. (7) A/D converters for collecting speech data and input from other analog devices. (8) Digital-to-analog (D/A) converters for reproducing speech, generating tones, and controlling other analog devices.

Hardware and software interfaces are needed for accessing and controlling each device.

Interrupt Processing

Many of the peripheral devices operate under interrupt control. That is, when a device has finished processing data or is ready to receive more data, the device sends a hardware interrupt signal that causes process control to jump to a special interrupt handling routine for processing the data. This avoids having to poll the device to see whether it is ready. The use of interrupts is essential in applications for which multiple operations must be performed quickly with strict timing constraints.

Interrupt processing must be fast and flexible. For many devices, a common procedure is simply to store the input value when an interrupt occurs. In applications for which values are obtained continuously (such as collecting A/D speech samples), the interrupt routine can buffer sample values automatically. For devices that collect input at irregular rates (such as a subject's keyboard), the interrupt routine can store the input value along with the time that it was received. This is faster than the more common procedure of passing a single value to a user routine that must explicitly store the input value and, if time is to be recorded, read the clock and store the time.

We also want to provide options for user control of interrupt processing, that is, to allow users to customize interrupt processing without changing the interrupt software. One possibility is to have the system send a software signal when an event is encountered during the processing of an interrupt. Examples of these events are the receipt of a character or a new line from a terminal, the occurrence of a specific buttonpress when the A/D or D/A converter has finished a conversion from one or several channels, and the occurrence of an error. A user-written subroutine invoked through this linkage

"catches" the signal and does whatever processing is appropriate. This mechanism is appropriate for non-standard processing that can be performed quickly for devices with low data rates. A recent example in our laboratory for which a signal routine seems appropriate is a simple algorithm that, based on A/D sample values obtained at 500 Hz from an eye-tracking device, determines whether the eyes are fixating or making a saccade. This decision must be made on-line, since the contents of a visual display are changed during saccades.

User Interface

One of our goals was to design a good user interface for controlling real-time devices. The interface must be flexible; it must be capable of controlling all device operations. At the same time, the interface must be consistent across devices. These requirements conflict, since a natural structure for specifying operations of one device rarely will fit other devices; consider, for example, a graphics display vs. a programmable, real-time clock. Finally, we want an interface that is understandable and easy to use, since most of our programming is done by experimenters who have little or no background in systems programming.

Another design decision was to use a general-purpose high-level programming language in our system, rather than develop a new language designed especially for running experiments or use an assembly language. One's approach to this decision clearly depends on the languages available. The availability of the C programming language (Ritchie, Johnson, Lesk, & Kernighan, 1978) was important. Given our previous experience with FORTRAN, the decision might have been different if this language had been the only option. It was our experience using FORTRAN that it was repeatedly necessary to write assembly language subroutines to get around the limitations of that language in our applications.

Timing of Stimulus and Response Events

An issue in designing a system for controlling real-time experiments is the ease of performing complicated timing operations. In the context of our old system, we found that the code to schedule a sequence of stimuli, while simultaneously monitoring for the occurrence of responses, can be complicated and error prone. Contingencies often are overlooked when such code is planned. This is especially true when the subject's responses begin a sequence of stimulus events, extended in time, that must fit in the basic fixed sequence of events in the trial. Also, our experiments often involve changing the temporal order of stimulus events. In a program that does its own timing, this might require having a separate sequence of code for each ordering.

Data Storage

The system must be able to store and retrieve large amounts of data at rates over 100 kbits/sec.

Program Development and Data Analysis Tools

Another system requirement is the availability of utilities for supporting program development and data analysis. Both of these activities are perhaps best done in a general-purpose timesharing environment.

CHOOSING A SYSTEM

The range of possible systems can be thought of as a continuum with a general-purpose timesharing facility at one extreme and a stand-alone dedicated micro-processor at the other (Vesonder, Lesgold, & Wolf, 1978). Between the two extremes are various host-satellite configurations.

General-Purpose Timesharing Facilities

A general-purpose timesharing system can provide powerful tools for program development and data analysis. It also has disk drives, tapes, and so on, for storing large amounts of data. A timesharing system can control experiments with minimal timing demands; however, in experiments for which timing of events must occur with millisecond accuracy, a timesharing system is inadequate without special provisions to disable time-sharing or lock processes and set priorities. Also, as Vesonder et al. (1978) point out, even with special provisions performance breaks down on heavily loaded timesharing systems.

Stand-Alone Dedicated Processor

On the other end of the continuum are stand-alone dedicated computers. One arrangement is a single system that is both powerful enough to run the necessary utility programs and flexible enough to handle real-time requirements of several experiments simultaneously (Lycklama & Bayer, 1978). A second possibility is to have several independent, and completely self-sufficient, stand-alone systems. Experience with this second arrangement has shown us two problems: First, it is difficult to keep the systems compatible. When there are many system disks, each maintained by its primary user, users who move between machines or switch disks must constantly move new user software between disks and check libraries for outdated software.

The second problem is that our support software (editors, compilers, etc.), although modern when purchased, stayed the same while that in the rest of the world evolved. This occurred because no other group supported these systems, and although we were able to develop software specific to our requirements, we lacked the staffing to develop general-purpose software. As a result, in the 1980s, the primary language on these old systems is still FORTRAN II.

Host-Satellite Configurations

In host-satellite systems, one host processor provides the necessary general-purpose utilities for program development and data analysis, and several small, dedicated satellite processors do all real-time work. Systems

at various points along the continuum differ in how dependent the satellite is on the host and in what level of service the satellite provides. With a satellite system, there is no problem of compatibility across systems, since the libraries and system software are kept in one place on the host system. The utility software remains as current as that of the host, since it is that of the host. However, it is not necessary to update satellite programs every time the host operating system is changed, since the software of the satellite is distinct from that of the host. The exceptions are the compiler and loader. Since the host and satellites are different machines, a cross-compiler and loader are used in developing software for the satellite processor on the host processor. Care should be taken to choose a strongly supported, portable compiler and related software.

The convenience of the host-satellite arrangement has a price, which takes two forms. The first is the necessity, for certain operations, that two sets of hardware work. The size of this problem depends on the reliability of the host system and how dependent the satellite is on the host. The second penalty of the host-satellite relationship is that programs must be shipped to the satellite, and data must be sent back. This requires a reliable, high-speed transmission line. Since faster lines require more expensive hardware and more commitment from the host system, a practical alternative is to use 9,600-baud serial lines that can be attached to the host to look like a normal terminal

Host-satellite systems differ in the amount of processing that is done on the satellite vs. the host. Moving functions from the host to the satellite (1) increases the independence of the satellite, since the satellite can operate longer without calling on the host, and (2) usually reduces the time required to perform operations by removing the time to communicate with the host from the total time to do the operation but (3) also entails increased hardware and software complexity for the satellite. At one extreme, the satellite microprocessor acts as a flexible, intelligent interface between the host and the outside world, which carries out specific operations but has no control function (Vesonder et al., 1978). At the other extreme, the satellite microprocessor can run an entire experiment without having to call on the host.

Three Host-Satellite Systems

We considered three different host-satellite system designs as potential starting points for developing our system. One type is a communications protocol between a satellite system and its host. It has provisions for using the satellite's terminal as if it were connected to the host, downloading the satellite, starting program execution on the satellite, and transferring blocks of information between the program running on the satellite and a process on the host. However, this system has no standard provisions for interrupt handling, device service, or I/O facilities. (This means that it places no restrictions on how those facilities are later implemented

by the user.) This lack of features makes such a system simple to implement but difficult to use.

Another system farther down the continuum (Lycklama & Christensen, 1978) provides the same communications facilities as above and, at the same time, supports the standard system calls of the host. (System calls are requests to the operating system. Services that an operating system might provide include doing input or output, allocating memory, creating, deleting, or modifying files, providing status information, and, in timesharing systems, creating, killing, or scheduling processes, doing process accounting, and providing communication between processes.) This is done by sending a message to a process running on the host when a system call occurs. The host process does the operation (often simply by making the same system call to the host operating system) and returns the result to the satellite. In this way, references to local devices are also handled through the host. This is a clean interface, providing most of the power of the host operating system to the user of the satellite. However, since so many functions are performed on or handled through the host rather than the satellite, it is too slow for many applications.

The PARASITE system (Murrell & Kowalski, 1980) continues the process of uncoupling the satellite from the host computer. The PARASITE system provides programs written in the C language with an operating system environment similar to that of the UNIX¹ timesharing system (Ritchie & Thompson, 1978). All supported system calls are handled by software on the satellite system. These are implemented as subroutines and macros that reference a set of device drivers and an interrupt handler. The function of one standard device driver is to reference the file system on the host through a communications protocol known as the packet driver. This interface allows programs on the satellite to read and write files on the host. This interface does not, however, provide other standard UNIX system calls to manipulate files on the host that are provided naturally on the second system.

The PARASITE system provides a terminal driver and drivers for several real-time devices (programmable clock, programmable parallel lines, A/D converters, and so on), as well as a general framework to incorporate drivers for new devices. All interaction with devices must be organized to fit within the framework of five standard system calls:

"Open" initializes the named device and returns a descriptor to be used in future references to the device. "Close" closes the device, turns off interrupts, and clears buffers. "Read" reads a word from the device (the terminal and packet drivers read a block of input). "Write" writes a word to the device (the terminal and packet drivers write a block of output). "Ioctl" performs other control operations by reading and writing flags associated with the device.

Unlike the two previous alternatives, the PARASITE system provides a set of user-oriented library functions.

These include utilities for formatting output for terminals and files, decoding formatted input, converting from ASCII to binary, translating characters, classifying characters, allocating memory, generating random numbers, manipulating strings, and other functions.

Because there is no file system on the satellite in the PARASITE system, it is dependent on the host for retrieving and storing data. In psychological experiments, subjects usually must be scheduled 1 or 2 days in advance, and frequently, the spacing of multiple sessions is an important variable. Thus, the reliability of the system that controls experiments is important. With the PARASITE system, an experiment can be run only if both the host and a satellite computer are available. On the other hand, a satellite system with its own file system could be used to run individual sessions, and possibly an entire experiment, without intervention from the host, since programs, stimuli, and data can be stored in the file system.

THE PARASITE-FS SYSTEM

We have described some requirements for a software system to run real-time experiments and reviewed alternative system designs in terms of our requirements. Next, we describe the system we built using the PARASITE system as a starting point. We call this PARASITE-FS, because it is a version of PARASITE with a file system.

Hardware

Because of the number of experiments run simultaneously in our laboratory, we started with three complete sets of satellite hardware. The machine is a Digital Equipment Corporation (DEC) LSI-11/23, a 16-bit high-performance microprocessor that runs on DEC's Q-bus. It is functionally similar to, and almost as fast as, the DEC PDP-11/34, which, however, is a UNIBUS machine. Our systems currently have 128 KB of MOS memory, memory management, and a floating-point processor. Current software does not make use of memory management and, therefore, is limited to accessing 64 KB of memory. We intend to develop ways to use the remaining memory. As a crude measure of the speed of this hardware, consider that adding the contents of one register to the contents of another register requires 1.88 microsec and adding the contents of one memory location to the contents of another, when the memory addresses are stored in the instruction as offsets of the program counter, requires 8.86 microsec.

To satisfy on-line storage requirements, each system has one or two RLO2 disk drives. These drives have removable cartridges, each of which can store up to 10 MB of information. Access is fast; for larger data transfers, the maximum sustainable data rate is 400 KB/sec.

For timing, and to control and collect data from the peripheral devices described earlier, a number of interface boards are required. The equipment that we have added includes the following: (1) A serial-line interface board, capable of driving four terminals, or other serial-

line devices, at separate rates up to 9,600 baud. (2) A raster-vector visual display processor interfaced through a stack-controlled DMA channel. (3) Two 16-bit real-time programmable clocks, one for general timing and one to time the refresh rate for the visual display. (4) Two parallel programmable interface boards. Each provides 32 lines that can be used to control or obtain information from many devices, including digital mini-tape drive, line printer, programmable slide projector, energy-sensitive hardware voice key, subject's keyboard, subject's response keys, touch-sensitive response panel, and so on. (5) One high-current, programmable, parallel output board that provides 32 lines capable of driving a relay. These lines are used to control devices that must be isolated from the computer or require signals that are incompatible with the standard digital signals. Examples of such devices include signal lights, audio-tape deck, electronic shutters, and so on. (6) One A/D converter, capable of sampling 16 input channels with 12-bit accuracy. These channels are used to collect speech data and input from other analog devices. (7) Two D/A converters, each capable of controlling four analog channels with 12-bit accuracy at rates up to 40 kHz. These channels are used to reproduce speech and to control analog devices. (8) A bootstrap-terminator-diagnostic board, whose primary functions are to condition the bus lines and hold the PROM that controls the operation of bootstrapping the system from disk.

The PARASITE-FS File System

Implementation. The approach we adopted for the data storage problem was to build a file system for the satellite. We wanted this file system to be similar to the file system of the UNIX operating system in its appearance to the user. At the same time, the file system needed to be faster and less likely to lose information in a crash than the standard UNIX file system. We modified an existing file system modeled after the UNIX file system, simplifying its structure and making it more efficient.

A schematic representation of the resulting PARASITE-FS file system is shown in Figure 1. The file system can be considered hierarchical, with three levels. The top level of the file system, the root, contains a fixed number of directories, 30. The names of these directories can be changed, but they cannot be created or deleted. Also, files cannot be created at the root level. Each directory can contain up to 18 files.

Files consist of a specifiable but fixed number of contiguous disk blocks. Within the limits of this fixed size, files can be used just as in the UNIX system; the system will not, however, automatically increase their size beyond this limit. The system is capable of increasing the maximum size of a file, if space is available, but this operation may be time-consuming, since the file may have to be copied.

The PARASITE-FS file system puts no restrictions on the content of files. They may be binary or ASCII,

Figure 1. Schematic representation of the PARASITE-FS file system.

and they may contain stimuli, data, programs, or anything else. Files can, however, be given read, write, or execute permission, and these permissions are enforced for all users. Since there is currently no way of identifying different users, it is not possible to establish different permissions for file owners vs. others.

The PARASITE-FS file system includes calls to change the current directory, create and remove files, change file size and permissions, and retrieve file information. Unfortunately, there is still no easy way to perform these functions for files in the host file system.

The ELF monitor. Since the satellite had a file system, we wanted a way to create, remove, examine, and copy files interactively from a terminal connected to the satellite. We also wanted to be able to use this terminal as if it were directly connected to the host. The monitor provides these functions. Although provided as a general utility, ELF is simply another program that runs under the PARASITE-FS system. To understand ELF's operation, it may help to consider the arrangement of the user's console, the satellite, and the host (see Figure 2).

The PARASITE-FS system is arranged with the user terminal or console connected to the satellite. After the satellite bootstrap has been performed, the ELF monitor runs to act as the interface between the user, the satellite, and the host. The host system is connected by a serial line to the satellite. This line is normally used through the packet driver. However, the ELF monitor can access it directly, as can any user program as long as the packet driver is not running.

Text typed at the console is interpreted by ELF as commands. Most commands are used to examine or alter the contents of the satellite file system; these commands are modeled after UNIX commands and include utilities for creating and removing files, changing the file size and permission, copying, moving, and concatenating files, changing directories, and so on. The ELF monitor is used to run programs that are in the satellite file system.

Figure 2. Arrangement of the pieces of a PARASITE-FS system: the host, satellite, file system, user's console, and other devices.

The monitor also has a command, "hcu," that simulates a direct connection of the console and the host, similar to the interprocessor communication facility provided in the UNIX system. Except for certain escape sequences, all text typed to the console after hcu is invoked is passed directly to the host. In this mode, the user can edit and compile programs or perform other operations on the host. The escape sequences allow the user to transfer files between the satellite and the host and to escape out of hcu back to the ELF monitor.

With the addition of the satellite file system and monitor, programs for controlling experiments are developed, compiled, and linked on the host system. Programs, parameter files, and stimulus files are transferred to satellite files using the hcu command of the monitor program. At any later time, the program can be loaded into the satellite processor memory and executed without further interaction with the host. Although in most applications it is more efficient to read and write files on the satellite system, it is still possible (as with the PARASITE system) to access files on the host system. Both host and satellite files are referenced through the same set of interface routines. When a program has finished executing, control returns to the monitor and, after any possible data reduction is done, the data are transferred from the satellite to the host for further processing.

New PARASITE-FS Device Drivers

Under the PARASITE system, all devices including files are controlled using a standard structure, that of the UNIX operating system, for the device drivers. There are advantages in treating real-time devices in this fashion.

Namely, it is possible to turn off or initialize all devices in a device-independent fashion, and multiple versions of the same device can be handled in a straightforward way.

The PARASITE system also provides the software signal mechanism described earlier. That is, by setting control flags for a device, it is possible to cause the interrupt routine for that device to send a software signal when certain events are encountered during the processing of an interrupt. A user-written routine catches the signal and processes whatever value is passed with the signal.

The major problem with the PARASITE system's structure for handling real-time devices is overreliance on the software signal mechanism and the inefficiency of the word-at-a-time approach to transferring data. For most devices, storing a sequence of values involves either repeatedly reading the values from the device or supplying a signal-catching routine that is invoked by the interrupt handler for each sample value. Explicitly reading each value complicates the programming of other functions done simultaneously with the sampling. The signal mechanism, on the other hand, involves overhead that limits sampling rates to well under 5 kHz, which is not nearly fast enough for speech and other applications. The situation is worse when sampling from multiple sources.

A faster and more efficient approach is to have device interrupt routines automatically buffer input values in arrays provided by the user. Also, it is more efficient if drivers for multiple-channel devices are capable of sampling or controlling more than one channel at each interval. Within this structure, the software signal mechanism still provides a way to customize interrupt processing.

A/D and D/A converters. We replaced the PARASITE system drivers for the A/D and D/A converters with drivers that incorporate the features described above. For example, the A/D converter stores samples in an array provided by the program that controls the experiment. The program also passes to the driver the address of an index used to indicate the most recently stored sample. Because the experiment program has access to the storage vector and the pointer, it can (1) keep track of the sampling process by monitoring the index location, (2) change the value in this location, thus forcing samples to be stored at a new location in the sampling array, and (3) perform computations on the sample values that have been stored.

Subject's keyboard and input response buffer. We wrote drivers for the subject's keyboard and the input response buffers using these same features. Since input from these devices is received at irregular rates, the time of each response as well as its value are stored.

Output buffer. This driver provides the usual operations of reading and writing an entire 16-bit word. Additionally, it allows the user to set and clear selected lines of the output buffer.

Display. A high-speed, high-resolution visual display is used in most of our research. The hardware we use includes a Hewlett-Packard 1311A display scope, with a fast-decay P4 phosphor, controlled by a variable raster-vector display processor that is interfaced through a stack-controlled DMA channel.

The driver that we wrote is unusual within the PARASITE framework, since it controls two devices: the display hardware and a dedicated programmable clock. The clock is used to time a refresh rate for the display. Whenever the display clock generates an interrupt, the display is restarted, if it is not still running. If the display is still running after the refresh interval, a signal can be generated indicating this error. By setting a refresh interval longer than the time required for the largest set of displays, it is possible to ensure that the intensity of the displays does not change as displays are added.

PARASITE-FS User-Level Routines

One problem with the PARASITE system's structure for controlling real-time devices is that it does not provide a good user interface. To use a real-time device involves a sequence of operations, some with complicated calling sequences, that we have found tend to be a source of coding errors.

Real-time devices are, for most purposes, not interchangeable; unlike a file and terminal, it makes no sense to replace a programmable clock with a display. The cost of allowing this interchangeability is having to refer to a device by two different names. It is necessary to first open a device with a string that names that device. The open call returns a file descriptor that must be used in all future references to that device.

A related problem has to do with the amount of information that must be specified to control some devices. The PARASITE system correctly takes the approach that all aspects of a device that can be controlled should be accessible to the user. However, for many available control options, a default value is acceptable most of the time. The result is that users wishing to perform standard operations must learn about non-standard operations also. Finally, there is the problem that the device-independent subroutine structure involves processing overhead, as does the ability to establish settings for all control options.

Our solution to these three problems is a set of device-dependent user-level control routines that coexist with the standard device-control routines of the PARASITE system. These routines do not allow device interchangeability. They each implement a standard function. Both of these restrictions allow the routines to be written so they run faster. Another important feature is that the device to be used and the function to be performed are determined by the name of the routine, not by arbitrary values of file descriptors or flags. Finally, the user need not specify standard options.

We provided user-level routines for the clock, subject's keyboard, input buffer, output buffer, and A/D and D/A converters.

Under this rubric, we are also including routines that perform generally useful functions that are not directly related to the operation of devices. These currently include various mathematical functions, routines to generate stimuli, and new routines to format I/O.

Display Specification Package

This package of user-level routines is used to specify and modify display tables for use with the display hardware and to control the operation of the hardware and the driver.

Pictures are created as hierarchical picture structures, any number of which can be displayed at once. Key features of the picture structures include: (1) four basic picture types—points, lines, raster or vector characters, and arbitrary raster or vector figures; (2) specification of picture content independent of picture size, intensity, or relative position; (3) complex pictures created by hierarchically combining basic pictures or other complex pictures; (4) no limit on the number of times a subpicture can be added into picture structures; (5) offsets that determine placement of subpictures within a complex picture that can either (a) each be relative to the point that the last subpicture stopped or (b) all be relative to one point, the starting point of the picture; (6) size, intensity, and relative position information that can be incorporated at any level in a picture hierarchy.

This package allows a complex subpicture to be built from a set of basic pictures. Multiple instances of this subpicture can then be used, possibly with other subpictures, to build a new complex picture. This can be done, since all placement of pictures within a picture structure is relative; it is only at the top level of the picture structure that absolute coordinates are specified. Since the content of a picture is specified independently of its size, intensity, and relative position, these parameters can take on different values for every occurrence of a subpicture. If no values are explicitly given, reasonable default values are supplied.

Other features of the display specification package include: (1) facilities to change the size, intensity, and relative position of a subpicture while it is being displayed; (2) facilities to add or remove pictures from any point of a hierarchical picture while it is being displayed; (3) the ability to change global display parameters, including refresh rate, default size, and default intensity; (4) facilities to synchronize the display refresh operations with other events; (5) dynamic allocation of storage for displays.

Activity-Event System

This package controls the timing of sequences of

activities that are started by the occurrence of events (e.g., a keypress, a saccade, or the end of an utterance). An activity sequence consists of subroutine calls at specific delays from the time of the initiating event. Several activity sequences may be in progress at one time. This package acts as a scheduler that (1) reduces the amount of user code necessary to perform a sequence of activities, (2) allows the order of activities to be varied by changing the value of variables rather than program text, and (3) enforces a discipline of specification that reduces the chances for errors in the timing of multiple events.

Before the start of an interval containing activities that must be timed, the user's program defines one or more activity time sequences. An activity sequence consists of two kinds of specifications. The first is the time, relative to the time the activity sequence is initiated by an event, when each activity in the sequence should be started. An activity is any operation that can be performed by a subroutine. This specification can also contain information about the relative importance of each routine and an indication of how late it can be started without being an error. The second type of specification indicates which events to anticipate at what times during the time sequence (again, relative to the time this activity sequence was initiated). An event here is anything that can send a software signal. This specification also determines which activity sequence should be started if the event occurs, an indication of how many times the event may occur, and an action to be taken if it does not occur.

Once the activity-event system has this information, the clock is reset and one activity sequence is explicitly initiated. The activities in this sequence will be started with the specified delays. If an anticipated event occurs, the activity sequence associated with it is started and both activity sequences continue in parallel. If more anticipated events occur, more activity sequences are started. Conflicts between activities, that is, between two activities to be performed at the same time, are resolved according to their relative priorities. This process continues until (1) all the specified activities have been performed and no more events are anticipated, (2) some activity cannot be started at its appointed time, (3) some event occurred that was not anticipated, or (4) the process is explicitly stopped by an activity.

The power of this package is the freedom it gives users to establish contingencies and its ability to sort out complicated interactions. One problem in using it is that, even for simple cases, a large amount of information must be specified. We are currently trying to work out a system of default values and a way of specifying them that will make this package easier to use.

SUMMARY

The PARASITE-FS satellite system was designed to control experiments that require real-time processing of stimuli or data. Because it includes a disk file system on the satellite, it is especially useful for tasks that, because of the data rates or the amount of storage required, need a local file system. Real-time device drivers that automatically buffer data transfers in user memory make fast interrupt processing possible. The user interfaces have been designed to make the task of implementing experiments easier, especially for the beginning user. The activity-event system provides a framework for specifying and controlling the timing of events and processes that take place during an experiment.

REFERENCES

- LYCKLAMA, H., & BAYER, D. L. The MERT operating system. *Bell System Technical Journal*, 1978, **57**, 2049-2086.
- LYCKLAMA, H., & CHRISTENSEN, C. A minicomputer satellite processor system. *Bell System Technical Journal*, 1978, **57**, 2103-2114.
- MURREL, S., & KOWALSKI, T. A real-time satellite system based on UNIX. *Behavior Research Methods & Instrumentation*, 1980, **12**, 126-131.
- RITCHIE, D. M., JOHNSON, S. C., LESK, M. E., & KERNIGHAN, B. W. The C programming language. *Bell System Technical Journal*, 1978, **57**, 1991-2020.
- RITCHIE, D. M., & THOMPSON, K. The UNIX time-sharing system. *Bell System Technical Journal*, 1978, **57**, 1905-1930.
- VESONDER, G. T., LESGOLD, A. M., & WOLF, R. V. The use of microprocessors in a timeshared computer facility for research, development, and teaching. *Behavior Research Methods & Instrumentation*, 1978, **10**, 510-515.

NOTE

1. UNIX is a trademark of Bell Laboratories.