SESSION III PRESIDENTIAL ADDRESS

PETER G. POLSON, *University of Colorado, President*

Maximizing the mini-uses of on-line computers

LEE W. GREGG

Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

This paper draws on our experiences at Carnegie-Mellon University to provide a decision framework for the acquisition of computer systems for laboratory applications. Also recommended are ways of creating an effective operating environment that can handle changing needs. Systems developed for our most recent research in eye-movement recording and children's concept learning using the Turtle will illustrate the design principles for hardware and software components of minicomputer systems.

Ten years ago there was a raging argument in small-scale computer technology between those who preferred timeshared utilization of computer systems and those who believed that small dedicated computers were the solution to most laboratory problems. Uttal, in 1972, presented the pros and cons of timeshared vs dedicated computer use and introduced some additional questions. For example, "Who does the programming? You or a hired hand?" I think it was at that same meeting that Murdock reported that his new computer made it possible for him to do half as many experiments as he previously had done at twice the cost. His hired programmer had discovered a way of displaying ASCII strings on an oscilloscope using FORTRAN format read and write statements. Other pioneering efforts of that era include networking. Networking was brought about by tying up a small-scale computer with a large-scale machine usually in the University Computation Center. By 1972, we had devised such a system and demonstrated it disastrously during Hurricane Agnes in June of 1972, in Pittsburgh. Agnes won.

By 1974, the proponents of the small dedicated minicomputer were clearly winning the battle, although there were exceptions to the notion that the small individual dedicated minis would soon capture the entire market. Earlier intermediate-scale systems included the Clipper Laboratory at the University of Colorado, which was established in 1970, the Indiana and Michigan 1800-based systems conceived by Restle and Greeno, and our own laboratory at Carnegie-Mellon University.

There are several reasons why dedicated minicomputers have achieved the market they now claim. First, computing systems have simply become cheaper. In just the past 2 years (\$25,000 to \$12,000) DEC PDP-11 systems have gone down in price by a factor of 2. This

means that the individual investigator can have the computing power that normally was reserved for the intermediate-level timeshared laboratory facilities. But the decision problem of the laboratory investigator still remains one of maximizing the type of system he can use in terms of both time and money. Presumably, less expensive systems could be purchased if sophisticated software was to be generated by someone other than the vendor. And what about data analysis? Most of us are familiar with statistical packages and data processing procedures. We can implement those ourselves if we have a large enough computer on which to operate. I guess it is the dream of most of us to be free from the hassle of computation center personnel. Unfortunately, the decision is not merely economic or technical—it is also political.

Fundamental to the process of making a decision about the acquisition of a computer are estimates of required capacity and future growth. These estimates in turn depend on the uses to which the computer system will be put. In what follows, I am going to discuss the nature of operating systems for particular hardware configurations, the nature of programming languages that have been developed for small-scale computer use, and some principles that I believe are important in determining the direction software development for minicomputers will take.

MINICOMPUTER CONFIGURATIONS AND OPERATING SYSTEMS

The minimum hardware configuration for a dedicated computer user consists of a central processing unit (CPU), a communication device like a Teletype, a device to be controlled that may range from scopes and pushbuttons for cognitive experiments to levers and

lamps for behavioral experiments, and appropriate mass storage. The size of the mass storage is determined by the nature of the experiments. Generally the individual experimenter wants to put together, as quickly and as efficiently as possible, programs to support his research effort. In our usual view of computing structures, an operating system is interposed between the user and the hardware of the computer for two reasons. First, it is desirable to have well-organized programs to handle the complex (but routine) I/O operations. Second, an operating system adhering to certain conventions makes it possible for the computer user to program and be reasonably independent of the hardware configurations and idiosyncracies of a particular system. A user can program in FORTRAN on any of a number of minicomputers available on the market.

The basic I/O functions of a supervisory system are to assemble, load, and store user programs, copy information from one storage medium to another, communicate with various devices ranging from the user's console to the experimental apparatus itself, and edit or debug machine language or higher language programs. These utility routines may be integrated into a single operating system.

Intermediate-scale computing systems shared by more than one user impose special and new demands on an operating system. An algorithm for job sequencing is required; decisions about foreground or background operations must be made and one user program must be protected from another. So the operating system is extremely complex. Furthermore, since there are alternative ways of compiling or assembling programs, decisions must be made about job scheduling and resource allocation. Thus, the complexity of the operating system for any timeshared machine goes up drastically as the number of job types increases. As will become evident, there is a tradeoff between the simplicity of a small dedicated specialized system and the large timeshared system in terms of the exportability or interchangeability of programs and functions which the computers can carry out. Small machines can be simple, but then the functions they perform are machine dependent and not transferable. The large machines, though complex, are generalizable in the sense that functions carried out on one can be carried out almost as readily on another.

The resident portion of an operating system for a single-user minicomputer with real-time capabilities will take up about 2K of 16-bit core. In a single-user configuration, the minimum memory size for carrying out useful psychological experiments is probably 8K. In a computer of that size it would be necessary to assemble programs on another larger machine and transfer only the binary coded instructions by means of paper tape, magnetic tape, or disk pak. The minimum machine size for carrying out one's assemblies is probably 16K of core and requires an external storage device. Our preference has been a disk system that is now priced within the reach of a small research project.

At the other extreme is a general-purpose multiuser

interactive operating system designed for computers that have memory management, large core capacity of up to 128K, and multiple languages and subsystems. UNIX is one of these and it can run on hardware costing as little as \$40,000, but the more usual configuration is close to \$100,000. UNIX, according to its authors, Dennis Ritchie and Ken Thompson (1974), offers features seldom found in much larger operating systems designed for very much larger computers. UNIX is a complete system providing a hierarchical file system, compatible device and interprocess I/O, the ability to carry out and initiate asynchronous processes interrupts in real time, a system command language selectable by the individual user and over 100 subsystems including a dozen computer languages. Like all operating systems, UNIX separates the command language from the programming language of the user's choice. The size of UNIX is 21,000 16-bit words. This minimal operating system, of course, expands to many thousands of words depending on the devices, the I/O buffers, and system tables required. A minimum system for running UNIX is about 25,000 words of core, but obviously most of the core is UNIX.

PROGRAMMING LANGUAGES FOR MINICOMPUTERS

Operating systems make it easier for the user to write programs in a programming language of his choice. But what choice does the minicomputer offer? The Digital Equipment Corporation supports a number of software packages for machines of a mini configuration (the smallest configuration described earlier). BASIC runs under an operating system RT11, as well as FOCAL/RT11, FORTRAN/RT11. There is also a basic plotting package, a laboratory applications package, library and incremental plotter, and other software. UNIX, the operating system developed at Bell Laboratories, supports a variety of languages but is designed to use C, a programming available on UNIX and on OS/360. This language offers modern control structures permitting flow of control without gotos. There is an economy of expression in this ALGOL-like language that eliminates many temporary variables and trivial statements which, according to the authors, give shorter and clearer codes as adequate linkage that encourages modularity and good program organization. This makes debugging easier. And, finally, there are facilities for many different kinds of data structures including pointers, character string, simple nonnumeric problems all the way up to complicated data aggregates.

Incorporating both the idea of laboratory-oriented control language and the principles of psychological research are languages like SKED and SCAT and the extended SCAT. These languages are capable of generating many kinds of psychological experiments, but they are useful principally in behavioral studies of animal learning and performance. A series of languages have been developed at Carnegie-Mellon University as well.

These include PSYCHOL (McLean, 1969), APCOL [reported here by Pinkus and Gregg (1973)], and more recently our version of a LOGO-like language which we now call TURTLE JARGON. Of course, at the base each experimenter can program in his own machine language. Once again the tradeoff is between machine dependence at one extreme and exportability, or generality, at the other. Exportability means that a programmer/psychologist can take his experimental control program from one computer to another and under some appropriate circumstances hope that the algorithm will run properly. Generality means that it is possible to incorporate a large number of experiments of a single programming language. One way to do this is by writing a single program which is so general that it can subsume all experiments. The other approach is to have a programming system that is so flexible, so easy, and so quick that new programs can be constructed very rapidly.

This was our initial hope in designing the PSYCHOL language and we almost made it. The language incorporated many features that are critical for real-time programming, such as temporal variables. But PSYCHOL was not exportable; it resided as a disk-based version on an essentially unique machine, the Honeywell DDP-116, which had very little use in other psychology laboratories. Our second attempt was APCOL. APCOL was programmed in FORTRAN; therefore, it should have been exportable. It resided on the computation center's large-scale IBM 360/367. Thus we had access to the text editing and I/O equipment of a large computer. But APCOL failed for a couple of reasons. First, a programming language has to be able to do whatever the experimenter wishes within the constraints of the system. APCOL was restricted in scope, it was slow in execution (even after it was compiled), and it lacked the transparency that we would like to see in a computer language for psychologists. This meant that APCOL became a fairly difficult language to program, even though it had a FORTRAN-like surface structure familiar to all.

The difficulty in this discussion thus far is that there exists no convenient mapping of devices and languages for experimental control. Some of the earlier operating systems, including the SCAT system at Colorado and the SKED system, required mapping skeletons, a control program, and a separate program for recording. This was also true of the IBM 1800 systems used at Indiana and Michigan. Hence, I will argue that there is still room for programming language development for minicomputers. Real-time BASIC-PLUS certainly has some of the properties that we desire.

DESIGNING LANGUAGES FOR REAL-TIME CONTROL

Why should we consider the problem of developing new computer languages for controlling experiments?

Higher order languages have already been designed and implemented. Is it not just a matter of time until a language such as real-time FORTRAN becomes the programming standard? I am going to argue that real-time control of psychological (and other) experiments can never be adequately programmed by a single higher order language and that although higher order languages are implemented for the purpose of insulating the user from the intricacies of the operating system, it is precisely these intricacies that determine whether or not a particular program will work as intended.

If we examine the critical parameters for real-time control, we see that each involves detailed knowledge of the processing times for executing lower level I/O or hardware-dependent operations. First, processing times for calculating stimulus values and their representations must be considered in order to determine the tradeoff between storage capacity vs sequential generation of stimuli. Can the computer retrieve or generate stimuli fast enough for the desired presentation rates? And, for computers with limited memory capacity, a second critical parameter becomes the time for accessing auxiliary memory. Is there time enough between presentations to read new information into core? A third critical parameter determines whether the stimuli can be presented properly. Knowledge of the refresh rate of the terminal display and transfer rates to terminal devices is essential for display and exposure of the stimuli. The problems associated with stimulus presentation are compounded by the cognitive and information processing psychologist's need to present substantial amounts of text under precisely controlled conditions. A fourth parameter is the processing time of the interrupt handlers. In addition, detailed knowledge of the priority of interrupts is necessary if control decisions about the responses of subjects are to be correct. Decisions about recording responses or storing them for later analysis may involve the same read/write parameters for disk access as those noted in stimulus retrieval. Lastly, control decisions involving the logic of the program itself depend on the critical parameter associated with the basic memory cycle times of the particular computer. Computers are fast, but as it turns out, we can always place demands on them that exceed their capability.

These critical parameters are precisely the ones that a systems programmer must deal with in isolating the user from the operating system. The main reason higher order languages have worked is because the user cared little about timing as such and because the flow of a program is conceived of as linear. In fact, real-time control programs incorporate, indeed require, the basic concepts of timesharing to make them work.

Consider the data recording function. In small computers the responses must be either held in core or written out to an auxiliary device, that is, tape, disk, or a hard-copy terminal. If there are large amounts of data, it becomes unreasonable to purchase enough core or impossible to address enough core in 16-bit machines.

Hence, I/O must be performed. Since the experiment is carried out in real time, we cannot simply isolate the user program from complexities of I/O; we must time-share the write operations with the logical control of the experiment. Thus, any I/O operation in real time demands a timesharing monitoring system. But the basic functions of the computer are not the only ones that must be shared. The experimenter who monitors the ongoing experiment must be free at any point in time to halt the experiment, preferably from two or more physical places. He should have a signal switch in the room with the subject and perhaps a console intervention through a command language in a different room. The program itself is a surrogate for the experimenter. Thus, two parallel streams of activity, experimenter's and subject's behaviors, must be managed by the control program. And finally, to carry the example to the limit, experimenter has two hands, so that at least two possible performance routines can be carried out simultaneously by the control program (for instance, adjusting both duration and attenuation dials prior to presentation of an auditory stimulus).

These critical parameters constrain the execution of the programs, but their very nature suggests that the usual distinction between operating system and programming language is invalid.

PROTOTYPES FOR THE DESIGN

The most recent Roster of Programming Languages (Sammet, 1976) lists 172 higher order, active languages in 1974-1975. Surely, from a list this large, there must be a suitable prototype for the design of a minicomputer control language. As can be imagined, most of the languages are for numerical and scientific applications, and there are 28 such languages. A total of 18 languages are designed either for continuous or discrete simulation, but only two languages are designed specifically for real-time control of processes. One is implemented on special-purpose hardware and the other is designed to check out software models, in real time, written in another special-purpose language, SVDSS. The latter is a simple language for modeling space vehicle data systems.

There are two classes of languages that might be candidates for controlling real-time laboratory experiments. First, multipurpose languages that are extensible are represented by LOGO or SAIL. The second class of languages is for computer-assisted instruction and contains PLANIT and TUTOR. Multipurpose or extensible languages can perform a variety of functions, but a great deal of modification would be required to incorporate real-time clock functions and direct control over hardware I/O devices. Computer-assisted instruction seems a natural. After all, the stimulus materials are presented to individual subjects, responses are recorded and checked, and these are the essence of a psychological experiment. Unfortunately, the computer-assisted

instruction (CAI) languages are highly specialized with respect to out devices; the languages are oriented toward text storage and retrieval; and the control functions are simple. At present, there are no languages that can be picked up whole cloth and modified in simple ways for the purpose of writing programs to control experiments.

The manufacturers have a much more difficult problem today than when IBM was able to claim software support for FORTRAN and induce users to purchase machines and software as a package. Minicomputers have such a wide variety of configurations, and products for them emerge from such a wide variety of sources, that software must be custom tailored to each machine in the series. Hence, laboratory users buy both hardware and software support, but the software is not exportable beyond the machine configuration or temporal version of the operating systems provided. For example, RT11 versions exist for DECtape operating systems and for disk-based systems, but are not interchangeable in simple ways. The interpretive language, FOCAL (which is too new to make the 1974-1975 roster), can live with RT11 and the 60-cycle clock, but not with the real-time clock for 11/05 systems. It is discouraging to observe that the candidate that I have mentioned as most likely to succeed as the standard real-time control language is perhaps the only one in the list of all programming languages that was not "designed" at all. FORTRAN simply was "kluged" together as computer scientists became more sophisticated with respect to symbolic languages. First, the line numbering features were invented for control purposes; then, format read and write statements were added; finally, subroutines and control statements were inserted. The only advantage of RT/FORTRAN is that many students learn the language and it is a familiar base for describing the programs.

Clearly, the objective of the design of real-time control languages is to facilitate the process of bringing up and executing real-time programs. Some users are satisfied with state notation languages or special-purpose languages peculiar to a class of experiments that are repeated over and over again. Others (the computer "nuts") will learn machine language and operating systems so that general facility with programming allows extensible experimentation. But for most of us who wish to maximize the *many* uses of real-time laboratory computers, there is still a need for a way to meet two subgoals of the general problem. We would like to learn quickly how to use a new system for a limited set of experimental operations. Ease of learning and running a limited class of experiments is one subgoal. Another subgoal is to be able to get the full power from whatever configuration we invest in. We want to be able to use the system in as general a range of problems as possible, that is, to maximize the generality of our system. What is needed is a base language in which control programs can be written that extend the base language. Some systems-building languages, like BLISS, and

some extensible languages, like LOGO, have the necessary properties.

The problem is not to isolate the user from the I/O, for that is precisely what we need to control. The problem is not to keep the intricacies of the monitor from the user. We must examine interrupts, and time interrupts and other events in the real-time sequence. Thus, the base language should have the general features of an

operating system command language. The user can select parts of it to construct programs for individual experiments, create one's own semantics, and hopefully debug and write in an interactive mode. However, the completed program should compile for run-time execution. (Notice that RT/FOCAL has the interactive feature and RT/FORTRAN the ability to compile.)

NEW DIRECTIONS

We are currently exploring a new concept of system building using BLISS to create LOGO-like languages for controlling the Turtle, a robot described at last year's conference by Seymour Papert. BLISS is not the solution in itself—it is a difficult and still obscure language. We hope that a number of the surface properties and structures of the programming language "C," described earlier in connection with UNIX, will be incorporated in the final language we will report. A sample of an early attempt is shown in Figure 1 for a language called TURTLE JARGON.

For the inexperienced programmer, it is easy to define symbolic events in the experiment and name them; for example, `DEFINE TR = TURTLE -226 R` causes the Turtle to turn right when the characters TR are typed on the experimenter's console. Procedures can be created and assigned to events like buttonpresses, typed characters, or outcomes of responses. Responses can be recorded, analyzed, and stored. The language has limited control and data representation. The only data structures are "stacks" in which all executable and symbolic information resides until disposed of.

TURTLE JARGON, then, represents the programmed output from a base language that is both extensible (i.e., it is Turtle Jargon) and powerful (i.e., other data structures of indefinite varieties can be generated). TURTLE JARGON, however, is a simple offshoot of the base language for a particular experimental situation. And its acronym? Well, none was intended, but as the wave of the future it may mean "Tool Up Real-Time Language Extensions."

REFERENCES

- McLEAN, R. S. PSYCHOL: A computer language for experimentation. *Behavior Research Methods & Instrumentation*, 1969, 1, 323-328.
- PINKUS, A. L., & GREGG, L. W. APCOL: A programming system for computer-controlled psychology laboratories. *Behavior Research Methods & Instrumentation*, 1973, 5, 165-172.
- RITCHIE, D. M., & THOMPSON, K. The UNIX time-sharing system. *Communications of the ACM*, 1974, 17, 365-375.
- SAMMET, JEAN E. Roster of programming languages for 1974-75. *Communications of the ACM*, 1976, 19, 655-669.
- UTTAL, W. R. Misuse, abuse, overuse, and unuse of on-line computer facilities by psychologists. *Behavior Research Methods & Instrumentation*, 1972, 4, 55-60.

```
.R PIP
TT:♦=INIT.TRT

CLOCK - 0
CONSOL -
BSW -
FILE -
TURTLE -

DEFINE TL = "TURTLE - 226 L "
DEFINE A = "TURTLE -;BSW - HEY "
DEFINE Z = "BSW - "
DEFINE C = "FILE - (FILE) CR (CLOCK) "
DEFINE PLAY = "FILE - (FILE) CR (CLOCK) PLA "
DEFINE T7 = "TURTLE - 226 R 226 R 750 F "
DEFINE FR = "TURTLE - 750 F "
DEFINE TR = "TURTLE - 226 R "
DEFINE PR = "FILE - (FILE) CR (CLOCK) PROBLM "
DEFINE T4 = "TURTLE - 226 R 750 F "
DEFINE T11 = "TURTLE - 750 F "
DEFINE T5 = "TURTLE - 226 L 750 F "
DEFINE T1 = "TURTLE - 750 F 750 F "

PROCEDURE BE12
FILE - (FILE) CR (CLOCK) L
IF (TURTLE) NEQ THEN FILE - (FILE) =
IF (TURTLE) = THEN TURTLE - 226 L
END
PROCEDURE BE13
FILE - (FILE) CR (CLOCK) F
IF (TURTLE) NEQ THEN FILE - (FILE) =
IF (TURTLE) = THEN TURTLE - 750 F
END
PROCEDURE BE14
FILE - (FILE) CR (CLOCK) R
IF (TURTLE) NEQ THEN FILE - (FILE) =
IF (TURTLE) = THEN TURTLE - 226 R
END
PROCEDURE B12
IF (BSW) = THEN BE12
END
PROCEDURE B14
IF (BSW) = THEN BE14
END
PROCEDURE B13
IF (BSW) = THEN BE13
END
DEFINE N = " HORNOK - TRUE"
DEFINE O = "HORNOK - "
PROCEDURE B22
IF (HORNOK) NEQ THEN TURTLE - H
END
HORNOK - TRUE
♦
```

Figure 1. Sample of early attempt for a language called Turtle Jargon.