Computerized process control in behavioral science research*

KARL W. SCHOLZ
Indiana University, Bloomington, Indiana 47401

A general approach to the design and development of real-time operating systems is discussed. Operating system design for small to medium scale laboratory computers is described at a moderately elementary level. Analysis of system design as a supervisory control hierarchy is presented in an attempt to bridge the gap between an elementary general understanding of computer operation and the more sophisticated understanding assumed by the writers of most computer systems operator's manuals. PROSS, a programming language developed at Indiana University, is presented as an example of the highest level of supervisory control.

Recent technological advances in computer design and manufacture have brought the possibility of computerized laboratory control within the realm of even modestly endowed research establishments. Yet, despite the spectacular increase in hardware sophistication and decrease in price, development of adequate software has, at best, proceeded at a moderate pace. The behavioral science investigator who adds computerized equipment to his laboratory is frequently forced either to hire a programming expert or to devote the time necessary to become an expert himself. Recognition of this unfortunate situation has prompted our group at Indiana University to devote considerable effort to the development of techniques allowing the nonsophisticated programmer to benefit maximally from a computerized laboratory. Our investigation has led to a definition of the major problem areas in process control computation and to the development of a programming language which limits the consideration of these problems to the implementers of the language, allowing the user of the language to focus attention on his research rather than on the techniques required for its study.

The following pages describe the major problems encountered in the implementation of a computer-controlled laboratory and outline methods for their solution. First, general system design considerations will be discussed in order to clarify the distinction between conventional numerical analysis programming and process control programming. An analysis of system design in the context of levels of supervisory control will be described. Secondly, a computer language designed to facilitate man-machine interface will be described in detail.

*This work was partially supported by PHS Grant No. MH16817.

THE PROCESS CONTROL COMPUTER

The major differences between conventional numeric computation and process control computation fall into two categories. The first category concerns specialized input/output, which in a typical laboratory setting consists of either digital switching circuits which are responded to or controlled by the computer or analog input/output (I/O) devices used for physiological or acoustic research. The second category concerns the timing of events in real time.

Input/Output

Digital analog output (DAO). DAO from the computer is typically effected by transferring the contents of an entire word or byte of computer memory to the appropriate output register. More sophisticated systems generally provide data-channel or direct-memory-access channel control of DAO, allowing the computer to issue a single "start output" instruction which initiates hardware-controlled transfer of entire blocks of memory to the output register. On the other hand, simpler hardware configurations or unique applications require precise control of single digital switching circuits which operate relays or stimulus lights. Such independent switching is easily managed if the output-controlling program maintains in memory a copy of each output register's current status. Such a practice permits modification of an arbitrary number of output bits in any register while insuring that the remaining bits, possibly assigned to a totally unrelated function, are left undisturbed.

Analog output from digital computers is generally controlled by a digital-to-analog converter (DAC), an inexpensive device consisting of little more than a network of resistors, and a small voltage amplifier. DACs convert the configuration of on and off bits present at their input to a corresponding discrete output voltage. Effectively continuous voltage change at the output of a DAC is achieved by outputting a continuous series of computer words or bytes from the CPU to the DAC at high data rates. Thus, although it would be possible to control a DAC with discrete program-controlled outputs, typical configurations drive a DAC with a direct-memory-access channel.

Digital input (DINP). Digital input to a computer generally presents more of a problem than DAO, since asynchronous external events will frequently demand immediate attention by the CPU which is currently involved in unrelated activities. Two techniques are commonly used to manage DINP in a manner designed to insure rapid response to external events. First, on computers equipped with priority interrupt hardware (see discussion of interrupt servicing below), arbitrary groups of input bits are assigned via interrupt request circuitry to subroutines which are automatically entered each time an external circuit becomes active. Secondly, computers lacking sophisticated interrupt-servicing hardware may be programmed to poll periodically all possible input registers and to branch to appropriate service routines if the status of any register has been altered by some external event. In the latter case, response time is limited to the polling frequency and may therefore be subject to objectionable variation.

Several methods are used to input analog information to digital computers. An inexpensive approach is to use a comparator, a device which monitors an analog voltage and outputs a digital warning signal if the voltage exceeds or drops below preset limits. However, the simple comparator is inadequate for most applications in the behavioral sciences, especially in psychoacoustics and

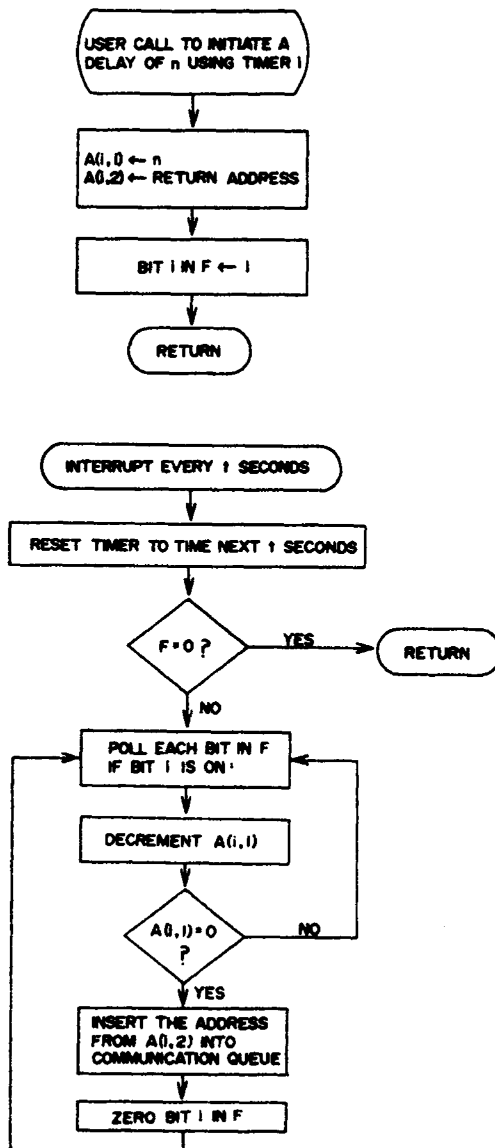

Fig. 1. An abbreviated algorithm for implementing several programmed timers using a single hardware timer. See text for explanation.

physiological psychology, where a continuous record of a varying analog signal is required. Such input is provided by a device known as an analog-to-digital converter (ADC), or digitizer, which periodically samples a varying input voltage and produces the digital representation of the voltage at its output. Depending on price, ADCs vary in sampling frequency from a few thousand to over 100,000 samples per second. Like analog output, analog input typically requires a direct-memory-access channel to adequately handle the high data rates.

Event Timing

The second major difference between numeric computation and process control lies in the area of event timing. Many computers designed for process control include at least one programmable hardware timer which is wired by the manufacturer to operate at a particular user-selected frequency. Such timers operate by incrementing a particular program-accessible register (often a location in main memory) until it reaches zero, at which time an interrupt is forced, transferring control to a user-written timer servicing routine. Most real-time applications require simultaneous timing of several independent events, yet the CPU may contain only a single hardware timer. Multiple event timing is usually achieved by providing several programmed timers driven by a single hardware clock. Programmed timers generally use two memory registers and an indicator bit for each timer (see Fig. 1). A user initiates a time interval by setting one register to the time he wishes to delay, a second register to the address of the program to be executed at the end of the interval, and the indicator bit to "on," indicating that a programmed timer is in use. The single hardware timer is set to interrupt to the timing service routine at an installation-determined frequency by loading the timing register with the appropriate value. At each timer interrupt, the timer is restarted to time the next interval, then all the indicator bits (frequently placed in a single word) are sequentially polled. For each bit set to "on," the associated time delay register is decremented by one and tested for zero, and if zero, the return address is entered into a queue for transfer back to the user's program. The number of programmed timers in such a system is limited only by available memory and consideration of instruction execution delay in the polling/queuing procedure.

INTERRUPT SERVICING

At this point, this discussion will shift from a general description of process control computation to a specific analysis of interrupt servicing. After a brief discussion of hardware interrupt circuitry, the programming problems which face the system designer will be discussed in detail.

Hardware Interrupt Circuitry

Most modern computers include an instruction such as a "return jump" to facilitate communication with subroutines. Return jump procedures typically operate by preceding an unconditional branch with storage of the program counter in a readily accessible location. Saving the program counter at the time the branch is executed allows the branched-to program to return to the instruction following the return jump. Hardware interrupt circuitry operates by forcing the execution of a return jump instruction to a specific memory location. A routine designed to service the interrupt will generally begin at this location. At the termination of interrupt servicing, the routine will return to the interrupted program using the stored value of the program counter.

Interrupt Servicing Software

An interrupt service subroutine (ISS) will generally be divided into three parts: a prologue, a body, and an epilogue. The prologue, executed immediately upon entry to the ISS, saves active central processor registers. The body of the ISS analyzes the appropriate input devices and takes whatever action the analysis dictates. The epilogue restores all active registers and returns to the interrupted program.

Several important considerations must be borne in mind when designing an ISS. Foremost is the issue of reentrancy: the fact that an ISS may itself be interrupted. The nature of the reentrancy problem should be clear in the following example. Suppose that program PROG is executing and an interrupt occurs which transfers control via the forced return jump to subroutine ISS. ISS begins by saving the active registers and the program counter (all of which represent the machine environment of PROG) in location SAVE. As the ISS is attempting to analyze the interrupt, a second external event occurs, forcing a second return jump to the beginning of the ISS. Again, the ISS obediently saves the active registers in SAVE, overlaying the previous register values for PROG. Thus, when the epilogue in ISS restores the registers, control is transferred back to the location within ISS which was in execution when the second interrupt occurred. ISS will then proceed to the second execution of its epilogue, and the system will loop indefinitely.

A second problem occurs when the interrupt analysis in the body of ISS indicates the necessity for the execution of some sequence of programs which is excessively time consuming or one which requires the same core storage locations currently in use by the interrupted program. In either case, it is often desirable to postpone program execution at least until the interrupted program completes execution. The scheme outlined below provides solutions to such problems.

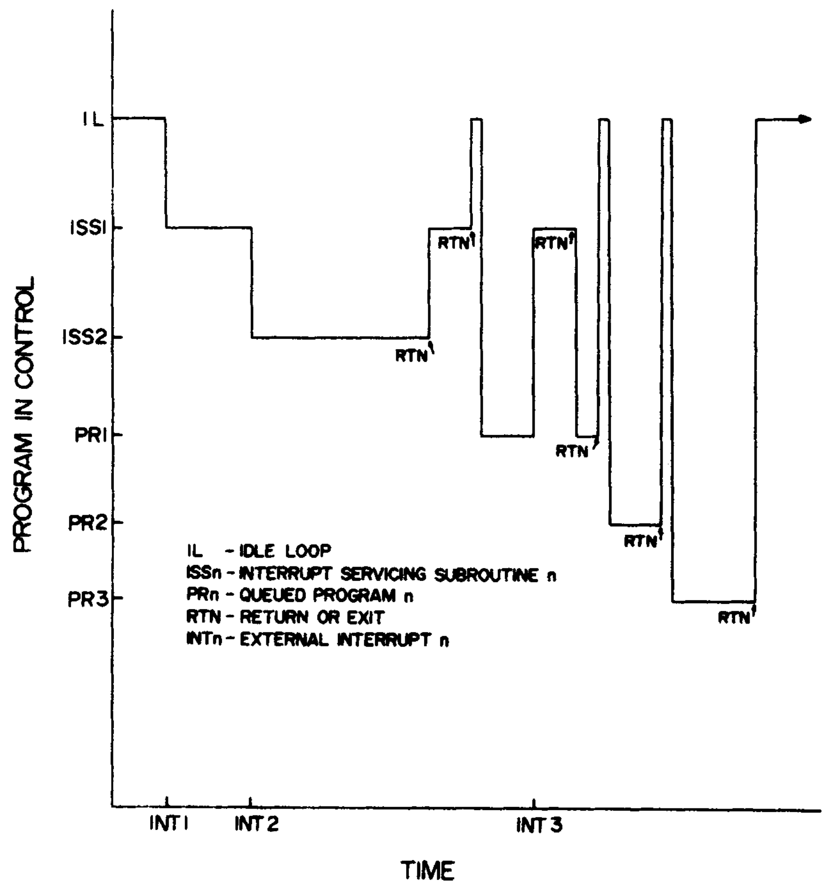

Fig. 2. Interrupt servicing by the resident monitor. In the operation summarized above, time starts at a point where the Idle Loop is in control. External Interrupt 1 occurs and transfers control to ISS1, which queues program PR1 before returning. While ISS1 is in execution, a second interrupt occurs and control is transferred to ISS2, which queues program PR2. ISS2 then exits, allowing ISS1 to complete and return control to the Idle Loop. The loop then interrogates the Communication Queue and finds and executes the call to PR1. While PR1 is in execution, a third interrupt occurs and control is transferred to ISS1. ISS1 queues PR3, then returns to PR1 (which it had interrupted). When PR1 finishes, control returns to the Idle Loop which calls PR2 and PR3 in turn. At this point, the queue is empty, so control remains in the Idle Loop until the next interrupt occurs.

SUPERVISORY CONTROL 1: THE RESIDENT MONITOR

At the nucleus of a real-time operating system is the resident monitor, a group of subroutines and communication tables which are, by virtue of their frequent use, permanently core resident. The resident monitor can be conceptually divided into five units: the Idle Loop, the ISSs, the Interrupt Save Stack, the Communication Queue, and the System Utility Functions. These units will be treated in turn.

The Idle Loop consists of a sequence of instructions which continually test pointers in the Communication Queue (see below) and branch out of the sequence only if pointer interrogation indicates that the queue is not empty. The Idle Loop is entered unconditionally by the exit statement of any program.

The Communication Queue, in the simplest case,[1] consists of a first-in, first-out queue of program or subroutine execution addresses. Entries into the queue are made by the body sections of the ISSs when interrupt interrogation indicates the necessity for communication to some program. When the Idle Loop finds an entry in the queue, the execution address at the front of the queue is removed and control is transferred to it.

The prologue of each ISS performs two functions. Immediately upon entry to an ISS, the interrupt system is temporarily disabled, or "masked,"[2] preventing all interrupts. The contents of all active registers are then placed on top of the Interrupt Save Stack. The interrupt system is then unmasked and control passes to the ISS body. If interrupt analysis indicates the need for some program, its execution address is obtained and stored in the Communication Queue. The ISS epilogue operates by again masking the interrupt system, removing the register set on top of the Interrupt Save Stack, and then unmasking and returning to the interrupted program.

The resident monitor Utility Functions consist of frequently used subroutines which perform common functions such as bit manipulation, I/O character conversion, or queue and stack pointer handling.

Once the function of each unit in the operating system is understood, operation of the system as a whole should become clear. At system deadstart (powering up the computer after it has been off), the queues and stacks are cleared, their pointers are initialized, the interrupt system is enabled, the first clock interval is initiated, and control passes to the Idle Loop. As is shown in Fig. 2, occurrence of an interrupt results in transfer of control to the appropriate ISS. Active registers are stacked, and

program starting addresses are placed into the Communication Queue if necessary. Even if ISSs themselves are interrupted, control will eventually be returned to the Idle Loop, at which point the oldest entry in the queue will be removed and executed. The nucleus of such a system (1) can be implemented on most small computers using only several hundred words of storage, and (2) permits effective utilization of all the system's resources, both hardware and software.

SUPERVISORY CONTROL 2: SPECIAL FUNCTION SUBPROGRAMS

The resident monitor which supervises I/O interrupts and event timing may be considered as the lowest or most basic level in a complete process control supervisory system. At the next higher level, special function subprograms may be provided which serve to interface the basic CPU hardware and specialized I/O equipment to the needs of the user. These subprograms, generally coded as user-callable subroutines, often reside in secondary storage and are loaded for execution only as needed. The routines are typically designed to control a particular I/O device by operating the appropriate output lines or sensing input lines associated with the device. For example, a call to one routine might result in advancing the paper in a memory drum or other stimulus presentation device, while a second call interrogates the status of input lines associated with response keys operated by the S who is viewing the drum. Such routines effectively isolate the programmer from detailed machine language bit-oriented I/O programming, thus freeing him to focus attention on the general organization of the controlled process.

At this same level, programs will frequently be provided which do not actually interrogate or operate I/O lines, but rather modify various status flags associated with general I/O operations. For example, it may become necessary to place the input-detection apparatus of the computer in a masked mode for some period of time, then subsequently restore the unmasked status. Simple routines which permit ignoring or accepting external inputs may be provided, which operate by setting or resetting bits in appropriate mask words which are sensed by other I/O routines.

SUPERVISORY CONTROL 3: THE PROCESS CONTROL LANGUAGE

At the third level of supervisory control are the programs which supervise the scheduling of events in real time and the collection and storage of data. Programs at this level are normally written in a high-level, user-oriented language and serve to integrate the elements of the system described in the preceding pages into a single functional entity. Unlike lower-level routines which form the permanent operating system, such user-written programs are temporary and become part of the system only long enough to perform a particular function, such as a single experiment in the behavioral sciences. A program package of this type may be conveniently divided into two parts. The first part consists of the actual process programming package, containing small subroutines which remain in storage at all times supervising timing functions and response acquisition. In addition, this part may contain secondary-storage residen programs which manage data transfer between small in-core buffers andlarger secondary storage files. The second part of each package contains support routines used to generate storage files containing stimulus information and to dump response files to cards or printer for subsequent analysis.

This third level of control supervision is exceedingly important from the view of system design, since it is at this level that the user is interfaced to the computer and through the computer to the programmed task. The system designer is thus obliged to establish a vehicle for communication of information between CPU and user which is, on the one hand, versatile enough to allow implementation of the most esoteric programming needs and, on the other hand, is conceptually straightforward to even the most naive user. The choice and design of a programming language thus becomes a critical issue in system design, an issue worthy of careful scrutiny.

Programming languages group themselves into several categories. The first category is the traditional sequential statement approach, represented by languages such as FORTRAN and BASIC for which the conceptual organization is similar to that of an Assembly language program. The second category includes languages organized by paragraphs or "blocks," such as ALGOL or PL/1. A third category includes languages designed to operate on abstract data structures, such as list-processing languages like SNOBOL or LISP. Consideration of these language types in the context of process control requirements leads to several conclusions. Process control in real time is essentially a repetitive, sequentially organized series of events which operate on a linear data base. In psychological research, "events" consist of stimulus presentations and response acquisition serially ordered in time, while the "data base" consists of ordered vectors of stimulus and response information. As such, languages designed to manipulate complex data structures such as lists or networks are clearly inappropriate. Language features such as nesting of blocks and recursion are superfluous and tend to appear complex and confusing to the beginning programmer. Although complex arithmetic processing is rarely needed in process control, the repetitive, sequential logic flow in process programs suggests the need for language features which facilitate control of iteration and conditional branching. At first glance, then, FORTRAN or BASIC might seem to be the logical choice for a control language. But consideration of FORTRAN-like languages in the context of the previous description of interrupt, process I/O, and timer servicing tend to suggest its inadequacy. In FORTRAN, a main program or its subroutines are typically handled as units, entered with their first executable statement, and allowed to run until execution of the appropriate CALL EXIT or RETURN statement. Thus, termination of a timed delay results in transfer of control to a program's entry address, at which point the programmer must determine what section of code is to be executed next, then transfer to it. The programming of timed delays, coupled with the necessity for complex bit manipulations for process I/O, tend to burden the programmer with complex yet petty bookkeeping which has nothing to do with the actual logic of the controlled process. At best, a FORTRAN-written process program degenerates into little more than a list of subroutine calls, where the subroutines do all the work. Years of struggling with cumbersome FORTRAN process programs and the burden of laboriously teaching the system to new programmers has led us to a rejection of FORTRAN in favor of a process language of our own design. At present, a year's experience with our language, PROSS, has shown it capable of effectively eliminating most of the FORTRAN language problems, especially in the area of new programmer instruction.[3]

THE PROSS PROGRAMMING LANGUAGE

PROSS is intended to be a system-independent language, readily

Table 1
Detailed Description of an Implementation of the PROSS Language for an IBM 1800 Process Controller Used for Research in Experimental Psychology

Summary of PROSS Non-I/O Statements

	C	Integer Constant
	LAB	Statement Label, Numeric or Alphanumeric
	VAR	Subscripted or Nonsubscripted Integer Variable
	I	Nonsubscripted Integer Variable
	PROG	Program Name (an External Symbol)
	M	Nonsubscripted Formal Argument
	LO	Logical Operator (GT, LT, EQ, NE, GE, LE)
	O	Arithmetic Operator (+, —, *, /, **)
	COMMON VAR1(C), VAR2(C), etc.	Array declaration
	CONSTANT I1 = C, I2 = C, I3 = 'literal string', etc.	Constant declaration
LAB	*IF* (VAR .LO. VAR) LAB1, LAB2	Conditional branch
LAB	*IF* (VAR .LO. VAR) any statement	Conditional branch
LAB	*IF* (VAR .LO. VAR) any nonconditional statement *ELSE* any statement	Conditional branch
LAB	*GO TO* (LAB1, LAB2, . . .), I	Conditional branch
LAB	*GO TO* LAB1	Unconditional branch
LAB	*DO* LAB *WHILE* VAR .LO. VAR	Iteration control
LAB	*INCREMENT* VAR, VAR	Loop index control
LAB	*DECREMENT* VAR, VAR	Loop index control
LAB	*DELAY* VAR	Programmed delay
LAB	*DEPART*	Indefinite programmed delay
LAB	*QUEUE* PROG	Queue out-of-core coreload
LAB	*LET* VAR = arithmetic expression	Arithmetic operation
	(*LET* is optional; expression in infix as in FORTRAN)	
LAB	*FETCH* VAR	Load to accumulator
LAB	*STORE* VAR	Deposit from accumulator
LAB	*MOVE* VAR *TO* VAR1 I	Array transfer
LAB	*ZERO* VAR I	Array initialize
LAB	*CALL* PROG (M, M1, etc.)	Subroutine call
	END	
	Sample PROSS I/O Statements	
	**READFILE* filename *INTO* VAR I	Disk read
	**PRINT* VAR I	Fixed format print
	**ALLOW*	Permit subject responses
	**SEARCH* I	Slide projector operate
	**FINISH*	Finalize experimental session
	**SAMPLE* VAR I	Monitor analog input point

implemented on any system which provides a macroassembler.[4] The primary purpose of the language is to facilitate the integration of the programming elements of Supervisory Control Levels 1 and 2 into functional process-control packages. A discussion of the structure of the language is included here as an illustration of a method for organizing a complex system into an easily understood command sequence.[5]

PROSS is designed to provide a convenient means for coding any experimental procedure which is organized as a sequence of discrete events ordered in time. The language contains three statement categories, concerned respectively with declaration, processing, and I/O control. In order to facilitate learning the language, similarity to FORTRAN has been maintained wherever appropriate. The three statement categories, and the PROSS statements within each category, will be discussed in turn. (Table 1 details an implementation of the PROSS language for an IBM 1800 process controller.)[6]

Declarations

PROSS includes two statements for the declaration of data. *COMMON* statements are used to allocate storage areas used both for data disposition and for interprogram communication. Common storage is placed in specific installation-determined data buffering areas rather than within a user's program. *CONSTANT* statements are used to declare variable or array storage within a user's program and to preassign specific values to variable names. Both integer values and alphanumeric strings may be preassigned.

Processing Statements

Processing statements provide arithmetic processing and program control which may be either time-contingent or response-contingent. Arithmetic operations are expressed in arbitrarily complex, fully parenthesized notation, as in FORTRAN. The frequently used operations of incrementing or decrementing a single variable (such as the index or counter in a loop) are facilitated by *INCREMENT* and *DECREMENT* statements which replace the familiar I = I+1 construction of FORTRAN.

Delay in program execution is effected by *DELAY* and *DEPART* statements. The *DELAY* statement introduces a timed delay in program execution, and a *DEPART* statement introduces an untimed or indefinite delay. In either case, resumption of program execution may be made response-contingent through prior execution of an **ALLOW* statement (see I/O statement discussion below).

Three statements are provided to allow the direction of program control. Unconditional transfer is directed by the *GO TO* statement as in FORTRAN, ALGOL, or PL/1. Conditional transfer may be controlled by a computed *GO TO* as in FORTRAN or by an *IF* statement. Since event-contingent transfer is frequently required in process control, special emphasis has been placed on the PROSS *IF* statement, and three distinct types are provided. All *IF* statements begin with a logical expression, which is followed by (1) two labels separated by a comma, (2) any statement in the language, or (3) any two statements in the language separated by *ELSE*. Iterative program control is directed by a statement of the form "*DO* label *WHILE* logical expression." The PROSS *DO*-loop construction allows the programmer to specify both the index type (subscripted or nonsubscripted variable) and the nature of the index manipulation used for loop control. *INCREMENT* and *DECREMENT* statements are typically used to manipulate the loop index; thus, for example, iteration count might be determined by decrementing a subscripted variable by a second subscripted variable until smaller than a third subscripted variable. The flexibility which PROSS *IF* and *DO* statements provide for the programmer are to a large extent responsible for the acceptance which the language has received.

PROSS provides two statements to direct interprogram communication. *CALL* statements operate identically to their FORTRAN namesake. *QUEUE* statements are used to transfer control to a noncore resident program. *QUEUE* statements would be omitted in an implementation of PROSS for a smaller computer lacking secondary storage.

PROSS facilitates array manipulation through *MOVE* and *ZERO* statements. *MOVE* statements are used to transfer an entire array, or any portion of an array, to some other location. *ZERO* statements are used to zero an array or portion thereof. *MOVE* and *ZERO* are provided

Table 2
Sample PROSS Program

The following program in PROSS performs the same function as the FORTRAN program described by Restle and Brown (1969) for a computer-controlled laboratory using a single Kodak random-access slide projector and four six-button S response boxes. The comments in parentheses are for clarification and are not part of the PROSS language.

```
*COMMON BLOCK 2
*LIST SOURCE PROGRAM
*PROCEDURE SAMPL
      COMMON NMAX, TIME1, TIME2, TIME3, TIME4, ISLID(50), JRESP(50)
      CONSTANT N = 1
      DO LOOP1 WHILE N .LT. NMAX
     *OPEN                               (Open shutter on slide projector)
     *ALLOW                              (Ready to accept response interrupts)
      DELAY TIME1
     *CLOSE                              (Close shutter)
     *IGNOR                              (Stop accepting response interrupts)
     *SEARCH ISLID(N)                    (Initiate search for given slide in
                                          random-access slide projector)
     *RESPONSES INTO JRESP(N)            (Transfer responses into user array)
      DELAY TIME2
     *CUELITE ON JRESP(N)                (Turn on feedback light)
      DELAY TIME3
     *CUELITE OFF JRESP(N)               (Turn off feedback light)
      DELAY TIME4
      INCREMENT N
LOOP1 CONTINUE
      QUEUE PROGM
      END
```

*Note that the *COMMON BLOCK control statement, in conjunction with the COMMON statement, specifies the communication linkage from the program into installation-defined core data buffers. SAMPL and PROGM are arbitrary external symbols (program names). The variables in COMMON, excluding the JRESP array, are initialized by program PROGM, which is automatically called on experimental session initialization as the result of compiler-generated coding.*

primarily to facilitate transfer and manipulation of blocks of stimulus or response information such as character strings used in CRT display devices.

Input/Output

PROSS uses a unique and highly general input/output facility. As discussed above, installations will typically develop a series of special-purpose input/output control subroutines, each designed to control or monitor some specific process. The appropriate calls to such subroutines can easily be generated using the facilities of a macro assembler, which are now available as standard supported software for most small- to medium-sized computers. The PROSS compiler generates calls to I/O subroutines through direct transliteration of the source statement, thus allowing each installation to establish its own macro library which is accessible to the PROSS programmer without compiler modification. PROSS I/O statements contain one or more words, separated by blanks, commas, or parentheses. The compiler interprets the first word as a macro name and interprets all remaining words as parameters to that call. The resulting macro call is then expanded by the assembler into either the appropriate subroutine call or into the actual code necessary to control the I/O operation. Effectively, the programming power normally provided by a macro assembler is brought into the reach of the beginning programmer instead of remaining accessible to only the sophisticated machine-language programmer.

The general structure of a PROSS program (see Table 2) resembles a FORTRAN program in external appearance. Each program begins with several compiler control statements, indicating program type (e.g., program vs subroutine) and data communication area. The communication area is one of several defined by the installation as being accessible to process programs and, as such, is analogous to FORTRAN labeled *COMMON*. The control statements are followed by conventional program declaration statements which allow the programmer to define variables and arrays. As the example in Table 2 suggests, executable statements consist of a temporally ordered sequence of events spaced appropriately in time by *DELAY* statements. The exit and reentry coding which is required for effective time sharing in an interrupt-oriented time-sharing system is automatically inserted by the compiler. The user thus creates his program with the idea that he is in control of the entire system while his program is in execution, while, in actuality, his and several other programs may be simultaneously in execution, each entered by the resident monitor upon recognition of the appropriate interrupts.

SUMMARY

I have attempted to outline the general structure of an operating system designed to provide user-oriented process control for a computer installation used for behavioral science research. Three conceptually distinct levels of control supervision were discussed, and special emphasis was placed on the interface between the user and the system at the highest level. A process-control language designed specifically to facilitate the user-system interface was described in general and is specified in detail in Tables 1 and 2.

REFERENCES

McLEAN, R. S. PSYCHOL: A computer language for experimentation. Behavioral Research Methods & Instrumentation, 1969, 1, 323-328.

RESTLE, F., & BROWN, T. V. A computer running several psychological laboratories. Behavioral Research Methods & Instrumentation, 1969, 1, 312-317.

NOTES

1. A more complex communication queue might include additional words or bits for coding execution priority. The priority would be set by the program which inserted the program name or address into the queue (typically an ISS). The system loop would then be required to interrogate the entire queue each time through the loop and would transfer to the routine having the higher priority, regardless of its position in the queue.

2. On small computers having only a single interrupt input, the system is effectively masked by simply disabling the interrupt system. On larger multiple interrupt systems, hardware instructions are typically provided to set or reset bits in an interrupt mask register, thus effectively disabling only selected interrupts. In the case of the latter (multiple interrupt) system, the ISS need only mask interrupts which transfer to itself; other interrupts can be left active without fear of interference.

3. The FORTRAN system in use in our laboratory is IBM 1800 FORTRAN II. It should be noted that a FORTRAN IV system allowing multiple entry points to a process program would be far less cumbersome.

4. Our compiler compiles PROSS source statements to macro calls hosted in IBM 1800 Assembly language. The macro output approach greatly simplifies the programming of the compiler and allows modification of object code by simply altering the PROSS macro library.

5. It should be noted that PROSS is superficially similar to PSYCHOL (McLean, 1969). PROSS has at least two advantages over PSYCHOL: first, its ease of implementation, and, secondly, its structural simplicity. We have taught naive programmers the PROSS language with less than 2 weeks of informal instruction.

6. A subset of this system has been implemented on a 4K PDP/8E with no secondary storage. The resident monitor-delay time portion of the system has been implemented on a 4K-byte Micro 810.