Behavior Research Methods, Instruments, & Computers
1985, Vol. 17 (5), 537-545

Microcomputers and conditioning research

PETER D BALSAM, JAMES DEICH, KEVIN O'CONNOR, and ROBERT SCOPATZ
Barnard College, Columbia University, New York, New York

The use of microcomputers in conditioning research is described and analyzed. The use of a finite state description of experimental procedures is advocated and the desirable characteristics for a real-time control language to define states and their transitions are discussed. Several programming languages are evaluated against these criteria. Various methods for interfacing microcomputers with the external environment for control and data acquisition in conditioning research also are discussed.

In this paper, we discuss the role of computers in operant and Pavlovian conditioning research. First, we provide information that should be helpful in designing and implementing microcomputer systems for use in conditioning research. We describe the important considerations in choosing a system and some solutions to common problems involved in implementing these systems. Second, we hope that our description of conditioning research and the concomitant use of computers will influence computer and software developers in their design of future systems. We advocate a way of describing conditioning research that is familiar to computer scientists and engineers. Finally, we are intrigued by the possibility of understanding the relationship between the formal structure of a research problem and the formal structure of the physical system that is used to analyze the problem. The relation is bidirectional. The research problem influences the computer hardware and software used to study the problem, and the computer system no doubt influences the way in which the research problem is studied and conceptualized. No model is offered of the interaction between a research problem and instrumentation; however, by sharply defining the problems of experimental control and describing the solutions to these problems, part of a data base is formed that may eventually generate such a theory.

PAVLOVIAN AND OPERANT CONDITIONING

Conditioning experiments are concerned with the effects of contingent changes in environmental conditions on behavior. Changes are sometimes contingent on the presence of particular environmental events or on the occurrence of particular behaviors. In Pavlovian conditioning experiments, the presentation of one stimulus is conditional on the presence of another stimulus. In Pavlov's prototypical experiment, the presentation of a bell for a relatively brief period of time is followed by the introduction of meat powder into the mouth of a dog. After a number of such pairings the sound of the bell alone is sufficient to elicit a copious flow of saliva. The presentation of the meat powder [unconditioned stimulus (US)] is contingent on the presentation of the bell [conditioned stimulus (CS)]. Many stimuli can serve as conditioned and unconditioned stimuli, and behavior in many response systems is modified by these contingencies. Defensive reactions such as freezing, blinking, and withdrawal evoked by shock USs, taste aversion learning with food CSs and illness-inducing USs, conditioned immune reactions, and conditioned motor and physiological reactions in anticipation of food presentation have served to produce rich theoretical and practical analyses.

In the prototypical operant conditioning experiment, the presentation of the US is contingent on behavior. The presentation of a food pellet to a hungry rat when it presses a lever has served as a common experimental preparation for studying how the consequences of behavior influence their future occurrence. Consequences, such as the food pellet, that increase the frequency of the behavior that produces them are referred to as reinforcers, whereas consequences that decrease the behavior on which they are contingent are referred to as punishers. Behavior may be reinforced by the presentation of a stimulus. For example, educators try to increase the frequency of creative thinking by praising students and providing high academic grades contingent on that performance. Behavior also may be reinforced by the termination or avoidance of a stimulus. For example, we escape from an unpleasant conversation or avoid the conversation altogether by not entering a room where someone unpleasant is located. In all of these cases, the change in an environmental condition is contingent on the occurrence of behavior, and the consequences of the behavior modulate the probability of that response in the future.

Although we have introduced operant and Pavlovian contingencies as though they were separate, environmental changes are contingent on both stimulus and response conditions in many experimental arrangements. In a discrimination procedure, for example, a response might only produce reinforcement in the presence of one stimulus but not in the presence of other stimuli. In the previous example of a rat's leverpressing, reinforcers might only be presented in the presence of a red light, but responses might not be reinforced when a green light is presented.

Address correspondence to P. D Balsam, Department of Psychology, Barnard College, Columbia University, New York, NY 10027.

The rat soon responds vigorously in the presence of the red light but makes few, if any, leverpresses in the presence of the green light. As this example illustrates, environmental changes in conditioning experiments often depend on both antecedent conditions and current behavior.

There are, then, three basic ingredients to conditioning experiments: antecedent stimuli that are correlated with prevailing reinforcement contingencies; consequent stimuli that are usually motivationally significant; and rules that specify the transitions from antecedent to consequent conditions. Transitions in Pavlovian experiments depend on the passage of time, and transitions in operant experiments depend on the occurrence of particular behavior, though they may also depend on the passage of time. The conditioning experiment, therefore, is summarized in a description of two states plus rules for the transition between them.

DESCRIPTION OF CONDITIONING EXPERIMENTS AS SEQUENTIAL SYSTEMS

Fifteen years ago, Snapper and his colleagues (Snapper & Kadden, 1970; Snapper, Knapp, & Kushner, 1970) proposed a notational system for the description of reinforcement schedules based on the theory of finite state automata (Mealy, 1955; Moore, 1956). Their purpose was to describe conditioning experiments in terms of sequential transitions between a finite set of environmental states. At that time, they emphasized the importance of such a system for accurate communication between researchers studying schedules of reinforcement. Snapper et al. (1970) mentioned that this system also might be helpful because it can be used to describe any sequential system and, therefore, might facilitate communication between workers in many different fields. Fifteen years later, the arguments for using this system of notation are even more compelling. The theory of finite state automata has received considerable formal mathematical study and is one of the basic foundations of modern computer science. It is associated with the language that computer scientists use to describe the problems they study. If psychologists can frame experimental control problems in a language familiar to the computer scientist and engineer, they will facilitate the design of hardware and software systems that meet the needs of psychologists. Also, as computers are becoming the dominant means of instrumenting experiments in psychology, a description of experimental procedures that is unambiguously translated into software is highly desirable. In the following section, we borrow examples from Snapper and his colleagues to illustrate how conditioning experiments are described as sequential systems.

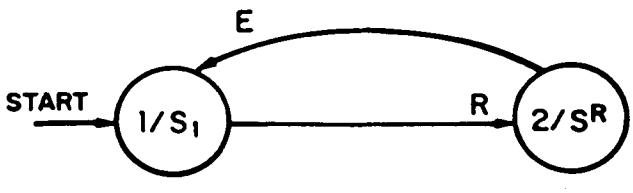

Figure 1. State diagram for a continuous reinforcement schedule.

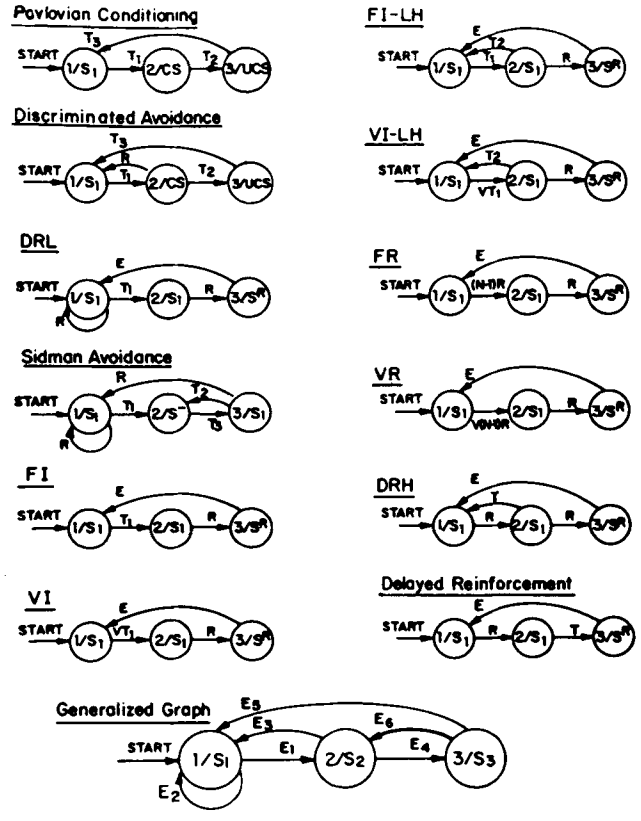

Figure 2. State diagrams for commonly studied reinforcement contingencies. Generalized state graph is shown in the lower portion of the figure (after Snapper, Knapp, & Kushner, 1970).

The stimuli and contingencies associated with a particular experiment can be divided logically into states. The present state of the system and current input uniquely determine the next state of the system (i.e., a Markov process). Our example procedure, in which the rat's leverpresses were reinforced with food, is illustrated in the state graph shown in Figure 1. There are two states. State 1 is associated with the background conditions of the experiment. The output of State 1 usually consists of houselight, white noise, and all other cues that define the experimental context. If a leverpress occurs, then a transition occurs from State 1 to State 2. State 2 is the state associated with the reinforcing stimulus (S^R). After a period of time (access to food) or some other event (consumption of a food pellet), the system returns to State 1. Any sequential procedure can be specified completely by a state graph such as this one. Figure 2 shows state graphs for most of the commonly studied conditioning procedures. The diagram in the upper left corner depicts a standard Pavlovian conditioning experiment. The intertrial interval conditions are in effect during State 1. After T_1 sec-

onds, a transition to State 2, the CS state, occurs. After T_2 seconds, there is a transition to the US state, State 3, for T_3 seconds, followed by a transition back to State 1 for the next intertrial interval. More complicated procedures and contingencies can be depicted in economical fashion by these state graphs. Even without prior familiarity with the schedules depicted in Figure 2, the reader will find that the experimental arrangements are presented without ambiguity. All of the protocols shown in Figure 2 can be derived from the generalized state graph shown at the bottom of the figure. Any computer system that is capable of implementing the generalized state graph is sufficient for implementing all of the procedures used for the study of conditioning. Such a system must consist of a language for defining states and for specifying the rules of transition between states. The hardware of the system must include ways of instantiating the output associated with each state and a means for detecting inputs which trigger state transitions.

REAL-TIME CONTROL LANGUAGES

In this section, we discuss whether some programming languages are better than others for real-time control. The desirable characteristics of a real-time language (many of which are desirable characteristics of any computer language) are described, and the manner in which some languages/language families meet these criteria is evaluated.

Although we do not evaluate all languages for defining states and transitions, the criteria we use are explicit and can be applied to an evaluation of any language.

The following items are desirable characteristics of a real-time control language:

(1) The language should be characterized by *clarity and economy of expression* for the real-time task involved. The language should be easy to read and feature suggestive names for its operations. For example, "CS-ON" is a more desirable command to turn on the conditioned stimulus than are "ON1" or "POKE(1645,1)." Keywords should be short but clear. Pascal, for example, uses the notation ":=" for the assignment statement rather than the usual "=." Although this may help a beginning programmer note the difference between an assignment statement and an equality test (e.g., "if A = B then," vs. "A := 3;"), it is an annoyance to an experienced programmer.

(2) Programs in the language should be *easy to modify*. Unlike many industrial control or commercial programs, progams that run psychological experiments are frequently subject to modification.

(3) Programs written in the language should be *easy to debug*. If a long compilation step intervenes between successive tests of the program, much time will be wasted. When an error is encountered, the language system should provide a specific and informative message and pinpoint the error in the program's source code.

(4) The language should *encourage the development of a set of frequently used subroutines* for common tasks. A modular type of programming structure allows for ease of subsequent application.

(5) The language should be *structured*. That is, it should avoid GOTOs and instead employ structured control statements (e.g., while, repeat, do). Programs that employ unusually nested GOTO statements are difficult to read and debug, because state definitions and transitions may be difficult to isolate. The language should encourage top-down development of programs. A mainline routine that serves as an "outline" of the program should come first, followed by subroutines that handle progressively more specific tasks.

(6) The language should *interact well with the operating systems* of the computer. The language should link easily to assembly/machine language routines, including those of the operating system, for purposes which involve critical timing.

(7) The language should be *portable* and relatively easy to transfer to another computer. This reduces duplication of effort and decreases the chance that the researcher will be stuck with an obsolete computer in order to run software that was developed at large personal or financial cost.

(8) The language should *lend itself to a parallel processing environment*. Powerful multi-CPU machines at relatively attractive prices should be available within the next decade. Because complicated real-time tasks place heavy speed demands on a computer, this characteristic will be increasingly important.

(9) The language should provide facilities for *easy handling of external input/output* and execute quickly enough to perform the many needed I/O operations.

Some of the more common computer languages have been used to control laboratory equipment and accumulate data in psychology experiments. Below, we describe the advantages and disadvantages of implementing experimental control systems in different languages.

BASIC

Interpretive BASICs are furnished with almost all microcomputers today. However, the computer science establishment increasingly rejects this language even for tutorial purposes.

Advantages. BASIC and BASIC programmers are widely available. Because BASIC is usually furnished as an interpreter, debugging is facilitated. That is, variables can be altered from the keyboard, and single statements can be modified at any time. Execution can be started and ended at arbitrary points. BASIC typically interacts well with the host operating system.

Disadvantages. In interpreted forms, BASIC is very slow. It is not well standardized and is not easily portable. Structured control statements have been added to many of these systems, but they are not standard. Subroutines are referenced by line number rather than by a suggestive name (e.g., "GOSUB 100" rather than "REINFORCE"). Furthermore, the subroutines as referenced do not bind local variables. These facts make the

development of useful libraries of subroutines difficult. BASIC does not lend itself to multiprocessor environments. It lacks clarity of expression for real-time tasks: Many PEEKs, POKEs, and CALLs are likely to be present. Finding the appropriate location for and passing data to machine language routines is often difficult.

With interpreters, it is often tempting to remove comments to make the program run faster (or, for that matter, to not add them in the first place). Some of the problems associated with this language become obvious when a user returns to a 2-year-old BASIC program and tries to evaluate what it does and how.

C Language

The C language was used to develop the UNIX operating system, which is well on its way to becoming a standard on more powerful micro- and minicomputers. It is well standardized and earns the current high mark for portability. It has not been widely used by psychologists for real-time programming.

Advantages. C is a structured language that encourages the development of programs in a top-down fashion. Because it uses symbolic names for subroutines and can use symbolic names for constants, it is easy to develop programs that suggest the subroutines' function, even with limited comments. The development of a large program as a set of easy-to-debug subroutines is therefore encouraged. C usually interacts well with the host operating system, and it is easy to integrate machine language subroutines with the program. Some C procedures for I/O are so general that they will work with multiple operating systems.

Many high-quality C compilers produce fast machine code and are clearly fast enough for many applications that involve the analysis of converted analog data. C contains "pointer" variable types that produce faster code than the indexed array structures that are used in BASIC.

Disadvantages. The C language presumes more knowledge of computing than do simpler languages. It usually is not taught as a first programming language. Therefore, even though many C programmers are available, they may be at a salary/status level that does not make them available to the experimental psychologist.

Although programs tend to be developed in a top-down fashion and in small chunks for easy location of problems, the debugging process itself can be tedious with the wrong tools, because most C systems are implemented as compilers. Multiple cycles of locating an error and recompiling are tiresome. The problem should be alleviated by the recent development of C interpreters for the debugging phase.

C, in its basic form, does not lend itself to multi-CPU environments or to very small systems (e.g., the widely used Apple II and TRS-80 systems). One 700-line C program written for an Apple IIe took 27 min to compile and link.

Pascal

Pascal was introduced by Niklaus Wirth in the early 1970s as a response to the "unstructured" languages that were then dominant (mainly FORTRAN). It has become a standard educational language, but has seen little commercial application. Its unpopularity with professional programmers seems to stem mainly from its rigid typing of variables and its uneconomical expressions. Wirth has recently introduced a successor language, MODULA II, which answers some of these criticisms.

The speed of Pascal systems varies enough that the language as a whole is difficult to characterize. Some fully compiled systems are very fast; other systems are only partially compiled and run much more slowly.

Advantages. Like C, Pascal is structured and uses symbolic names for subroutines. Named subroutines and variable binding facilitate the development of subroutine libraries in Pascal. The wide use of Pascal as an instructional language provides a good source of programmers. Extensions are being added to Pascal's successor, MODULA II, that may make it appropriate in a multi-CPU system.

Disadvantages. Pascal was designed for educational clarity, not for convenience. Its expression is uneconomical, although exacting. The programmer types "Begin," "End," ":=," "array of," "procedure," and a number of other keywords and symbols a tiring number of times. Routines must be declared and defined before they are referenced; this does not encourage the development of programs in a top-down fashion.

FORTH

The FORTH language was developed by Charles Moore in the early 1970s to control observatory equipment and to gather data. Unlike the languages mentioned previously, the initial impetus for the development of this language was real-time control and data gathering.

FORTH uses reverse-polish notation [1 2 + 3 / rather than (1+2)/3]. This notation is not used simply for calculation, but also to control execution, as in IF statements. For example, "IF A > 3 THEN" becomes "A @ C @ > IF."

The language uses a push-down stack to pass all parameters to subroutines (called words in FORTH). Unlike most other languages, which use stacks internally, FORTH gives the programmer direct control of this stack. For example, the top-stack elements can be swapped or duplicated. This set of unusual features has made FORTH one of the most distinctive languages in computer science.

In recent years, there has been an increasing, but limited, use of FORTH, especially for real-time applications. The Rockwell Corp. is now marketing a chip that has both a 6502 microprocessor and FORTH in read-only memory, which is likely to encourage the growth of this language.

Advantages. FORTH is structured and fast. Expression is extremely economical. Fully compiled versions of FORTH have been developed that are even faster than the standard version (e.g., FIRST for the Apple II).

FORTH subroutines can be named suggestively so that letter reading is facilitated. It is easy to create a useful set of subroutines. FORTH encourages programs to be broken down into extremely small segments and enables the programmer to test these individually from the keyboard. The arguments are simply pushed onto the stack and the name of the routine is typed in. This is a valuable feature that other languages would do well to add. This feature speeds debugging, as does FORTH's fast compilation.

FORTH uses memory very efficiently, and works in systems as small as 16K bytes. Moderate to complex applications actually take up less space in a FORTH system than in the equivalent assembly language program. Most FORTH systems contain integral assemblers. Linkage to assembly language routines, therefore, usually is easy.

FORTH standards are set every few years by a committee; code that meets these standards is very portable. However, many implementations of FORTH add new functions that tempt the programmer to depart from the standards with a concomitant loss of portability.

With correct coding, FORTH is the fastest of the languages listed here. It is well suited to the rapid acquisitions and treatment of data from external sources. FORTH systems for parallel processing environments already have been developed. This should facilitate the use of this language on future multi-CPU computers.

Disadvantages. The reverse-polish notation that FORTH employs is hard to read and often makes it difficult to note the structure of the program. The language may intimidate people who have learned more traditional languages. These problems are aggravated by the use of "editing screens" that are of a fixed size, usually 15 lines by 66 columns. Many programs seem to begin with spacing that shows their control structure, but as they undergo successive modification and extension, this space is squeezed out because the screen size is fixed, leaving a program that runs on without obvious structure.

As in Pascal, subroutines must be declared and defined before they are used, so that top-down programming is not encouraged. While debugging is facilitated by the interactive nature of the language and the fast compilation, error checking is limited due to the nature of the language. As is the case with assembly language, some of the errors can cause system crashes and other unusual behavior.

Educational materials describing FORTH are increasingly available, but FORTH programmers are difficult to find.

SKED and Other State-Transition Languages

The development of SKED and the language's use in psychology are due almost entirely to the efforts of Arthur Snapper. This language was developed with the direct aim of controlling operant behavior experiments in a real-time setting. It makes use of state notation which has become a standard in computer science and engineering for the description of systems which feature specifiable states of affairs and operations that cause movement between them.

The use of state notation emphasizes the orientation of programs to changes based on time and real-world events in a way that no other notation can. For example, '6"→S2' as a program line makes clear that a transition is to occur in 6 sec in a way that the statement, "Repeat readclock (A) until A=6;", does not. The superiority of state notation in cases of multiple events and times is clear. SKED sets a standard in clarity for real-time expression that supersedes that of a language that simply uses suggestive names.

Advantages. SKED features a structure that maps very well onto real-time tasks. Each program can have multiple sets of connected states, and each SKED system can run multiple programs. The value of multiple sets of connected states for complex tasks cannot be overestimated. SKED connects these separate "state sets" with a primitive intercommunication device (the Z pulse). Among the languages considered, SKED certainly shows the most promise for future multiple-CPU systems. A system that assigns one CPU per state set, with other CPUs performing background tasks (I/O, computation, etc.), should be powerful indeed.

SKED encourages top-down development of programs. Most programs have some sort of master state set that controls the function of all others. SKED features extreme economy of expression for real-time tasks due to its use of state notation.

Disadvantages. Historically, the biggest problem with SKED is that it was only available for the obsolescent DEC PDP-8 series of computers. These computers were a design/price breakthrough in the mid-1960s, but have relatively poor performance and memory usage by today's standards.

SKED is now available for PDP (LSI) 11 computers. However, the combined computer, software, and interface cost will probably exceed the budget of many researchers. More limited state-transition systems are available for the TRS 80, Apple II, and soon the IBM-PC systems. Portability among these systems is poor overall.

SKED generally is only partially compiled and is not fast enough to be easily used for the most demanding analog-sampling tasks. SKED's mathematical capabilities are limited. Furthermore, frequent use of mathematics, especially in conjunction with loops, damages the system's performance and can cause timing errors.

Although SKED contains a useful set of subroutines, adding user subroutines to create a library is not easy. The ease of linking a SKED program to a machine language program varies from implementation to implementation, but is almost always more difficult than linking a machine language program to a C or FORTH program.

Members of the general computer science establishment generally are not familiar with this language. For most bright students, however, the language is not difficult to learn.

TIMING METHODS

State transition in conditioning experiments are often time dependent. Timing is therefore one of the important tasks that an experimental control system must perform. The major factors to consider when choosing a timing method are: (1) cost, (2) ease of use, (3) operations to be performed while timing, and (4) resolution. Two common methods used to achieve accurate timing of events within an experiment are loops and clocks.

Timing Loops

Each statement in a program requires approximately the same amount of time for each execution. Counting the number of times a statement is executed can therefore be translated into a time value. In BASIC, for example, the FOR/NEXT loop is an integer counter usually set to start with an initial value of 0 or 1 and increment by +1 with each pass of the "loop." The looping stops when the terminal value is reached. Timing is achieved by varying the terminal value, and thus changing the amount of time it takes for the loop to complete the required number of passes.

Calibration of loops requires external timing. The most accurate means of timing is either a clock that is started and stopped by the computer, or a loop that runs in isolation from the main program and counts the number of passes in a given amount of time.

Often it is necessary to check the status of an input line while timing, for instance to collect subjects' responses and latencies. Data collection within a timing loop can cause a large increase in the time it takes to complete the loop, thus making it necessary to calibrate such loops separately. A more serious problem arises because of the need to use conditional (IF... THEN) statements to process the incoming data. The code takes different amounts of time to execute depending on the nature of the different conditionally executed statements. The basic timing loop involves another problem for accurate calibration. The first time the loop is executed, the counting variable is likely to be anywhere but at the top of the "variable stack" (a list of locations in memory to which variables and their current values are assigned). Because the variable stack is searched sequentially for the variable needed, the first execution of the loop requires a longer search than the second through final executions of the same line. It is advisable, therefore, to force any variable to be used in a loop to the top of the stack by calling it before entering the loop.

Interpreted BASIC runs very slowly. Because of this, it is often desirable to compile the BASIC program. Compilation causes all of the statements of a program to run faster; this includes FOR/NEXT loops. Compilation requires a recalibration of the loops and a recompilation in order to test the calibration; this compilation often is a lengthy process.

One common solution to the above problems is to write the timing section of the program in Assembly language. A program written in Assembly language is able to perform a complex function at least several hundred times faster than one written in BASIC, and can execute a simple function even faster. Whereas each BASIC statement must be interpreted before execution, which takes on the order of 6-8 msec, assembled code can be executed in a matter of a few microseconds. This speed makes Assembly language very useful for precise timing or control, particularly when at least millisecond accuracy is required. Because the different statement types are executed at very nearly the same fast rate (the differences in execution time are in the range of microseconds), adding statements within the loop (to check for responses in our example) does not lower the accuracy of the timing. An example of an Assembly language timing routine is provided in the Appendix.

A second advantage of calling Assembly language routines from a main program is that the routines are independent of changes in the main program. The main program may thus be compiled to run at a greater speed without making it necessary to recalibrate the timing loops.

Clocks

Many computer operating systems hold the current time (and date) in memory. Because the system must update these memory locations with the passage of time, they may be used to time events. For example, in the TRS-80, a 25-msec "heartbeat" timer is provided along with timers for seconds, minutes, hours, days, and months. To use the "heartbeat" and other timers in memory requires knowledge of the precise memory locations where timers reside. Usually this knowledge may be gained from the operating system manual or from a user's group.

The actual resolution of this timing method is not equal to the nominal time between "heartbeats," as one might expect (25 msec in the TRS-80), but it is, conservatively, approximately half of that value (50 msec). The first reason is that small fluctuations in line current and increases in system demand cause some heartbeats to be longer than others. The variability is unnoticeable in most applications, but may be important if highly accurate timing is desired. A more serious limitation is that, in most cases, the clock is accessed using BASIC programs. The execution speed of BASIC statements is slow enough that a loop designed to PEEK at the timer locations, check for responses, and record latencies will "skip beats" from the 25-msec heartbeat timer. This is the reason that in most practical applications using the system clock is not advised for timing where resolution below .1 sec is desired.

One method is entirely independent of system demands and from the number of statements executed while timing: an external hardware clock that can be read by the computer. Clocks are available with any desired degree of accuracy. Again, the resolution of the timer may be

limited by the rate of program execution and the efficiency of the program rather than by the clock speed, per se.

We offer two final notes about timing: First, it is advisable to design programs to do as little as possible while timing. The more statements included within a timing section of a program, the lower the resolution of timing. In many systems, the most serious effects on timing accuracy are caused by disk access operations during a timing loop: Calls to service a disk take more time than any other single statement, and the read or write operation takes significant amounts of time, depending on how far the head is sitting from the location it must finally reach. Sometimes the head will be in the correct position, and control will return to the program within a few milliseconds; other times the user may wait a second or more before regaining control. In some systems, heartbeats are inhibited during disk I/O.

Another serious consideration is that there is a tradeoff between resolution in "catching" input and resolution in timing. When a program is checking response inputs, it is not checking the clock, and vice versa. While the program is checking the clock, inputs may be lost. While inputs are being processed, time may pass without the clock being checked or perhaps even updated (depending on the timing method employed). In real-time control applications, one must achieve a premeditated balance of these factors.

EXTERNAL INPUT/OUTPUT

Given that the selected language is adequate for specifying states and state transitions, the system must interface with the experimental environment. Specifically, in order to use a microprocessor to perform conditioning experiments, the system must have the capability of controlling the presentation and timing of events in the external environment and must be able to sense and record behavior. The form that the solution takes is a good example of how the form of a problem dictates the computer hardware and software. Most conditioning experiments involve the presentation of a constant stimulus over repeated trials and repeated recording of a response specified in advance of an experiment. The experimenter need only control whether the stimuli associated with a particular state are present or absent, sense a response-produced switch closure, and change states at appropriate times. A digital I/O interface performs this function. In our TRS-80-controlled experiments, a standard digital interface manufactured by Alpha Products, Inc. (Interfacer-80) is employed. These interfacers have eight input and eight output lines and plug into the TRS-80 parallel I/O port. The TRS-80 is capable of controlling up to eight of these units. Each of the eight output lines switches a relay which allows for easy switching of external power sources. The eight input lines are in series with opto-isolators which prevent electrical interactions between the external inputs and the computer. External inputs activate the opto-isolators, which in turn are detected by the computer system as a change in the state of one of the bits of an input port. The adequacy of a digital input rests on the assumption that the behavior being studied can be adequately characterized by a binary variable (response/no-response). There are occasions in which there is interest in the dynamic properties of behavior or in the form of the response. Moment-to-moment changes in response amplitude and sequential changes in response form are sometimes of theoretical importance. In these situations, a researcher might wish to detect changes in some continuous property of behavior. The only solution to this problem is to define different input states corresponding to different aspects of response form and to construct a representation of the continuous changes as changes in states over time. There are two ways to achieve a representation of a continuous process. First, a user can rely on external equipment to produce binary-state changes that correspond to different values of the continuous variable. For example, different inputs can change state as a function of the amplitude of an input, or spatial information can be represented in a binary code as in a digitizing camera. Alternatively, one can employ an analog input device which maps continuous changes in input into an arbitrary but continuous scale. In either case, the investigator must be sure to sample the input device at a fast enough rate to be sure of detecting the changes in response form.

A similar consideration arises in the control of external outputs. Most conditioning experiments employ unchanging stimulus conditions. However, there is growing interest in the manner in which rhythmic and dynamically changing stimulus conditions affect learning (Gibbon & Allen, 1984; Hulse, Cynx, & Humpal, 1985). Rapid changes in the dynamic properties of stimuli require analog output. One cannot help but wonder whether the recent interest in the dynamic properties of antecedent stimuli was nourished by improved computer technology for producing analog output, as has surely been the case in the study of speech perception.

Buffering

Another important aspect of implementing external input/output functions that is influenced by the speed of the system is insuring that all inputs are sensed by the computer, but counted only once. In the I/O interfaces that we use, the standard configuration consists of eight input lines that change state only while an input activates the opto-isolator. If the inputs are of brief duration, inputs must be sampled at a rate faster than the duration of the briefest input. On the other hand, if the sampling rate is too high, then a single input would be detected multiple times. There are several ways to prevent this problem. A simple modification can be made to an interface, so that an input line is switched at the onset of an

input and remains in that condition until the input has been read by the computer (this is a buffered input). It is possible to have several levels of buffering.

Interrupting

An alternative to buffering involves the use of interrupts. Many computer systems are designed so that processing tasks can be sequenced in a hierarchical structure. Tasks are performed according to a predetermined priority. It is possible to design a system so that processing of external inputs is a high-priority task. If input processing is of higher priority than processing tasks currently taking place, the current processing is interrupted until the external input is processed. Both the buffering and interrupt systems minimize the chances of missing incoming information.

Counting Input

Each input must be counted once and only once. If inputs are of a known fixed duration, then software can be written so that sampling of inputs only occurs a fixed duration after the preceding input. Alternatively, although inputs might be sensed many times, counters might be incremented only for inputs separated by a minimum interval. If inputs are of variable duration, then a different technique is required to insure that single inputs are not counted more than once. Counters must be incremented at either the leading or trailing edge of an input. Again, it is clear that speed is important, and that not only does the speed of incoming information set speed requirements for a system, but the nature of the processing task during input also puts constraints on the system characteristics required to implement the experiment.

STORAGE AND ANALYSIS OF DATA

There are two common methods for organizing data on disk. The first method is to write the data in one long string. This method requires the least possible amount of space, but it may generate difficulties at the data analysis stage. First, a single missing data item will cause all subsequent items to be read into the wrong variable in the analysis (trial types will be read as latencies, etc.). It is very hard to visually identify missing data in a streamed output. In addition, if data are to be transported to a mainframe computer for analysis, the record length of a data stream may be too long for the mainframe's editor, or input lines and the statistical package may not accept streamed data in the first place.

A second method which avoids the obvious problems of streamed data, but at a large expense in storage space, is to output data by records so that each line in the data file corresponds to a trial number in the experimental session. Data on a line are usually separated by either a single space or a comma. Such a file is easy to check for missing or out-of-range values. It can be easily transferred to a mainframe computer, and most statistical packages accept data delimited by spaces and/or commas as input. While this method is good, it is not the easiest and most generally acceptable data format for analysis programs.

Analysis programs, especially spreadsheet programs, are usually designed for a fixed-field format organization in the data. In recent versions of most statistical packages, free-field data formats are supported, but can still be of limited usefulness for certain analyses requiring the skipping of data items. Spreadsheet programs demand data in columns in order to perform any calculation and plotting. Because of this requirement, and because it takes only a bit more space on the disk than delimited data (spaces or commas), it is advisable to output the data in fields of columns. For example, the first three columns can be set aside for the trial number, the second three columns for the trial type, the third set of columns for responses, and the last set for latencies. Within each field, the data should be right justified. An additional benefit of columnar data is that it is far easier to check than any other type of organization. Missing data corresponds to blank fields; out-of-range data can be found either by visually scanning each column for large or small values or by using a standard editor to perform the search quickly.

EXTRAEXPERIMENTAL CONSIDERATIONS

Most microcomputers can perform the tasks that we have outlined. The costs of microcomputers range from less than $100 to several thousands of dollars. In general, the more expensive machines can do more things faster, but for many applications neither the size nor the speed is a necesary requirement. Similarly, I/O interfaces can cost from less than $100 to several thousands of dollars (Clower & Calabraro, 1983; Perera, 1980). Of course, budgets play a large role in the choice of systems.

One aspect of system choice that should be emphasized is the ease with which an application can be implemented and the ease of maintaining hardware (Balsam, Fifer, Sacks, & Silver, 1984). The availability of programmers and/or the "friendliness" of the software are initially important considerations. If programmming can be done in a commonly taught language, then assistance should be readily available. Spreadsheet programs with integrated graphics packages are extremely attractive pieces of software for integrating with data acquisition and storage aspects of experimental systems.

Finally, there is often an advantage to selecting the same system used by colleagues. The sharing of software and expertise is valuable and should be an important consideration.

REFERENCES

BALSAM, P. D., FIFER, W., SACKS, S., & SILVER, R. (1984). Microcomputers in psychology laboratory courses. *Behavior Research Methods, Instrumentation, & Computers*, **16**, 150-152.

CLOWER, P., & CALABRARO, P. (1983, October). Seeking a cheap output? *Microcomputing*, pp. 82-84.

GIBBON, J., & ALLEN, L. (1984). *Timing and time perception*. New York: Annals of the New York Academy of Sciences.

HULSE, S., CYNX, J., & HUMPAL, J. (1985). Pitch context and pitch discrimination in birds. In P. Balsam & A. Tomie (Eds.), *Context and learning*. Hillsdale, NJ: Erlbaum.

MEALY, G. H. (1955). A method for synthesizing sequential circuits. *Bell System Technical Journal*, **34**, 1045-1079.

MOORE, E. F. (1956). Gedanken experiments on sequential machines. In C. E. Shannon & J. McCarthy (Eds.), *Automata Studies*. Princeton: Princeton University Press.

PERERA, T. P. (1980). Universal one- and two-component interfacing techniques. *Behavior Research Methods & Instrumentation*, **12**, 236-237.

SNAPPER, A. G., & KADDEN, R. M. (1970). Timesharing in a small computer based on use of a behavioral notation system. In B. Weiss (Ed.), *Digital computers in the behavioral laboratory*. New York: Appleton-Century-Crofts.

SNAPPER, A. G., KNAPP, J. Z., & KUSHNER, H. K. (1970). Mathematical description of schedules of reinforcement. In W. N. Schoenfeld (Ed.), *The theory of reinforcement schedules*. New York: Appleton-Century-Crofts.

Appendix
Assembly Language Timing Routine

The following routine, written in assembler code, illustrates a simple method for timing response latency, in milliseconds, after a stimulus has been presented:

```
1.  TIME     LD      DE,0
2.           IN      A,(0)
3.           XOR     0FEH
4.           JR      NZ,BEGIN
5.  NORESP   INC     DE
6.           LD      B,81H
7.  LOOP     DJNZ    LOOP
8.           IN      A,(0)
9.           XOR     0FEH
10.          JR      Z,RESP
11.          JP      NORESP
```

After finishing (initiating) the stimulus presentation, the program jumps to Statement 1, labeled TIME, where register pair DE is reset to 0. This register pair serves as a counter for the number of cycles made through the entire timing loop. Each cycle corresponds to 1 msec. Statement 2 scans the 8-bit input port 0, which is connected to a microswitch, to check whether a response has been made. This is determined in the next statement by the XOR operation which performs a logical comparison against the hexadecimal value FE, and returns a 0 if a response has been made.

If a response has been made at this point, it is interpreted as a premature one, in which case the program jumps back to BEGIN to report the stimulus. If no response is made, register pair DE is incremented by 1. At this point, Register B is loaded with value 81 hexadecimal. This value is variable and depends on the number of statements in the entire loop. Statement 7 is itself a loop—the value in Register B is decremented by 1 with every cycle. When 0 is reached, the next statement in the routine is executed, input port 0 is again scanned, and the value returned is compared in Statement 9. If a response is made, the program jumps down to a statement labeled RESP; if not, it jumps up to NORESP and the loop continues.

Note—Some statements in this routine not pertaining to timing have been omitted for clarity; for this reason, the value 81 hexadecimal is only approximate.