COMPUTER TECHNOLOGY

Software management of timing in computer-controlled on-line experiments*

C. DOUGLAS CREELMAN

University of Toronto, Toronto, Canada M5S 1A1

Some uses of clocks by programming systems which control on-line experiments contain dangers of inaccuracy and variability in the times between events. The reasons for these dangers are discussed, and an alternative approach to the problem of timing is outlined. A sample of current systems is reviewed with regard to approaches to timing, and five questions for users to ask of potential designers or suppliers of on-line programming systems are suggested in light of the considerations raised.

This paper focuses on the design of programming systems to handle clocks and timing during an experiment. A number of systems aimed at the special needs of experimental psychologists are currently available. An unknown number of special-purpose systems have been designed for particular laboratory settings, and a growing number of general-purpose systems can be expected from computer manufacturers in the near future, some of which show promise for application in psychological contexts. It would seem that intelligent selection among alternative systems is now possible. The way in which timing is handled by a programming system should be an important criterion for selection and should weigh more heavily than more superficial aspects such as ease of learning the source language. An elegant language cannot be of much help if the times of events which the language controls cannot be counted on to be exactly those specified by the E.

This paper is concerned with programming systems, and not with hardware. Markowitz and Nickerson (1968) have discussed various types of clocks; the programming considerations apply irrespective of the type used. Clocks generate interrupts after a period of time specified by the programmer. Clocks are based on successive "ticks," which are counted either by a programmed counter or a piece of external hardware. Between interruptions from the clock, the power of the computer is available for ongoing calculation on recently acquired data or for other "background" tasks. The rate at which the clock ticks determines the grain of the timing of an experiment, the smallest time unit that can be used in controlling durations or measuring elapsed times.

TYPICAL TIMING SYSTEMS

A subsection of an experiment is diagrammed in Fig. 1, which shows, as an example, a stimulus (Event A) presented for a specified duration. The presentation is made several times (the condition). Then a response manipulandum is activated (Event B) and, if no response is recorded after a set time, another series of stimuli is presented. The flow diagram could easily be expanded for entry into any of the current experimental control systems, such as ACT (Millinson, 1968) or SCAT (Snapper, 1971, 1973).

Fig. 1. Sequence of events in a typical timed experimental control application. Waiting times—WAITs—follow bursts of input-output control and intermediate computing (rectangles). Directions of control flow are conditional on the number of times through a loop, or on S responses.

*Supported by Grant APA-111 from the Canadian National Research Council. Special thanks are due Howard L. Kaplan for many stimulating conversations, some small fraction of which are reflected here, particularly the notion that these ideas might be of interest to others.

THE PREFERRED TIMING SYSTEM

Fig. 2. Sequence of events in a preferred experimental control system. Bursts of input-output control follow setting of the clock, and are followed by IDLE time, while clock finishes timing. Conditional branching and intermediate computation occur while clock is running.

The important feature of such systems is an interval during which the computer does nothing but wait. WAITS determine the durations of actions which the computer takes, such as initiating and terminating stimuli, calculating new values for parameters, etc. In any such system, all actions taken by the computer, as schematized in boxes in Fig. 1, in addition to the actual initialization of the internal or external clock, must be completed between the end of the waiting time and the occurrence of another clock tick.

If all operations programmed between two WAIT events are not finished in the time between two ticks of the clock, two results immediately follow. First, the timing of the successive events within the experiment will not be precisely those called for by the program. The initiation of one time interval is not immediately on the heels of its predecessor, as implied by the WAIT statements. Instead, computation and input-output activity intervene. The total time for an experimental trial will be greater than specified in the program by the number of missed ticks of the clock. The second and greater danger arises when intermediate computations involve conditional branching. Then the time between successive events can vary, depending on the number of clock ticks missed while doing different amounts of computation. These dual dangers—lack of precision and lack of consistency in timing—can be avoided with careful design of programming systems.

Systems which guarantee precise timing must be designed so that the clock is always running. There must be no way for the user to program for computation or initiation of events outside those times which are successively measured by whatever clock is available.

The solution is conceptually simple but somewhat difficult to diagram; Fig. 2 attempts to point the way. The diagram is conceptually misleading in a few minor ways, but these will be described below. The most important principle is clear, however—there is no WAIT event standing alone. Instead, the first action performed after the clock marks the end of an interval is the resetting of the clock for the time to follow. Only after the clock has been restarted are the particular control and computing instructions carried out. The state called WAIT in Fig. 1 is replaced by the IDLE state of Fig. 2, which finishes out an already running time interval. Within such a system, the only way for times to be uncertain is for the clock to tick before its parameters are reset. This can be avoided by the system designer, since only a few instructions will be required so that no time will be missed, even with fast clocks and very tight control on the grain of the timing.

In both the WAIT state of a traditional system and the IDLE state of the preferred system, nothing is done except to await the end of a time period. The difference lies in the use made of the computer's hardware interrupt capability to allow computation while the clock is running. In a preferred system, the machine IDLES because it has completed all the tasks which must be finished before the current timed event is over. In a traditional system, the machine WAITS until the time is over before going about the tasks which must be done before the next timed event can be started.

The time available for computing in a preferred system is lengthened from that between two ticks to the whole of the duration. There remains a danger, of course, that the whole duration will elapse before computation is finished, but such a contingency is easily checked for by the system program. When entering the IDLE state in Fig. 2, the clock can be checked to make sure it has not finished timing yet and that the current time interval has not completely elapsed; alternatively, upon the completion of a timed interval, and after restarting the clock, the program can be checked to make certain that the interruption occurred while in the IDLE state and not from the midst of a computation. These checks are not shown in Fig. 2, but must be incorporated into any actual system.

The programming problem, from the point of view of the builder of a system, is to work at least one time period ahead of the ongoing process. This is true for two

reasons. The system must be ahead of the ongoing process because, when each interrupt occurs, the system must be primed to restart the clock immediately with the next time interval. This will involve a bit of computation immediately before the IDLE state in Fig. 2. The easiest solution is to list the times of successive events and use an index pointer. This is another way in which Fig. 2 is incomplete; the next clock start must be arranged ahead of when it actually happens.

A second way in which the system must stay ahead of the ongoing process can be introduced by distinguishing between two kinds of computation and control. One kind is termed "immediate." Immediate tasks must be executed "simultaneously" at the start or at the end of a timed interval. Bookkeeping before each IDLE time must include setting up (but not executing) a string of control commands to external devices to initiate or to terminate stimuli or record data quickly after the clock has been restarted. The bookkeeping must be done by the system before the IDLE state to assure that no time will be lost in getting the important work done right at the start of the following timed interval. To assure tight control of timing, there should be no decision-making by the system at the time that an interval is ended. All such preliminary work must be finished before settling back to await a clock interrupt. Less pressing tasks, usually involving computation on data from the prior trial or timed interval, but not involving actual control of external devices, are called "foreground" tasks. Foreground tasks often compute values to be used during the succeeding time interval. As long as the computation is finished before the next interval is due, the computation can be carried out in relative leisure.

The easiest way for a system to order the relative priority of immediate and foreground tasks is to rely on the user. In the source language that describes the experiment, an implicit priority is laid down simply through the order in which a list of commands is written for each time interval. More elegant solutions are possible in the building of a system, however.

In the interest of completeness, it should be mentioned that the IDLE state could be used for background tasks where useful work of very low priority is done. In fact, in the current version, our PSYCLE system (Creelman, 1971), as modified by H. L. Kaplan, IDLE time is used to print messages about the current status of the experiment and to monitor the Teletype keyboard for control characters to change or abort the current experimental run. In other words, even a relatively simple experiment must have three, and may have four, levels of a sort of priority. At the highest priority is the clock, which is restarted with the next time interval whenever it calls. Next comes immediate tasks, executed immediately after each clock time is initiated. Third is foreground tasks (whose results are needed for the progress of the experiment), and finally comes background tasks, which are entered once the

Fig. 3. Detail of events following the end of a timed interval; an expansion of portions of Fig. 2. Note that there is no exit except for another interrupt, or a timing error.

bookkeeping in preparation for the next clock interrupt is completed. This sequence is summarized in Fig. 3.

DISCUSSION

How crucial is the distinction between what is here called the traditional way to handle timing and the preferred way, as outlined above? The distinction is probably not particularly crucial in most current user-oriented experiment or process-control systems with timing appropriate for human or animal behavioral data gathering. The importance is minimized with a relatively new and fast computer, especially if a line-frequency (60 Hz) clock is used, with a single experiment being serviced. However, danger exists if any of these qualifications is not met. Psychophysical experiments, requiring stimulus timing to fractions of milliseconds, and physiological data gathering, where the time between samples is crucial, are examples of situations where care must be taken because a typical system is likely to cause inaccuracies. Interactive experiments, in which considerable computation is often required during an experimental trial, represent another situation where the traditional approach may lead to

difficulty. Fortunately, such interactive uses of the computer are becoming less rare; unfortunately, as the interactive aspects become more complex and sophisticated, the need for the preferred approach to timing becomes greater.

SOME CURRENT SYSTEMS

Programmers have coped with timed events in just about all possible ways, so it might be instructive to review a few published systems in this respect.

For SKED, the approach seems to be a straight traditional one, in that source-language events which satisfy a contingency and move to another state can be the termination of a timed duration. There is no way, within the language, to differentiate an immediate event from a foreground event which might be executed during the time the individual E's clock is running. Control and computation are done during transitions between states, and these transitions are not included in the timing of the program.

PSYCHOL (McLean, 1969), on the other hand, was designed with "temporal labels" for its ALGOL-like statements which, together with relative temporal labels, allow a group of coding to be done (with implicit ordering of immediate and foreground tasks) followed by an idle until the specified duration has elapsed. There are other ways within the PSYCHOL language to specify timing, but this feature is clearly of the preferred type. It has the further advantage that the E-programmer can be expected to perceive how the timing is handled through the relatively easy-to-use source language.

Lewis, Osgood, and Herbert (1973) seem to have encountered some difficulty in making an excellent system available to Es. Their PLEIADES system includes an experiment writer, designed for ease of learning and coding, which has only a traditional WAIT instruction. It can be invoked only after computation or control is completed; there is no way to do foreground tasks while a time is under way but not completed. This restriction is particularly unfortunate since their underlying system was designed specifically to do timing in the preferred manner, by deferring execution of a specified subroutine from encountering the statement until a stated time. Meanwhile, until the time has elapsed, control continues with commands in sequence, including immediate and foreground tasks. This facility is not available to Es who are unwilling to spend the 20-30 h the authors say is necessary to learn the larger, more flexible, system—a subset of which the experiment writer has access to.

A most elegant set of solutions is the PSYCBL system (Braid, Callahan, & Herman, 1971), designed as an interpretive FOCAL-like language. In this system, the programmer-E specifies sets of device-control and timing instructions which are to be interpreted into machine code immediately but which are deferred for execution until the end of the current time interval. At that time, the clock is (automatically) restarted, and then the immediate tasks are done. Interpretation and calculation

are foreground tasks, and the operation of the timing is transparent to the user.

Finally, this selective summary should mention a system described by Doll (1972). In his system (HECL), a clock is not used; timing is done with counting-loops in software. By definition, then, no other computing can be done while an E-defined time duration is passing. The interrupt facility which is available on all machines used for on-line control makes more sophisticated procedures possible and opens the possibility of designing a preferred system. However, traditional systems might just as well scrap their expensive external clocks and interrupt facilities (in single-user configurations) and move to systems such as HECL.

SUGGESTIONS TO USERS OR BUYERS

When considering a programming system for experiments, the following questions can profitably be asked of a system programmer or salesperson, in light of the above discussion:

(1) What is the time base on which the clock will run? Will it provide units of time which are appropriate for the anticipated control?

(2) How many machine instructions (or better, source-language instructions) can be executed in the time between clock ticks? The answer will depend on the speed of the machine and the efficiency of the software. The answer can and should be a precise number, not an "average" or an "estimate."

(3) What is the *maximum* number of instructions which will be executed by the system after a timed interval is up and before another timed interval is begun? If the answer is that it depends on the user's program, you are discussing a typical timing control system and not one of the preferred type. The number of instructions should be known, and fixed, and not accessible to the user. The number should, of course, be significantly less than the answer to Question 2, above. If this is not the case, careful consideration of the current and projected uses of the system is necessary to make certain that difficulties will not occur.

(4) What warning is there to the user that a timing error has happened? It is much easier to build such warning into the preferred system than into the typical one, although in the latter case, it is not impossible. However, with an external clock without special and unusual hardware flags, such protection is difficult in a typical system.

(5) If the system is capable of handling more than one independent user, how does the system allocate priority to a user when immediate tasks are to be done? Priority must be very high, and the immediate tasks must not be made to wait their turn. In most "time-shared" systems, such considerations are not particularly pressing, for it is assumed that a user does not mind a variable delay of large fractions of a second if he is typing at a terminal. In such systems, there are no immediate tasks. These are the types of systems that programmers know best, and

have for models. Unfortunately, this approach can range from inconvenient to intolerable for the on-line E.

In summary, in the control of experiments, "waiting" is not to be considered another event in the flow of the experimental program. Initiation of times and the wait for the times to elapse should bracket the events and computations which the computer controls.

REFERENCES

- Braida, L. D., Callahan, D. J., & Herman, P. W., Jr. FOCAL-CALCBL-PSYCBL: User's guide and reference manual. Communications Biophysics Group, Research Laboratory of Electronics, M.I.T., 1971.
- Creelman, C. D. Rapid response and flexible experimental control with a small on-line computer: PSYCLE. *Behavior Research Methods & Instrumentation*, 1971, 5, 265-267.
- Doll, T. J. A 4-K computer language for experimentation with human subjects. *Behavior Research Methods & Instrumentation*, 1972, 4, 27-31.
- Lewis, J. L., Osgood, G. W., & Herbert, J. J. PLEIADES: Real-time-sharing control in the behavioral science laboratory. *Behavior Research Methods & Instrumentation*, 1973, 5, 365-370.
- Markowitz, J., & Nickerson, R. S. On the timing of events in computer-controlled experiments. *Behavior Research Methods & Instrumentation*, 1968, 1, 82-91.
- McLean, R. S. PSYCHOL: A computer language for experimentation. *Behavior Research Methods & Instrumentation*, 1969, 1, 323-328.
- Millinson, J. R. A general language for on-line control of psychological experimentation. DECUS Proceedings, Spring 1968, 137-144.
- Snapper, A. G. The SKED software system. DECUS No. 8-465, Digital Equipment Co., Users Society, 1971.
- Snapper, A. G. Use of a notation system for digital control and recording. *Behavior Research Methods & Instrumentation*, 1973, 5, 125-129.

(Received for publication May 31, 1974;
revision received July 19, 1974.)