STUDENT AWARD PAPER

Software control of reaction time studies

TIM POST

Syracuse University, Syracuse, New York 13210

A description of some problems encountered and solutions generated during the programming of two reaction time tasks is provided; a few examples of the software are listed. Some of the software methods are extendable to general-purpose reaction time algorithms.

During the initial programming of a new microcomputer for experimental data collection, several problems were encountered with both software design and hardware limitations. Satisfactory answers to the problems were developed, most importantly in the areas of stimulus generation and response coding. These solutions can be used in many situations, and they can be easily transferred to other installations.

We set out to design a program or a group of programs to be run on a microcomputer that would generate a display according to our specifications, present the display with several control parameters, and measure and record a subject's reaction time to the display with a high resolution (between 100 and 500 microsec). After receiving and coding the responses, the program performs the following processing: (1) Find the different frequencies of responses (four types for the task called TASK1, six types for the second task, TASK2), (2) calculate the means and standard deviations for each type of response, (3) sort the data to find the medians, (4) write the information out to the subject's data file.

The system used is a Polymorphic 8813 microcomputer driven by an Intel 8080A microprocessor. The S100-compatible CPU board has 3K of read-only memory (ROM) that houses the monitor system, 512 bytes of on-board random-access memory (RAM), and an 8251 universal asynchronous/synchronous receiver/transmitter (USART). Currently, installed boards supply an additional 24K of RAM; other features include a video driver board that references 1,024 screen locations as dynamic memory, two minifloppy disks with corresponding disk support board, a vector interrupt keyboard, and a minicard communications interface. Included in the software is a 12K two-pass macroassembler and an 11K interpretive BASIC (both developed by Polymorphic Systems). This BASIC has excellent machine-level interface capabilities that are extremely valuable in our experimental applications.

Our tasks, patterned after general information processing paradigms, were designed such that the microcomputer has full control of a subject's session. TASK1 measures the reaction time needed to perform a visual

search. Three independent variables are manipulated: The target letter is either present or absent, the display size for any one block is either one or four characters, and the choice of distractors presented with or instead of the target letter come from four different distractor sets (phonetic letter distractors, visual letter distractors, numeric distractors, and Greek letter distractors). The display format for TASK1 has four letter positions around a fixation point, all of which fall within a subject's foveal diameter. With display size and distractor size held constant throughout any one block, a minimum of eight blocks must be generated. We use 16 data gathering blocks to balance any learning effects; 2 blocks of practice with random alphabetic characters are sufficient for familiarization with the experimental apparatus. With 17 trials per block (the first not entered as subject data), the microcomputer needs to generate 17 trials/block $\times$ 18 blocks = 306 trials, with 50 discarded from the data.

TASK2 has the following format: A memory set (MS) of two characters is presented with every trial for 2 sec and then removed; after a 500-msec pause, during which a fixation point appears, another pair of letters, the test set (TS), is displayed, and the subject responds according to whether or not the TS is the same as the MS. The MS and the TS always appear as a vertically oriented array. Two blocks per subject are required by this task, one MS from the English alphabet and one from the Greek alphabet. There are 30 recorded trials per block, chosen from three possible TSs: TS = MS (exact match); upper character differs (mismatch); lower character differs (mismatch). With two practice blocks (one Greek, one English) of 12 presentations added, a total of 86 displays are needed.

During the preliminary programming and piloting of the tasks, it quickly became apparent that machine-level software had to be implemented in order to present the displays and to measure the reaction time.

Outputting the characters with a maximally efficient BASIC algorithm took approximately 150 msec. This is quick enough to generate the entire display before eye movements can occur, but with perceptible apparent motion. When the letters were displayed from one

corner to another, a diagonal line appeared to sweep across the screen, with the letters appearing when the line crossed the appropriate locations.

A desire to keep the machine-level programming as simple as possible weighed heavily in formulating the algorithm to simulate our display problem. In TASK1, if character(s) and configurations were stored in matrix form at the basic level, a machine language function of memory required only a very simple loop written in machine language to output the characters at an increased rate. For example, 1,024 characters (a full screen) can be output in specific positions in approximately 32 msec. This loop, outlined in Figure 1, can be easily adjusted to any amount of characters in one display, with the amount of time for the total display to be output equal to $.532(59N + 20)$ microsec, where N is the number of characters to be output. With the locations for each character defined previous to testing the first subject, a multitude of display-oriented tasks could use the same loop, ranging from Sperling paradigms to pattern configuration evaluation. Although TASK2 was fundamentally different in design and subsequently required a different loop to be written, the amount of carryover knowledge from the TASK1 subroutine algorithm made the writing of the TASK2 loop a much simpler programming task.

Another problem encountered and solved was that of obtaining a highly precise measure of reaction time.

```

LETHOLD DS      9          ;CHARACTER ARRAY
LOCHOLD DS     18          ;LOCATIONS ARRAY
;
;
LXI      B,LETHOLD      ;LOAD CHARACTER ARRAY LOCATION
                        ;INTO REGISTER PAIR B
LXI      H,LOCHOLD      ;LOAD LOCATIONS ARRAY LOCATION
                        ;INTO REGISTER PAIR H
MOREOUT LDAX     B        ;PUT A CHARACTER INTO
                        ;ACCUMULATOR
MOV      D,M           ;GET HIGH BYTE OF SCREEN
                        ;LOCATION
INX      H             ;INCREMENT FOR NEXT BYTE
MOV      E,M           ;AND GET LOW BYTE OF SCREEN
                        ;LOCATION
STAX     D             ;STORE CHARACTER ONTO SCREEN
INX      H             ;INCREMENT THE POINTERS
INX      B
MVI      A,LOCHOLD     ;MOVE LOW BYTE OF LOCHOLD INTO
                        ;ACCUMULATOR
CMP      C             ;AND CHECK IF CHARACTER POINTER
                        ;IS POINTING TO LOCHOLD
JNZ      MOREOUT       ;BACK FOR MORE IF IT IS,
                        ;CONTINUE IF NOT

```

Figure 1. Character output loop.

```

HEREAGN INX      H          ;INCREMENT THE COUNTER
A          LDA     RESPLOC   ;GET THE RESPONSE CHARACTER
          CPI      0         ;SEE IF IT'S STILL ZERO
          JZ       HEREAGN   ;GO BACK IF IT IS,
                              ;CONTINUE IF IT'S NOT

NEWISR     IN      KEYPORT   ;GET THE CHARACTER FROM THE KEYBOARD
B          STA     RESPLOC   ;AND STORE IT IN THE RESPONSE LOCATION
          LHLD     ISRHOLD   ;GET THE NORMAL INTERRUPT FROM WHERE
                              ;IT WAS STORED
          SHLD     ISRLLOC   ;AND PUT IT BACK IN IT'S PLACE
          JMP      IORET     ;FINALLY, RETURN THROUGH A SYSTEM
                              ;INTERRUPT ROUTINE

```

Figure 2. (A) Training loop. (B) Interrupt service routine for timing loop.

BASIC has a TIME function, but its resolution is over 16 msec.

One solution is to disconnect the real-time clock interrupt and reconnect it to the clock generator, which delivers a pulse every 532 nsec. This was not an appealing alternative.

Instead of hardwiring, we added a section of code (outlined in Figure 2A) to the machine subroutine. This loop accesses a storage location, RESPLOC, and compares its contents to zero: If RESPLOC is zero, return for another pass, if not, return to BASIC. Some initializations are required: The register pair HL, used to count the number of passes through the loop, must be set to zero, along with RESPLOC, and the normal keyboard routine is replaced with a specialized one. The new interrupt routine (outlined in Figure 2B) inputs the keyed response from the proper port, stores it in RESPLOC (this is done in order to signal to the timing loop that a response has been made), restores the normal keyboard interrupt, and returns to the checking loop. Since all the clock cycles can be systematically counted with accuracy, a fine-timing resolution of approximately 100 microsec was obtained. This resolution can be increased to a much coarser one by adding delay operations or loops, as outlined in Figure 3.

Because the simplicity of this timing loop is hinged on the ability to use a vector interrupt, the keyboard was chosen as the vehicle for subject response. The keyboard already had the required hardware interrupt present and was able to present clear differentiation between possible responses.

The outer level programming was done in the Polymorphic BASIC, a convenient, easy-to-read, higher level language. It is possible to write all of the software at the machine level, but BASIC provides many advantages that cannot be overlooked. The inherent ease of programming in any higher level language allows processing to be accomplished with three or four concise

```

NOP          ;NO OPERATION PERFORMED,
              ;TIME DELAY-ABOUT 2 MICROSECONDS.

XTHL         ;EXCHANGE TOP 2 BYTES ON THE STACK

XTHL         ;WITH REGISTER PAIR HL TWICE,
              ;TIME DELAY-ABOUT 18 MICROSECONDS.

MVI    B,N
MVI    A,0
HERE   DCR    B
CMP    B
JNZ    HERE   ;PERFORM THIS LOOP N TIMES,
              ;TIME DELAY-10.108 N MICROSECONDS FOR
              ;0 < N < 256, 2.6 MSEC. FOR N = 0

```

Figure 3. Some methods of increasing timing resolution.

commands, as opposed to 10 or 15 in machine language. BASIC programs are also straightforward enough to allow understanding of their purposes from reading the source code, an often difficult task with machine language.

Another advantage is the ease in transferring algorithms from one software paradigm to another. Software libraries that are properly documented can be written and maintained, thus enabling users access to routines that they would normally have to write from scratch. Such a library has already proved useful. In addition, communication between installations that differ in hardware can be accomplished with greater ease: It is a much simpler task to translate a program written in MITS BASIC to PET BASIC than it is to translate code written in 6800 machine code to 8085 machine code. The differences between the instruction sets of different processors drastically inhibit transference between systems. With a higher level language, such as BASIC, transference of code can be accomplished with relative ease.

The use of BASIC as a major processing language is not without faults. For example, presentation of stimuli often requires the generation of random numbers for balancing certain dimensions. Although a BASIC function exists that produces numbers on an effectively random basis, such a function will not create a sequence of numbers that meets criteria necessarily defined by an experimental paradigm. In our applications, we require that equal presentations of possible stimulus situations occur without noticeable repetitions. Within one block of TASK1, after the first warm-up trial, eight presentations with the target character present and eight without the target letter occur. Using the random-number generator cannot meet this criterion. Even a mathematical solution does not suffice; such a method still relies on the random-number generator in such a way that the interstimulus interval varies. To solve this problem, we use an initialized array as a random-number

table. This solution requires more memory space, but the control of events is more important to our design.

Most of the BASIC programming is concerned with the generation of displays (stimulus choice and configuration) and the subsequent coding of a subject's response to the stimulus. The software was required to perform these tasks with diverse inputs; many variables were under control of the microcomputer that varied according to complicated criteria. The conventional method of dealing with these situations is to use an algorithm that uses if-then processing.

The if-then statement is one of the most widely used programming techniques. It is simple and self-explanatory and exists in one form or another in most higher level computing languages. However, this type of statement can be disruptive in experimental work.

If-then processing has two major problems: It takes a great deal of time for processing, and it does not consistently take the same amount of time. To process an if-then statement, a condition must be evaluated using finite mathematics, and then, depending on whether the condition was true in a logical sense, a statement, a set of statements, or a program branch must be executed. When these statements are used to code a group of variables, a relatively large amount of time can be used.

When many variables must be considered, not only may the cost in time produce a large delay, but the time of delay may vary, depending on the conditions present. An if-then table, a group of if-then statements used in conjunction with each other to evaluate a complicated situation, sometimes has only its first condition tested; other times, all of the conditions are examined.

Finite mathematics can be used by a BASIC programmer to deal with processing of several conditions without using if-then processing. Since most of the relationships involved in the control of a reaction time study can be broken down and considered mathematically, using such mathematics becomes a preferred alternative to other methods of processing.

Three steps are needed for finite mathematical processing: First, a linear transformation from the discrete occurrences of an event to an object set must be performed. Second, a Boolean algebra is used to process the object set. Third, a transform back to the discrete structures is performed (often, the last step is skipped and the information is stored in the object set form). These steps can be accomplished in one line of arithmetic, a form of computing that is more easily compiled or interpreted than an if-then structure. The linear transforms are not very complicated. A dichotomous event, such as target presence/absence, can be coded to a binary representation of 0 or 1; other, more complicated events (e.g., distractor set choice) can be numerically coded (i.e., 0 = phonetic distractors, 1 = visual distractors, etc.). Boolean algebras exist in many forms; two examples are the logical "or" and

“and,” and arithmetic, addition, and multiplication. Statements made in one Boolean algebra can be translated to another by simply interchanging the Boolean equivalent operators. For example, the logic statement equivalent to an exclusive-OR function,

$$(\neg P \wedge Q) \vee (P \wedge \neg Q)$$

is equivalent to the BASIC statement,

$$((P=0)*Q) + (P*(Q=0))$$

Note that the statement $(P=0)$ is equal to one when $P = 0$ and is zero when $P = 1$, thus effecting a negation on P .

This method has many uses in our experimental design. One of these is the recycling of bounded counters, such as the index variable that references the number in the random-number table that is to be chosen next. This index runs from 0 to 63 (the last entry in the table) and then cycles back to 0. One method to accomplish this is provided by the following two statements:

$$I = I + 1$$

$$\text{IF } I = 64 \text{ THEN } I = 0$$

We use the statement,

$$I = I + 1 - 64*(I = 63)$$

This line simply increments I for $I < 63$; when $I = 63$, 64 is subtracted and the net result after the incrementing is 0.

The benefits of the method are more easily apparent

when coding of input response is considered. If the keys A and L (ASCII 65 and 76) are used to indicate target presence and absence, respectively, the linear transform would appear:

$$R = (R - 65)/11,$$

yielding $R = 1$ for an L keystroke, 0 for an A keystroke, and some number other than 0 or 1 for any other keystroke. If a letter P denotes actual target presence ($0 = \text{absent}$, $1 = \text{present}$), such a response can be coded to a signal detection paradigm with the equation,

$$S = R + 2*P.$$

For the different values of S , we have: $S = 0$, correct rejection; $S = 1$, false alarm; $S = 2$, miss; $S = 3$, hit. The information carried by the variable S can be used to store the subject's reaction time to the stimulus in an appropriate location in a two-dimensional array, to be used later for statistical analysis (e.g., median) and eventual output to a subject's data file.

The system is flexible enough that implementation on another system can be accomplished, subject to certain limitations. First, a microcomputer similar to the one described is necessary: The nature of the timing loop requires not only a hardware interrupt, but also a CPU that has no task to perform other than executing the timing instructions (while waiting for a response). The software requirements are at two different levels: At the BASIC level, a machine language interface is required; at machine language level, if an 8080A CPU is not in use, a translation must be performed. If the hardware limitations are met, the software requirements are general enough to allow flexible and relatively simple transference of software.