Behavior Research Methods & Instrumentation
1981, Vol. 13 (6), 735-740

COMPUTER TECHNOLOGY

Controlling behavior experiments with BASIC on 6502-based microcomputers

FREDERICK RAYFIELD
Roosevelt University, Chicago, Illinois 60605

and

JOHN CARNEY
University of Oklahoma Health Sciences Center, Oklahoma City, Oklahoma 73190

A combination of high-level BASIC and transparent machine language routines forms an inexpensive (about $800), powerful, and easy-to-use microcomputer system for behavior research, based on the AIM-65. The interrupt-driven machine language routines keep track of real time, poll inputs, and are not typically modified for different experiments. Expandable machine language listings that support eight input lines and eight output lines on the AIM-65 are presented. The user-written BASIC program controls experiments and records data. BASIC programs for simple reinforcement schedules are used as examples. The software is readily transferred to other 6502-based microcomputers with BASIC.

For controlling experiments with microcomputers, BASIC is easy to use (Thompson, 1979), but it typically has no timing functions and it interprets program lines too slowly to poll input ports at the speed needed for some response recording. On the other hand, machine language programs are fast, but difficult to write and debug. A suitable compromise between BASIC and machine language lies in standardizing the input polling and timekeeping functions in a few machine language routines that operate on an interrupt basis, transparent to BASIC. This leaves BASIC free to handle the inputs decoded by the machine language routine, collect data, make decisions, and control inputs. In short, machine language is used for the tasks that do not change from experiment to experiment, whereas BASIC is used for features that the experimenter might wish to be very flexible.

Machine language routines to poll inputs and to keep track of time have been written and extensively tested for Rockwell's AIM-65 microcomputer. These routines and the BASIC program lines that utilize them for simple behavioral experiments are described here. Adaptation to complex experiments and to other 6502-based microcomputers is relatively straightforward.

A system used for behavior experiments is shown in Figure 1. The AIM-65 microcomputer (see Heth, 1980), made by Rockwell International, when configured with 4K RAM, 8K BASIC in ROM, and a power supply, costs about $600. It includes a full keyboard, 20-character display, cassette interface, Teletype interface, and thermal printer. Other features are: 8K monitor in ROM, including a miniassembler for machine language

Reprint address and technical questions may be addressed to the first author in the Department of Psychology, 430 S. Michigan Avenue, Chicago, Illinois 60605. The optically isolated interface circuit is available upon request. Portions of this research were supported by NIDA Grant DA 02099 to the second author.

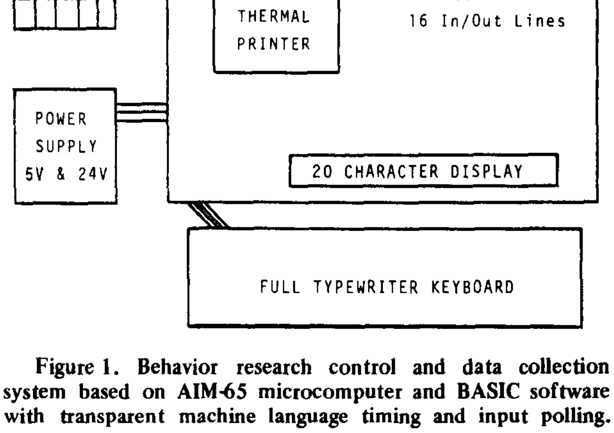

Figure 1. Behavior research control and data collection system based on AIM-65 microcomputer and BASIC software with transparent machine language timing and input polling.

programming and a text editor; 16 in/out pins; and full expansion capability of the 65K address field. The printer sets the AIM-65 in a special class for this price range.

The 24- (or 28-) V dc solid state interface used to control stimuli and detect switch closures in the experimental chamber is an optically isolated circuit. It simply translates 24 V dc to a 5-V dc TTL-compatible logic signal on the input side and translates a TTL signal from an in/out pin on the AIM-65 to a 24-V dc closure to ground on the outputs, as shown in Figure 2. A switch input causes 24-V dc current to flow through an LED in the optoisolator, current-limited by the 2-kohm resistor. The light of the LED operates a phototransistor, connecting the computer input line to ground (Logic 0). An open switch produces no light, and the computer input line floats high to Logic 1. For output control, the computer output port bits are each buffered by a TTL driver (SN7407) that lights an LED inside an optoisolator, with an external 220-ohm resistor limiting current. The resultant operation of the optoisolator's phototransistor allows current to flow from the base of a TIP-115 power Darlington transistor, thereby controlling the 24 V dc from a peripheral device (e.g., food dispenser or stimulus light) to ground. The use of optoisolators (about $1 each) prevents miswiring of the peripheral devices from damaging the computer. Except for burned out power transistors, also about $1, when short-circuited beyond their 2-A rating, these circuits have proved very durable.

The capability of the AIM-65 system shown in Figure 1 is limited by two factors. First, the number of in/out lines determines the number of inputs that can be sensed and the number of outputs that can be controlled. Therefore, the number of experimental chambers that can be controlled simultaneously is determined by the number of lines allocated to each chamber. The AIM-65, with 16 in/out lines, for example, can control four operant chambers if each has only two levers, one pellet dispenser, and one houselight. (An additional 64 in/out lines can be added to an AIM-65, allowing simultaneous experiments in four chambers, each with three levers, houselights, two reinforcement devices, three stimulus lights, and cumulative recorders.)

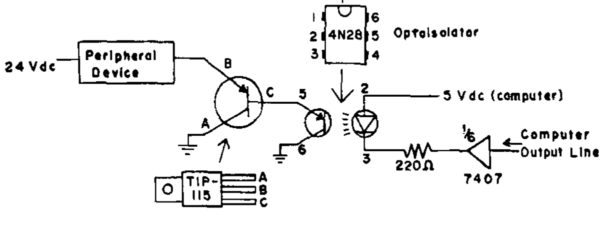

Figure 2. Interface circuits for 24- or 28-V dc input and output control, optically isolated, to TTL-compatible microcomputer in/out lines. Switch closure input shown above ground connect output. Similar optoisolators (e.g., 4N33) and power transistors (e.g., TIP-125) may be substituted.

The second limitation is the speed and size of the BASIC program. If the BASIC program is large, it may require more memory than the 4K RAM on the AIM-65. Although memory may be expanded, such a large BASIC program may not react quickly enough to control experiments precisely. A rough estimate of 5 msec/line is a reasonable rule of thumb. Conservatively estimating 20 lines of BASIC per input handling routine, the software described here handles a maximum of about 10 responses/sec.

Microprocessor BASICs usually have no ability for precise timing on a continuous basis. Fortunately, the AIM-65 has a built-in time interval generator. This timer can provide regularly timed interrupts. By counting these interrupts (100/sec), it is easy to keep track of elapsed time. BASIC can examine the count of interrupts with the PEEK instruction and can then make decisions on the basis of time.

The BASIC program depends upon the machine language routines to keep track of (count) time and to identify any valid inputs (i.e., switch closures). The BASIC program therefore has two tasks: (1) to check the elapsed time to decide if there should be any changes in the interface outputs or other conditions as specified on the BASIC program, and (2) to see if any valid inputs have been found by the machine language polling routine. A few lines of the BASIC program check these operations. When a valid input has been found, the appropriate BASIC subroutine is called. If a check of elapsed time reveals that something is to occur, a "timedown" subroutine is called. The flow of the BASIC program, from the start of the session, is shown on the left side of Figure 3.

Inputs and time, in conjunction with logical or mathematical manipulations, determine any changes in output or data handling for any possible experimental condition. The timer on the AIM-65 microcomputer interrupts the ongoing BASIC program every 10 msec, and it calls a subroutine that adds 1 to the .01-sec counter, with carry to high digits. Time is kept in terms of seconds since the session began. The interrupt routine next checks for valid inputs, storing information on inputs until BASIC handles them. Each input has an identification number, which the interrupt routine places in a location for BASIC to read directly and quickly. The interrupt does all this very quickly (in about .1 msec) and then returns to wherever BASIC was when the interrupt occurred. The flow of the machine language routines is shown on the right side of Figure 3.

Communication between machine language and BASIC is always initiated by BASIC, which can do four things: (1) determine the time, (2) obtain an input

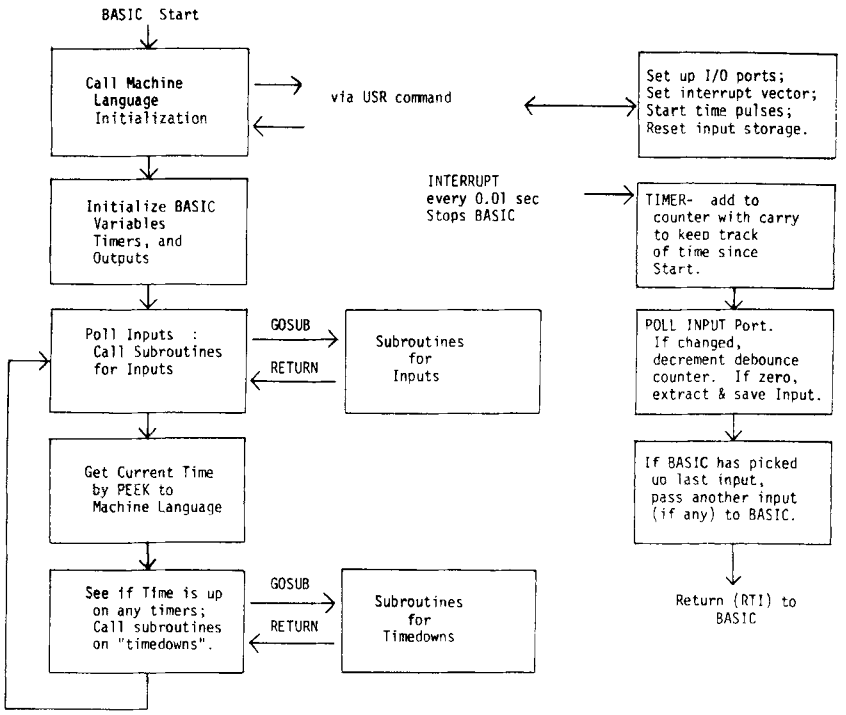
BASIC Start
Call Machine Language Initialization
via USR command
Set up I/O ports; Set interrupt vector; Start time pulses; Reset input storage.
Initialize BASIC Variables Timers, and Outputs
INTERRUPT every 0.01 sec Stops BASIC
TIMER- add to counter with carry to keep track of time since Start.
Poll Inputs : Call Subroutines for Inputs
GOSUB
RETURN
Subroutines for Inputs
POLL INPUT Port. If changed, decrement debounce counter. If zero, extract & save Input.
Get Current Time by PEEK to Machine Language
If BASIC has picked up last input, pass another input (if any) to BASIC.
See if Time is up on any timers; Call subroutines on "timedowns".
GOSUB
RETURN
Subroutines for Timedowns
Return (RTI) to BASIC

Figure 3. Flowchart of BASIC program for input recording and output control (left) shown with flow of machine language routines which keep track of time and support input polling and passing to BASIC (right). Exit from program can be made a function of time, reinforcer deliveries, or any other variable.

identification number (if any valid inputs have occurred), (3) tell the machine language routine that the last input identification number passed has been read and that BASIC is ready to handle the next input in line, if any, and (4) request the machine language routine to initiate itself for a session (i.e., reset the time count to zero, start the timer, turn all outputs off, etc.).

A user can write programs in BASIC and run experiments without understanding the detailed operation of the machine language routines, as they operate transparent to the BASIC program, which varies from experiment to experiment. The machine language routines may need changing for particular use of the 16 in/out lines, or for normally closed switches, or to adjust the debounce period controlled by the input polling routine. A larger number of input lines, for example, would require a different initialization and additional iterations of the input polling and passing routines. The miniassembler in the AIM-65 makes such changes relatively easy to someone familiar, if not experienced, with microprocessor machine language. The machine language routines presented here set up eight input lines and eight output lines and require a switch closure to persist 20 msec before being considered valid.

Four machine language routines are used with BASIC for experimental control on the AIM-65. A flowchart is shown in Figure 3. The initialization routine is used only at the start of a session. The other three (timekeeping, input polling, and input passing) are called by an interrupt every .01 sec. They are described here for the user who wishes to understand or modify their operation. For most applications, the routine may be used by a user without detailed understanding.

The initialization routine is called early in the BASIC program and is listed in Table 1 as disassembled by the

Table 1
Machine Language Initialization Routine

```
0F84  LDA #FF     Set A000 to outputs
0F86  STA A002      with D.D.R.  and
0F89  STA A000    Turn outputs off.
0F8C  LDA #0F     Set MSB of interrupt vector
0F8E  STA A405      for timer interrupt
0F91  LDA #04     Set LSB of interrupt vector
0F93  STA A404
0F96  LDA #40     Start on-board timer, with
0F98  STA A00B      continuous interrupts
0F9B  LDA #C0       (see AIM User manual or
0F9D  STA A00E       6522 data sheet for full
0FA0  LDA #00        explanation)
0FA2  STA A004    Set LSB of on-board timer
0FA5  LDA #27     Set MSB of on-board timer
0FA7  STA A005      (fine and coarse adjust)
0FAA  CLI         Clear interrupt status
0FAB  LDA A004    Start timer-  last step.
0FAE  LDA #00     Clear input polling,
0FB0  STA 0F3A     debounce, and waiting
0FB3  STA 0F3B     locations. Details in
0FB6  STA 0F3C     Table 2.
0FB9  STA 0F00    Clear all timer storage
0FBC  STA 0F01     bytes. (Reset time to
0FBF  STA 0F02     zero.)
0FC2  STA 0F03
0FC5  RTS         Return to BASIC
```

AIM-65 monitor, with comments added. Lines 0F84-0FC5 set Port A (A001) to all inputs and Port B (A000) to all outputs and turn all outputs off. They put the starting address of the interrupt routine (0F04) in the interrupt vector (A404, A405), set and start the .01-sec timer, and clear the input storage and timer locations.

The interrupt routine includes the time, input polling, and input passing routines, as shown in Table 2. Each interrupt, every .01 sec, calls the input polling routine (0F40-) and then the input passing routine (0FC7-) and then goes back to count the .01 sec (maximum about 17 X 10^6 sec, or about .5 year). The input polling routine checks for a change of any bit on the 8-bit input port (A001) by comparing it with the last valid reading of the port (stored at 0F3A) and decrements a counter (0F3B) if a change is present. If the counter reaches zero, that is, if enough time passes and the input port is still changed, the input change has been debounced and is considered valid. The particular bit or bits changed are then extracted logically, so that only changes from Logic 1 to Logic 0 are considered, corresponding to a switch closure to ground, and not a switch release. These valid input bits (or bit) are then combined to any others already waiting (0F3C) to be passed to the BASIC program.

The input passing routine (0FC7-) checks to be sure that a previously passed input is not still waiting for BASIC to act. The routine then identifies the next input waiting and places its identification number (corresponding to Bits 1-8) in the location that BASIC will PEEK to find inputs (0FFE, or 4094 in decimal). Once an input has been set up for BASIC to pick up, the associated storage bit at 0F3C is cleared and will be set again by the input polling routine when the input occurs again. The BASIC program obtains the input identification with a PEEK and POKEs zero into the pass location in order to enable further inputs to be passed (see Line 30 in BASIC programs to follow).

A simplified main line of the BASIC program, with comments, is shown in Table 3. BASIC must be started using only Locations 0200-0EFF, so as not to overwrite the machine language routines. (There is a fair amount of wasted space from 0F00 to 0FFF.) Initialization in the BASIC program is followed by the continuous input and time checking loop. Note that after initialization (Lines 10 and 20), BASIC alternates between checking for inputs (Lines 30 and 40) and fetching the current time (Line 50). The second half of Line 30 clears the input location (POKE IN, 0) so another input identification number, if any, can be put there by the machine language program. If an input is found (Line 30), identified by the numbers 1 through 8, the appropriate subroutine in the BASIC program is called. So Line 100 might be 100 PRINT "INPUT #1": RETURN, and Line 200 might be 200 PRINT "INPUT #2": RETURN. In fact, a program that prints out the name of each input is very helpful for checking the wiring of the

Table 2
Time, Input Polling, and Input Passing Routines Called by Interrupt Request Every .01 Sec

```
0F00           Divide by 100 byte
0F01           Seconds up to 255
0F02           Seconds up to 65535
0F03           Seconds up to 17x10^6

0F04 PHA       Save accumulator
0F05 LDA A004  Restart timer
0F08 TXA       Save X register
0F09 PHA
0F0A TYA       Save Y register
0F0B PHA
0F0C JSR 0F40  Do input poll and pass
0F0F LDY #00   Convenient zero
0F11 INC 0F00  Increment time counter
0F14 LDA 0F00  See if even second is up
0F17 CMP #64    (64 hex is 100 decimal)
0F19 BNE 0F2E  If not up, branch ahead
0F1B STY 0F00  Clear 0.01 sec. counter
0F1E INC 0F01  Increment seconds counter
0F21 BNE 0F2E
0F23 INC 0F02   with carry to higher
0F26 BNE 0F2E    order bytes for timing
0F28 INC 0F03    up to about 6 months.
0F2B NOP       Unused
0F2C NOP
0F2D NOP
0F2E PLA       Restore saved registers
0F2F TAY        Y
0F30 PLA
0F31 TXA        X
0F32 PLA       and Accumulator
0F33 CLI       Clear interrupt flag
0F34 RTI       Return from interrupt to
                 BASIC program.

0F3A           Previous valid input reading
0F3B           Debounce count-down
0F3C           Inputs waiting to be passed
                to BASIC

0F40 LDA A001  Input Polling- load input port
0F43 CMP 0F3A  Compare to previous reading
0F46 BEQ 0F5F  If unchanged skip ahead
0F48 DEC 0F3B  If changed, decrement debounce
0F4B BNE 0F64  If not debounced yet, skip
0F4D EOR 0F3A  Extract changed bit 1-0 only
0F50 AND 0F3A   logically.
0F53 ORA 0F3C
0F56 STA 0F3C  Store with other inputs
0F59 LDA A001  Load new port status
0F5C STA 0F3A  Save as previous reading
0F5F LDA #03   Reset debounce period (30 msec.)
0F61 STA 0F3B
0F64 JMP 0FC7  Jump to passing routine


0FC7 LDA 0FFE  Passing routine- check for
0FCA BNE 0FDA   other input waiting still
0FCC LDA 0F3C  If none, load inputs waiting
0FCF BEQ 0FDA  If none, skip ahead
0FD1 TAY       Save what's waiting
0FD2 LDX #08   Set input I.D. #
0FD4 ROL A     Check bit 7 (input #8)
0FD5 BCS 0FDB  If set, go to pass I.D.#
0FD7 DEX       Set to try next ID #
0FD8 BNE 0FD4  Loop back to check
0FDA RTS       If none waiting- give up.
0FDB STX 0FFE  Store ID # in pass location
0FDE TYA       Recover inputs waiting
0FDF AND 0F6B,X
0FE2 STA 0F3C  Clear that waiting input
0FE5 RTS        with mask, and then return

0F6C FE        Maskings for clearing waiting
0F6D FD         input bits at 0F3C after passed
0F6E FB         to 0FFE for BASIC to pick up.
0F6F F7         Note that each is all ones (FF)
0F70 EF         except for a single bit.
0F71 DF
0F72 BF
0F73 7F
```

Table 3
Simplified Main-Line BASIC Program for Behavior Research

```
 Memory size?  3839     (BASIC can use up to location 0EFF)

10 POKE 4, 132: POKE 5, 15     Tells where routine is.

20 A=USR (A)                   Runs machine language initialize.

30 A=PEEK (IN): POKE IN, 0     Get input, clear input for next.

40 ON A GOSUB 100, 200, 300,   Goto subroutine for that input,

   400, 500, 600, 700, 800     if any

50 T= PEEK (TM)                Get current time

60 GOTO 30                     Loop back

 T = Time since session start, in secs.
 A = Scratch, temporary variable
 IN = Input pass location (0FFE)
 TM = Timer location (0F00)
```

manipulanda in the experimental apparatus (this program, TESTM, is available from the first author at cost). If inputs other than simple switch closures (or openings) are needed, the same general scheme can be adapted by rewriting the machine language polling routine.

Suppose each Number 1 input is to turn on a feeder for 3 sec. Two lines must be added to the program. One line is needed to turn the feeder on (Output Bit 4, in this example) and set up a 3-sec timer. A second line must be added to the main loop to check when that 3 sec is up and turn off the feeder. The input routine at Line 100 would be: 100 A=4: POKE C1, (PEEK (C1) AND 255 − 2/\A). Line 100 shows a convenient way to turn single outputs on or off. The output port of 8 bits (C1) is PEEKed and logically AND-masked to turn off (Logic 0) Bit 4. Logic 0 means that current is flowing, which means that the device is turned on, in this case the feeder, when the new information is POKEd back out to C1 (A000). The equivalent line to turn off the feeder, again using PEEK and POKE instructions and a logic OR mask, would be: A=4: POKE C1, (PEEK (C1) OR 2/\A): RETURN. The logical masking of the AND and OR functions is not the same for all versions of BASIC (cf. APPLESOFT), but the same effect can be created in other ways. Timing the 3 sec for the feeder requires a variable, in this case T1 (Timer 1), which we set in Line 110 to the current time (the variable T) plus 3 sec: 110 T1 = T + 3: RETURN. So Lines 100 and 110 turn the feeder on and start the timer. We add to the main loop the following line: 55 IF T>T1 THEN GOSUB 160. After the current time is fetched (T) in Line 50, it will be compared with the time the feeder is to go off, T1. When 3 sec have passed, T will be greater than T1 and Line 160 will be called as a subroutine for timedown and the feeder will be turned off. This program is satisfactory, except that every time the program loops through Lines 50 and 55 thereafter, until the next input, T will still be greater than T1 and time will be wasted calling Line 160 to turn off a feeder that is already off. Therefore, T1 is reset to a very large number, one that the session time will not exceed. The variable J is assigned as a very large number, say a million, and T1 is set equal to J in Line 160. Assigning a variable, J, for the large number, which can be used for resetting all timer variables as needed, rather than using the actual number, 1000000, saves memory and results in the program's running faster because BASIC does not have to convert the variable to binary, as it would a constant. By using several constants, such as the address of the output port (C1), stored as variables at the start of the BASIC program, significant space and time savings are achieved. Combining the lines described above, we have a "complete" program in Table 4. Note that Line 5 now sets our useful constants as variables.

Because each PEEK returns just one 8-bit byte, which in decimal ranges from 0 to 255, more than a single PEEK is needed to obtain all the digits needed for the time variable, T. Therefore, Line 50 includes 4 bytes of time information: 50 T =(PEEK(TM)/100) + PEEK (TM + 1) + PEEK (TM + 2) * 256) + (PEEK (TM + 3) * 65536)). TM (0F00) is the location of the .01-sec time counter that the machine language controls. TM + 1, TM + 2, and TM + 3 are the locations (0F01-0F03) of the digits of the time counter in whole seconds.

An experiment requiring many timers may result in a large main-line loop, with many line statements similar to IF T>Tx THEN GOSUB XXXX. Using a matrix variable, T(x), instead of T1, T2, T3, . . . and using a FOR-NEXT loop around an ON-GOSUB instruction, we can save space and keep speed of execution up when a program requires many timers. This would change Line 55 (and others that might be needed following 55) to: 55 FOR A = 1 to 5; 60 IF T>T(A) THEN ON A GOSUB 160, 260, 360, 460, 560; and 65 NEXT. Adding 5 or 10 more timers, then, does not require adding as

Table 4
BASIC Program for a Continuous Reinforcement Schedule

```
 5 J=1000000: IN= 4094: C1= 40960: TM= 3840      (Initialize)

10 POKE 4,132: POKE 5,15

20 A= USR (A)

30 A= PEEK (IN): POKE IN, 0                      (Main Loop)

40 ON A GOSUB 100, 200, 300, 400, 500, 600, 700, 800

50 T= PEEK (TM/100) +PEEK (TM+1) +PEEK (TM+2)*256

55 IF T > T1 THEN GOSUB 160

90 GOTO 30

100 A=4: POKE C1, (PEEK (C1) AND 255- 2^A)       (Input Subroutine)

110 T1= T+ 3

120 RETURN

160 A= 4: POKE C1, (PEEK (C1) OR 2^A): T1=J      (Timedown Subroutine)

170 RETURN

 A=  Scratch, temporary variable
 J=  Large number to clear timers
 IN= Input pass location (0FFE hex)
 C1= Output port location (A000 hex)
 TM= Hundredths of seconds location (0F00 hex)
 TM+1, TM+2, TM+3=  Seconds, seconds*256, seconds*65536 locations
 T1= Timer, length of reinforcement event
```

Table 5
BASIC Input and Reinforcement Subroutines for FR 10

```
100 C(1)= C(1)+ 1                              Count responses.

105 IF C(1) < F1 THEN RETURN                   If ratio not done,
                                                return to main loop.

110 A= 4: POKE C1, (PEEK (C1) AND 255- 2^A)    Turn feeder on.

115 T1= T+ 3: C(2)= C(2)+ 1                    Set rft. interval
                                                and count rft.
118 F1= C(1)+ 10                               Set up next ratio.

120 RETURN                                     Return to main loop.

 C1- Output port location (A000)
 C(1)- Response counter
 C(2)- Reinforcement counter
 F1- Total response count needed for next reinforcement
 T1- Timer, length of reinforcement event
```

many more lines to the main-line loop, just changes in 55 through 65.

A matrix variable is also a convenient way of assigning counters for several chambers. The variable C1 (x) might be a group of counters for Chamber 1, and C2 (x) the same for Chamber 2, and so on. Counters are easily incremented by adding 1; for example, C1 (3) = CI (3) + 1.

To convert the program in Table 4 to an FR 10 schedule of reinforcement, F1 can be specified as a temporary variable to store the ratio requirement, C(1) can be specified as the number of responses cumulating, C(2) can be specified as the number of reinforcers received, and Lines 100-120 can be changed as shown in Table 5, with comments on the right. Responses during the 3-sec reinforcement are counted toward the next ratio in this example. If Line 118 is moved to the timedown subroutine, say as Line 165, then the ratio is reset at the end of the 3-sec reinforcement and responses during the 3 sec are counted overall, but not in the next ratio. Note that at the end of each reinforcement, F1 is set equal to the current total response count plus the required ratio.

An interval schedule can be programmed by checking the time on each response and comparing it with the time at last reinforcement. Line 105 in Table 5 would be changed to check if the interval is done, and Line 118 would be changed to set up the next interval with each reinforcement: 105 IF T<T(1) THEN RETURN and 118 T(1) = T + 60 for an FI 60-sec schedule.

For variable ratios and intervals, the RND function can be used. In general, for a VR schedule ranging between A and B, with an average of (A + B)/2, the formula (B – A) * RND (1) + A can be used. For a VR 100 schedule ranging from 10 to 190, Line 118 would be: 118 F1 = C (1) + 180 * RND (1) + 10. The same tactic can be used to generate VI schedules. For a VI 60-sec schedule ranging from 5 to 115 sec, Line 118 would read: 118 T1 = T + 110 * RND (1) +5.

While the examples used here have been simple schedules of reinforcement, this method can be used in a wide variety of laboratory-control and event-recording situations. The technique of doing input polling and timekeeping in machine language in order to squeeze the necessary speed and timing capability from BASIC can be used with other microcomputers. The input and output ports and their control (DDR) registers may have different locations. In an Apple, for example, they depend on the location of parallel cards in the expansion slots. Since most microcomputers have no real-time clock to generate regular interrupts, such must be added and programmed appropriately to generate interrupts and the appropriate location of interrupt vectors must be determined. Depending on the computer's memory usage, the location of the machine language routines is varied. (In the Apple, the routines may be placed just below the disk operating system.) Because some versions of BASIC have different AND and OR functions, machine language routines may be needed to achieve output control of individual lines.

Methods for optimizing the speed of any BASIC program are usually included in the user's BASIC manual. Omitting unnecessary spaces, using variables for constants, putting often-used subroutines early in the program, combining several commands on the same line, and leaving out remark lines are a few of the most important. This need to promote quick execution time is the only caveat in an otherwise very powerful and flexible system.

REFERENCES

Heth, C. D. The AIM-65 microcomputer as a laboratory control device. *Behavior Research Methods & Instrumentation,* 1980, 12, 364-366.

Thompson, G. C. Behavioral programming with the Apple II microcomputer. *Behavior Research Methods & Instrumentation,* 1979, 11, 585-588.

(Received for publication April 1, 1981;
revision accepted October 27, 1981.)