SESSION IV TECHNICAL ISSUES IN SYSTEM DESIGN

STUDENT COMPETITION AWARD

Using an Apple II microcomputer for real-time control in a behavioral laboratory

WINFORD A. GORDON, DON FOREE, and DAVID A. ECKERMAN
University of North Carolina, Chapel Hill, North Carolina 27154

This report describes the hardware and software developed to implement an Apple II (48 KB) as a real-time control device for operant experiments. The hardware has a straightforward design, so that it is readily understandable and can be built by individuals with only minimal experience in the use of integrated circuits and other electronic components. The software routines listed below represent an approach to controlling and handling the data generated by an operant experiment. Using these routines, we are able to record each response and experimental event, the time of the occurrence, and the conditions at that time.

The advent of microprocessor systems has made real-time computer control broadly available. Yet, lacking the electronic skills needed to interface a microprocessor with laboratory equipment, many researchers are reluctant to implement a computer control system. Commercially produced hardware and software systems are available, but the cost of these systems is often prohibitive. We have developed a control system that is straightforward and inexpensive. It is our hope that this description will help others who lack extensive experience in programming or circuit design to implement microprocessor control of laboratory experiments.

Our laboratory is equipped with an Apple II microprocessor configured with 48-KB memory, an AIO serial-parallel interface card (SSM Microproducts, Inc.), a Mountain Hardware Apple clock, and a single disk drive. The sections below describe the hardware expansion we built for the AIO parallel interface and our software approach.

This system was developed and documented as part of the dissertation research of Winford A. Gordon. The work was partially supported by Public Health Service Grant MH-14269, awarded to the experimental psychology program of the University of North Carolina. There was also support from the Smith Research Fund of the University of North Carolina and from a Sigma Xi grant in aid of research. Reprints of this paper and program listings can be obtained by writing David A. Eckerman, Department of Psychology, UNC-Chapel Hill, Chapel Hill, North Carolina 27154.

THE INTERFACE EXPANSION

The first element in the system is the AIO serial-parallel interface card, which has two 8-bit parallel ports and two serial ports. Since we do not use the serial ports in this application, the expansion circuit could be adapted to any device that provides two 8-bit parallel ports. The AIO card is inserted into one of the main board slots in the Apple II to provide input and output (I/O) lines. We use one of the ports for output and the other for input. Each port is represented in the Apple as a single memory location and can be accessed by using a PEEK or POKE from APPLESOFT BASIC, by using a free union variant record in PASCAL, or by using the load or store command in an assembly language routine.

With this card, additional interfacing logic is required in order to control more than eight devices. The first system described below is a simple, inexpensive circuit that expands the output of the interface to as many as 48 separate devices. It is not uncommon for laboratory studies to require such an array of outputs. Our operant studies of stimulus control, for example, require at least 36 output lines to operate the 12 stimulus lamps in the three in-line projectors that display the training and test stimuli. We also need output lines to present reinforcement and operate the chamber houselight. The circuit described here meets these specific needs: It operates all 36 stimulus lamps, although only one at any time, and it operates the houselight and hopper at any time.

Our circuit recognizes logical combinations of the 8 bits in the interface port. Bits 0-3 are combined to specify a value between 1 and 16. Since we have only 12 stimulus lamps in each projector, we use only values 1-12. The value in interface port Bits 0-3 determines which of the 12 lamps in the projector is potentially enabled. Bits 4 and 5 produce four mutually exclusive states: OFF, ENABLE PROJECTOR A, ENABLE PROJECTOR B, or ENABLE PROJECTOR C. Thus, these 2 bits turn on the selected lamp in the selected projector. Bits 6 and 7 are operated independently to control the houselight and the reinforcement hopper, respectively. Table 1 shows how these 8 bits from the AIO card are combined to provide the number of outputs we use.

As an example of how these combinations work, consider that placing the value 19 in the output port turns on Lamp 3 in Projector A, whereas placing the value 35 in this location turns on Lamp 3 in Projector B. The binary representation of these values reveals that the lowest 4 bits (Bits 0-3), which select the lamp, are equivalent, but Bits 4 and 5, which select the projector, differ. In binary notation, these values are 00010011 vs. 00100011.

Table 2 lists the code that would be used to produce the various outputs of our interface system. Decimal values are listed, with their hexadecimal equivalents. These values would be given in a table for the table lookup, as described in the software section.

The interface hardware can be described as having three stages (see Figure 1). Stage 1 involves optoisolation of the computer from the interface. The eight lines in

Table 1
Logical Combinations of Eight Lines to Yield 38 Outputs

AIO LINE #	OUTPUT LINE #	
	0 --->	
	1 --->	
0 --->	2 --->	LINES 0-3 COMBINE
	3 --->	LOGICALLY TO CREATE
1 --->	4 --->	12 LINES FOR THE
	5 --->	STIMULUS LAMPS IN
2 --->	6 --->	THE THREE PROJECTORS
	7 --->	THIS IS DONE WITH
3 --->	8 --->	A 4-TO-16 LINE
	9 --->	DECODER IC COMBINED
	10 --->	WITH AN INVERTER
	11 --->	
<hr/>		
4 --->	12 ---> A	LINES 5 AND 6
	13 ---> B	COMBINE TO ENABLE
5 --->	14 ---> C	ONE OF THE THREE
		PROJECTORS
<hr/>		
6 --->	15 --->	TURN ON THE HOUSE
		LIGHT
<hr/>		
7 --->	16 --->	RAISES THE FOOD
		HOPPER

Table 2
Output Code for the Apple II AIO Card in Slot 4
Output Port = 49312 (\$C0A0)

OUTPUT #	OUTPUT GROUP		
	A	B	C
0	16 (\$10)	32 (\$20)	48 (\$30)
1	17 (\$11)	33 (\$21)	49 (\$31)
2	18 (\$12)	34 (\$22)	50 (\$32)
3	19 (\$13)	35 (\$23)	51 (\$33)
4	20 (\$14)	36 (\$24)	52 (\$34)
5	21 (\$15)	37 (\$25)	53 (\$35)
6	22 (\$16)	38 (\$26)	54 (\$36)
7	23 (\$17)	39 (\$27)	55 (\$37)
8	24 (\$18)	40 (\$28)	56 (\$38)
9	25 (\$19)	41 (\$29)	57 (\$39)
10	26 (\$1A)	42 (\$2A)	58 (\$3A)
11	27 (\$1B)	43 (\$2B)	59 (\$3B)
12	28 (\$1C)	44 (\$2C)	60 (\$3C)

Note—To use either of the independent lines, add 64 (\$40) for Line 7 or 128 (\$80) for Line 8 to the values listed above.

and out of the AIO card are passed through standard optoisolators (Radio Shack 76-1628) to protect the Apple from any circuit overloads. The light-emitting diode (LED) in the optoisolators is driven by the 5-V signal from the Apple, and the phototransistors switch a 5-V signal generated by an external power supply. The second power source is needed if the Apple is to be completely isolated and protected from circuit overloads; it also avoids potential problems with power demands on the Apple power supply.

Stage 2 in the circuit contains the logic needed to create the additional output lines. All signals in this stage are standard 5-V transistor-transistor logic (TTL) signals. The outputs of Optoisolators 0-3 are passed into a 4-line-to-16-line decoder (LS74154). The decoder provides 16 discrete outputs: one output is grounded, or "pulled low," for each of the possible configurations of Bits 0-3. These outputs are inverted (LS7404), so that the grounded output becomes a 5-V signal, and passed into Stage 3 of the circuit. The outputs of Optoisolators 4 and 5 are logically combined with inverters and a set of two-leg AND gates (LS7408). A different gate produces a 5-V output for each of three conditions of Optoisolators 4 and 5: on/off, off/on, and on/on. The outputs of Optoisolators 6 and 7 are passed directly to Stage 3.

Stage 3 consists of 17 additional optoisolators. The input to these optoisolators is the 5-V TTL signal generated in Stage 2. The output is a 12-V signal that operates NPN power transistors (Radio Shack 76-2042). The power transistors switch the 28 V needed to operate the laboratory equipment we use. In our application, the equipment consists of small incandescent bulbs and relay coils. Figure 2 details the circuit between the transistors, relays, and bulbs. The maximum power demand placed on our 28-V power supply never exceeds 1.5 A.

Figure 1. The full diagram of the first interface expansion circuit. The three stages of the circuit are labeled across the top of the figure. The resistance values are: $R1 = 470$ ohms, $R2 = 4,700$ ohms, and $R3 = 10$ kohm. The part numbers are standard TTL integrated circuit numbers, except for the optoisolators (Radio Shack 276-1628), the voltage regulator (7812), and the NPN power transistors (Radio Shack 276-2042).

Figure 2. Details of the connections between the transistors and the relays and bulbs from the first expansion circuit.

Input to the Apple passes through a different circuit, shown in Figure 1. The common side of three microswitches is supplied with 5 V from the external supply so that their status, open or closed, can be taken directly into the TTL circuitry (the bottom three lines in Figure 1). A Schmitt trigger (LS7414) eliminates contact

bounce for any of the microswitches, and the signal is passed into an optoisolator and then to the AIO card input port. We use only three of the eight discrete inputs available in the port and ground the other input lines. (Note that all unused IC inputs should be grounded.) Each of the three possible inputs is assigned a separate bit in the input port; thus our possible input values are 1, 2, and 4.

While the current system provides the many stimuli we need for studies of stimulus control, it cannot illuminate more than one stimulus lamp at any time. To allow for multiple stimulus presentations, a somewhat different arrangement would be used. By accepting the limitation of using only three stimuli in each of three projectors, we could arrange the presentation of any stimulus or stimulus combination at any time. As in the circuit already described, the AIO Lines 0-5 would control the output lines that operate the stimulus lamps. Pairs of bits would control the stimulus projectors: Bits 0 and 1 to determine which stimulus would be presented by Projector 1, Bits 2 and 3 for Projector 2, and Bits 4 and 5 for Projector 3. Lines 6 and 7 would still be used to control the reinforcement hopper and the houselight. Table 3 shows how the eight lines from the

Table 3
Logical Combinations of Eight Lines to Yield 11 Outputs

AIO LINE #	OUTPUT LINE #	
0 --->	0 --->	
1 --->	1 --->	LEFT STIMULI 1,2 and 3
	2 --->	
<hr/>		
2 --->	3 --->	
3 --->	4 --->	CENTER STIMULI 1,2 AND 3
	5 --->	
<hr/>		
4 --->	7 ---> A	
5 --->	8 ---> E	RIGHT STIMULI 1,2 AND 3
	9 ---> C	
<hr/>		
6 --->	10 --->	TURNS ON THE HOUSE LIGHT
<hr/>		
7 --->	11 --->	RAISES THE FOOD HOPPER

AIO card would be combined to provide these 11 outputs, and Table 4 lists the code that would be used in the table lookup.

The hardware details of this system would be very similar to those of the earlier circuit, except for some differences between the two circuits in Stage 2, as shown in Figure 3.

These circuits are not necessarily the most efficient or optimal arrangements, but they are adequate. Importantly, the design is simple, straightforward, and inex-

pensive. Given that most behavior laboratories have 28-V power supplies available, the cost of the parts for developing this expansion system should be less than \$200. The cost of a card to supply the parallel ports should be between \$100 and \$150.

SOFTWARE FOR CONTROL

The BASIC Approach

APPLESOFT BASIC is usually the resident language for the Apple II microcomputer. Often, BASIC is the first, and only, language an Apple user employs. Given this popularity, it seems that many users could be served by a laboratory control system that uses BASIC as a control language. There are, however, limitations in BASIC that complicate the control of I/O. Control routines for I/O often demand very complex BASIC programs that are too slow to be practical.

What the Apple user needs is a system that allows high-speed BASIC, or assembly language, routines for I/O control to be activated with simple BASIC instructions. When the complexities of I/O control are simplified by these routines, the programming becomes "transparent" to the programmer and the I/O control routines look just like other BASIC applications. Our system seeks to be "transparent," to serve the novice programmer in three ways. First, it simplifies the programming of external control. Second, it provides easy access to a real-time clock. Last, it performs data storage in an efficient, simple manner with minimum effort. These features are detailed below.

In arranging transparent external control, the I/O functions must be markedly simplified. Communication between the Apple II and external devices is accomplished through "memory mapped I/O." This means that I/O

Table 4
Matching-to-Sample Output Codes AIO Card in Slot 4 Output Port = 49312 (\$C0A0)

IF THE LEFT KEY IS TO BE CORRECT (RESPONSE = 1)	IF THE RIGHT KEY IS TO BE CORRECT (RESPONSE = 4)
STIMULUS 1 AS THE CENTER KEY SAMPLE --> 4	
STIMULUS 2 ON THE RIGHT KEY --> 37	STIMULUS 2 ON THE LEFT KEY --> 22
STIMULUS 3 ON THE RIGHT KEY --> 53	STIMULUS 3 ON THE LEFT KEY --> 23
STIMULUS 2 AS THE CENTER KEY SAMPLE --> 8	
STIMULUS 1 ON THE RIGHT KEY --> 26	STIMULUS 1 ON THE LEFT KEY --> 41
STIMULUS 3 ON THE RIGHT KEY --> 58	STIMULUS 3 ON THE LEFT KEY --> 43
STIMULUS 3 AS THE CENTER KEY SAMPLE --> 12	
STIMULUS 1 ON THE RIGHT KEY --> 31	STIMULUS 1 ON THE LEFT KEY --> 29
STIMULUS 2 ON THE RIGHT KEY --> 47	STIMULUS 2 ON THE LEFT KEY --> 30

Note—To use these codes, load the sample code and, for the test, load the value next to the incorrect choice you wish to present. For example, if you present Stimulus 2 on the center as a sample and want the left key to be the correct choice, then you can present either Stimulus 1 or Stimulus 3 on the right key. This would be done by loading 26 or 58, respectively.

Figure 3. The change in Phase 2 that implements the second expansion circuit. All part numbers are as detailed above.

is arranged by loading a number into a particular address in the computer's memory. For example, the two 8-bit ports provided by our AIO interface card are individually represented by a single memory location in the Apple. Each bit in the port is associated with one line for either output, control of a device, or input, the monitoring of a device. One programming approach to external control is to isolate, check, and change individual bits in the I/O address. Such a bit-by-bit orientation allows the user to turn a single device on or off, but the user must carefully monitor the other bits in the port to avoid accidental changes and inappropriate combinations. Developing and using subroutines to monitor each of the eight lines for external control is quite difficult.

An alternative to the bit orientation, however, is an approach that changes all 8 bits, or the byte, in the interface port. Since each byte is a unique combination of bits that are on and off, each byte exactly determines what lines are on and off. Since the byte represents a finite number of values (0-255), various control codes can be passed to variables in a table lookup. As an illustration, if the AIO interface card is in Slot 4, the output port is represented at Memory Location 49312. Suppose the value 1 turns on the line corresponding to the blue lamp over the left lever in an operant test chamber and the value 0 turns it off. A program could initially set the variable OP (for output port) equal to the port address, the variable LB (for the left blue lamp) equal to the output code for the left blue lamp, and the variable OFF (for all stimuli off) to 0. Once set, these variables can be used as constants, and control becomes transparent. The program simply uses the POKE instruction to load the desired value directly into the specified memory location.

Using variable names, rather than bit configurations or numeric values, is desirable for the ease of writing,

understanding, and maintaining a control program. A table lookup can use variables names like LB, which are mnemonically symbolic, and set them equal to the correct control code. The table for the lookup, which is developed only once, can then be included in all subsequent control programs. For example, see Listing 1 in the appendix. (Note that the line numbers in Listing 1 are grouped to allow subsequent examples to be combined in an expanding program.)

The second concern in developing a transparent system is easy access to a real-time clock. In the example in Listing 1, a short FOR-NEXT loop was used to control the duration of the presentation of the stimulus and the reinforcer. This is sometimes called "software timing." Since such a routine will take different amounts of time to execute, depending on the other aspects of the program, and since such routines demand complete dedication of the computer during the timing cycle, software timing is not the best option. It would be better to access a real-time clock as an interval timer. Unfortunately, programs that access a peripheral clock often involve program and execution overhead, since the user must explicitly arrange to halt execution while the time data are passed between the clock and the control program and then explicitly arrange to resume execution. The transparent option should provide the user with a timer that can be easily read within the control program without explicitly halting and resuming control execution.

We use the Mountain Hardware Apple clock, which provides two timing options. The clock provides the time data in a series of memory addresses (-16172 through -16176 for Slot 5). Time can be read from these addresses and then reformatted for use in a control program. This use will be described in a later section. With BASIC programs, we adopted a different approach. We utilize the interrupt signals issued by the Mountain Hardware Apple clock. A TIMER subroutine is executed each time the interrupt signal is issued from the clock. Since the interrupts occur regularly without intervention from the control program, the clock passes time data to the control program with the routine being transparent to the user. The specifics of this routine are quite simple. An initialization routine, which begins at Address 816, sets the Apple clock to issue an interrupt 10 times each 1 sec. Each interrupt from the clock increments a first-order counter (.1 sec). When this first-order counter equals 5, a second counter (.5 sec) is incremented and the first-order counter is reset to 0. When the second-order counter equals 255, a third-order counter (127.5 sec) is incremented and the second-order counter is reset to 0. These three counters cumulate time up to 32640.5 sec, which is more than 9 h. Each of these counters occupies one address in memory. To determine the current time, the control program simply PEEKs into these locations. These particular time counts (.1, .5, and 127.5 sec) are used in order to obtain .5-sec accuracy in timing experimental events.

This represents a convenient level of accuracy, but a greater resolution is possible. Timing could be accurate to the millisecond.

Once the interrupt is enabled at the start of the program, time is cumulated continuously. The .5-sec counts, which reside at Address 830, and the 127.5 sec counts, at Address 831, are available via the PEEK command. The program has only to read and add the values to determine the time. For example, the **TIMER** routine would control timing were the lines in Listing 2 of the appendix added to the example in Listing 1.

As with the table for the lookup, the **TIMER** subroutine would be included in each program. **TIMER** would be loaded into memory early in the program as a binary file, and then the user would be able to call for the time at any point in the control routine.

A third concern in simplifying BASIC programming is arranging for efficient data storage. One approach to storing data is to fill successive elements in a BASIC array. This method is slow and uses an excessive amount of memory. For example, in our experiments, data are recorded for each event (e.g., response, component change, reinforcer). The data are stored as 8 bytes. These bytes include a 1-byte code for the event, a 1-byte code for the conditions at the time of the event, 3 bytes for the time at the onset of the event, and 3 bytes for the time at the offset of the event. If these eight values are stored as elements of an integer array, then one event will fill 16 bytes of memory (2 bytes/integer). A single session with 5,000 such events requires 48,000 bytes of memory for data storage. If this much space were reserved for a storage array, there would be no room for the control program. Further, writing an integer array to disk would introduce substantial delays in the experiment. In addition, the time taken to assign these eight values to an array and to increment the array counters would cause the program to miss events.

In light of these problems, we adopted a different storage strategy. Since the data are of a standard format, 8 bytes in the sequence described above, we store the data directly in a reserved range of memory. To allow direct storage, we use the **HIMEM** command to reserve the top 18,432 bytes of memory in the Apple (Addresses 20224-38656, \$4F00-\$95FF) as storage range. We then use a machine language routine to store the data values in successive memory locations within this range. Each time an event is recorded, the control program passes a single value for the event to the data storage subroutine. The subroutine finds the other 7 bytes of data, increments a counter that refers to the next available memory location, and puts the data in this location. When the first half of the session is complete, an **END OF DATA** flag is stored and the data are written to the disk.

This transfer to the disk is made using the DOS command "**BSAVE**." **BSAVE** creates a binary file that consists of the information from a range of memory.

We save all of the memory between 20224 and 38656 (\$4F00 and \$95FF), which can contain the information from as many as 2,274 events. The time needed to write the data to the disk using the standard Apple DOS is about 30 sec; we schedule this 30-sec operation during a single extended intertrial interval. Three 1-h sessions (110,592 bytes of data, or 13,824 events) are stored on a single 5.25-in. disk. In subsequent analyses, these binary files are loaded back into memory and accessed through the PEEK command.

Thus, this system avoids the problems associated with limited storage space, storage speed, and memory-to-disk transfer speed, and it is quite simple. To the user, the data storage subroutine requires only two commands. First, the control program must pass the event code to the data storage subroutine via Address 252 (**POKE** 252,x), and second, it must call the data storage routine (**CALL** -19968). As with the table for the lookup and the **TIMER** subroutine, the routine for storage is included in the control program as part of an initialization procedure.

The lines in Listing 3 of the appendix expand the previous example to monitor the input port for the occurrence of a response. When the response occurs, the event code for a response is passed to the data storage subroutine and the subroutine is called.

The PASCAL Approach

BASIC is very easy to learn and already very popular; therefore, it occupies an expanding place in microprocessor applications. There is a second language, however, that has much to offer the Apple user, especially when one is working with large complex programs that must operate quickly in real-time. This second language is Apple PASCAL.

The PASCAL system allows frequently used subroutines (PASCAL procedures and functions) to be combined into "libraries" for use by multiple programs. Programs can call the procedures and functions from the library by name. These routines can be progressively combined into larger procedures or larger subroutines. Thus, "transparency" is easily obtained in implementations of PASCAL language programs.

In addition, assembly language programs written with the PASCAL assembler may be added to the libraries as easily as are PASCAL language routines. The user does not need to know which language performed a given procedure or the specifics by which the procedure performs its duties; the user needs to know only what parameters the procedure requires and what the procedure does.

Three other features of PASCAL bear directly on external control. First, PASCAL uses eight characters rather than two characters in identifying variable names. This provides greater opportunity to use mnemonically significant variable names. Second, PASCAL allows the programmer to declare constants that cannot be changed

throughout the execution of the program. This is in contrast to BASIC, in which a variable used as a constant can be changed by inadvertently reusing the variable name. Finally, one of the greatest strengths of PASCAL is the faster execution speed of most routines. To obtain this speed, PASCAL compiles programs before running them. The inconvenient part of this system is that the user must learn how to operate the PASCAL compiler and take the time to compile the routines. The additional effort and time involved in compiling programs sometimes discourages novice users and prevents their pursuing PASCAL as a control language.

What follows is a description of how PASCAL routines could perform the control functions detailed in the previous discussion of BASIC. As before, the three topics are external control, interaction with the real-time clock, and efficient data storage.

Although PASCAL does not provide a PEEK or POKE command for external control, similar functions can be developed in library subroutines. The procedure that resembles a PEEK will be called TURNON. This name was chosen because it is closely related to the function it performs. TURNON requires one parameter, the integer value of the BYTE to be loaded into the output port. Checking for external events, or input, will be performed by the function EVENT, which requires no parameters. EVENT will return an integer value that corresponds to the BYTE in the input port.

The PASCAL code shown in Listing 4 of the appendix looks for a response, indicated by a value of 1 in the input port, and branches to provide reinforcement when a response is detected. This is continuous reinforcement (CRF). Note that the constant declaration at the beginning of the program is similar to the table lookup used in BASIC. As with the table in BASIC, this declaration would be a required part of every program.

For timing via PASCAL, we followed a strategy different from that used in the BASIC routines. The Mountain Hardware Apple clock provides time resolved to the millisecond. That time is available in five memory addresses that can be read from a program. The contents of these addresses are coded in a complex fashion and must be reformatted to be used in timing. To read time at a very fast rate, we elected to read the memory addresses and save the time data to be reformatted later when timing was less critical.

To access the time data, we use a PASCAL routine we have written and called TIMEPEEK. TIMEPEEK reads the lower 3 bytes of time data from the Apple clock (0.4096.999 sec). Should we need to time longer intervals, we could read the higher order bytes of time data. The 3 bytes may be reformatted into a real number representing time, resolved to the millisecond, by a

function we wrote called TIMEFORM. TIMEPEEK cannot check to see if a byte on the clock board rolled over during a reading operation, but since the procedure call takes less than 1 msec this error is very infrequent.

Finally, PASCAL allows us to parallel the BASIC approach to storing large quantities of data. PASCAL does offer an advantage over BASIC, since assembly routines do not have to be written for this function. In PASCAL, a user can define the data structure to allow for the most efficient use of memory. For example, the data structure used in the BASIC example above could be defined in PASCAL as shown in Listing 5 of the appendix.

In this case, one event (of the TYPE event data) occupies 8 bytes of memory. Note that the PASCAL packing algorithm requires that the data structure have an even number of bytes. Should it be absolutely necessary to store data in records with an odd number of bytes, then assembly language routines could be written, as in the BASIC example.

To know how much memory space is available for data storage, the MEMAVAIL function can be used. The user might want to define an array of a fixed number of elements of the type "event type," to be filled and saved when full. Alternatively, the user might allocate memory for events dynamically with the procedure NEW (eventdatpointer) and save the data to mass storage only when the available memory falls below a certain level. In either case, one could make use of the high-speed I/O operation BLOCKWRITE to save the data to mass storage. With the BLOCKWRITE procedure, 8 KB of memory can be saved to disk in about 1 sec.

CONCLUSION

The combination of these circuits and a carefully developed software system can serve the novice programmer with relative ease. The software routines need be developed only once, and then they may be provided to less skilled programmers. The novice programmer may decide to move beyond the transparent system in order to understand how interfacing, real-time control, and efficient data storage are accomplished, but the move becomes an option rather than a necessity.

As a final emphasis, we would like to note that the implementation of a computer system, like the one we describe, is not difficult. By working with simple electronic designs and relying on carefully developed subroutines, a microprocessor-based system can be implemented with minimum effort and expense. The growing interest in microprocessors should spread to laboratory applications both as an educational opportunity and as an efficient control technology.

Appendix

```

1  REM THIS ROUTINE WILL EXECUTE A
  TABLE LOOK-UP TO SET THE VARIABLES
  EQUAL TO THE CONTROL CODES AND THEN
  PRESENT A BLUE LIGHT OVER THE LEFT LEVER,
  AFTER A BRIEF DELAY THE LIGHT WILL BE
  TURNED OFF AND A REINFORCER WILL BE
  PRESENTED, AFTER WHICH ALL STIMULI
  WILL BE TURNED OFF

10  GOSUB 1000
60  POKE OP,LB
61  FOR N=1 TO 100:NEXT
180 POKE OP,SR
181 FOR N=1 TO 1000:NEXT
200 POKE OP,SR
210 END

999 REM LINES 1000-1080 ARE THE TABLE
1000 OP=49312
1010 OFF=0:SR=80
1020 LB=1
1080 RETURN
2000 REM L=LEFT, B=BLUE
      OFF = ALL STIMULI OFF,
      SR=REINFORCER

```

Listing 1.

```

1  REM LINES 61 AND 191 ARE DELETED
2  REM THIS ROUTINE EXECUTES THE TABLE LOOK-UP
  TO SET THE VARIABLES EQUAL TO THE CONTROL
  CODES. IT THEN PRESENTS A BLUE LIGHT OVER
3  REM THE LEFT LEVER, AFTER A 10 SEC DELAY IT
  TURNS THE LIGHT OFF AND PRESENTS A 3 SEC
  REINFORCER

20  PRINT D$;"BLOAD TIMER"
40  CALL II
50  POKE IE,1
70  GOSUB 500
80  ST=TM
130 GOSUB 500
140 IF TM-ST<30 THEN 130
150 GOSUB 500
160 ST=TM
170 GOSUB 500
190 IF TM-ST<6 THEN 180

500 TM=PEEK(830)+(PEEK(831)*255)
510 RETURN

1030 II=816:IE= -16167:C1=830:C2=831
1050 D$=CHR$(4):REM CHR$(4) IS CTRL D

2000 REM TIMER= INTERRUPT SERVICE ROUTINE,
      ST=START TIME:TM=CURRENT TIME
      C1=.5 SEC COUNTER:C2=127.5 SEC COUNTER
2010 REM II=INTERRUPT INITIALIZATION ROUTINE
      IE=INTERRUPT ENABLE

```

Listing 2.


```

6 REM THIS ROUTINE SIMPLY RECORDS ANY RESPONSES WHICH
  OCCUR DURING THE BLUE LIGHT PRESENTATION

30 PRINT D$;"BLOAD DATA STORAGE ROUTINE"
90 EV=PEEK(IP)
100 IF EV=0 THEN 130
110 POKE EA,EV
120 CALL DR

1000 OP=49312:IP=49314
1040 EA=252:DR= -19968

3000 REM EV=EVENT CODE: EA=EVENT CODE ADDRESS FOR THE DATA
      STORAGE ROUTINE: DR=ADDRESS OF THE DATA STORAGE ROUTINE

```

Listing 3.

```

PROCEDURE CRF; (* a continuous reinforcement schedule *)

CONST
  darkreinforce:128; (* hopper presented, houselight off *)
  lightreinforce:196; (* hopper presented, houselight on *)
  lightonly : 64; (* only the houselight on )
  blackout : 0; (* all devices off *)
  greenkey : 19; (* projector A; lamp 3; green stimulus *)
  noresponse : 0; (* no events occurring *)
  leftkeypeck : 1; (* left key switch activated *)

  (* note that the other stimulus combinations *)
  (* are not needed in this example *)

VAR
  currentevent: INTEGER;

PROCEDURE TURNON ( stimulus: INTEGER);

EXTERNAL; (* we use the assembly language version of TURNON *)
FUNCTION EVENT: INTEGER;

EXTERNAL; (* We use the assembly code version of this as well *)

BEGIN (* CRF *)
  turnon (lightonly); (* turn on the houselight *)
  currentevent := noresponse; (* initialize this variable *)
  While currentevent = noresponse DO
    currentevent :=event; (*loop here until an event occurs *)
  If currentevent = keypeck THEN turnon (darkreinforce);
END;

```

Listing 4.

```

TYPE byte: 0..255; (* this defines one byte of memory)
eventdata: PACKED RECORD
  eventtype: byte;
  stimtype : byte;
  starttime : PACKED ARRAY [0..2] of byte;
  endtime : PACKED ARRAY [0..2] of byte;
END; (* of record definition*)

```

Listing 5.