SESSION V APPLICATIONS: HARDWARE

William L. Palya, *President*

An inexpensive experiment controller for stand-alone applications or distributed processing networks

DONALD E. WALTER and WILLIAM L. PALYA
Jacksonville State University, Jacksonville, Alabama

An inexpensive microprocessor-based experiment controller available as a commercially produced bare printed circuit board is described. It provides for up to 8 inputs and 40 outputs. An 8K Experiment Control BASIC (ECBASIC) and a compiled Experiment Control Language (ECL) are also available. Both languages have internal structures that enable time and responses to be dealt with in a very convenient and error-free manner. Time and response count are automatically incrementing variables that need only to be tested by the user program. ECBASIC provides a 10-msec accuracy, whereas ECL provides a 1-msec accuracy. ECL provides extensive support for event-driven programs. The board and software are designed to provide an orderly migration from a stand-alone controller running ECBASIC as a relay-rack replacement, to a network of ECBASIC machines on a small computer, to a true error-checking network of ECL machines on a time-sharing operating system of a large computer. Network communication is over the three wires of a single unmodified RS232 asynchronous serial communication line.

As a computer is used to carry out more of the activities of a laboratory, the demands placed on the functionality of the system increase. The more the computer does, the easier it is to envision additional tasks for it to do. Users quickly come to want fast response times and user-friendly software, while at the same time requiring that the system perform screen editing, database management, data analysis, and graphics. All of these uses are resource intensive and require a relatively sophisticated computer. On the other hand, the lab computer is typically also responsible for the control of experiments. This task is very simple and requires few resources. Unfortunately, as the lab computer evolves toward greater functionality, justifying its use as an experiment controller becomes increasingly difficult. Each cycle that the computer uses to perform the simple task of arranging the contingencies of an experiment diminishes its ability to provide expensive resources to users.

As a result of these divergent niches, it appears reasonable to consider the use of two computers: a

computer optimized for running experiments, which is relatively simple and inexpensive, with few resources; and a machine optimized for general-purpose use, with many resources and which provides a high degree of functionality. Fortunately, two distinct machines can be integrated into a single system that fulfills both needs economically. A network can be formed in which a general-purpose computer passes experiment-control tasks down to inexpensive peripheral processors and provides for the orderly collection and storage of the obtained data. In this type of system, experiment control is relegated to an inexpensive machine, while a computer with many resources provides for general services, such as software development, database management, and analysis. A network of simple experiment controllers imposes negligible overhead on the main general-purpose machine while at the same time providing large-machine support for running experiments. An additional advantage of such a network is that the main computer can be upgraded or changed without a substantial loss of software investment.

We developed a networkable experiment controller capable of functioning in a stand-alone mode. The controller is implemented as an inexpensive, easy-to-assemble printed circuit board. The bare printed circuit board is commercially available from D & D Electronics, Inc.,

The authors gratefully acknowledge the contributions of Elizabeth Palya, Timothy Black, and Robert Swan. Requests for reprints should be sent to William L. Palya, Department of Psychology, Jacksonville State University, Jacksonville, AL 36265.

5175 Moores Mill Road, Huntsville, Alabama 35811, for \$35. Parts cost about \$150. Relays are about \$4 each. The stand-alone configuration provides for a minimum start-up cost and an orderly and affordable increase in functionality. Initially, the controller can be used alone. Later, it can be upgraded and used as a peripheral processor of a general-purpose machine.

Three versions of the controller use the same printed circuit board. They differ only in firmware on the PROM. The simple stand-alone version executes a BASIC tailored for experiment control (ECBASIC), which automatically handles the arduous aspects of collecting inputs and determining real time. The networkable ECBASIC version of the controller executes the same ECBASIC but also provides firmware that enables many controllers to conveniently communicate with a main computer over a single RS232 line. The most powerful version of the controller is the networkable ECL version. It executes a precompiled Experiment Control Language (ECL) that provides a powerful instruction set, automatically collects and reports complete interevent time records, and provides for the convenient use of elapsed time down to 1-msec ticks. The networkable ECL version also provides for true error-checking, packet-based networking. All applicable software is extensively documented, is in the public domain, and is available for the cost of distribution.

Figure 1 depicts the general architecture of the controller and illustrates the additional parts that are to be installed if the simple stand-alone version is desired.

The controller uses a 6809 microprocessor with 16K dynamic memory and 16K PROM memory. The RAM provides space for program and data storage, and the PROM is used to provide nonvolatile storage for the system software ("firmware"). Each version of the controller is on a different 8K PROM. There is also

a programmable timer chip that provides the time base.

A single serial port is used when the controllers are networked. Many controllers can be attached to the same asynchronous RS232 serial port of the main computer. Each controller monitors the data flow over the serial line and loads data only after its unit number is detected. Each controller can respond on the serial line without confusion in that only one controller at a time is given the authorization to respond. The stand-alone controller uses the first serial port to support a CRT terminal and implements an additional serial port for a printer as well as the circuitry to interface a Kansas City Standard audio tape port. The audio tape is for program storage.

The controller provides for eight inputs, which can be either normally open or normally closed contacts. The board also provides for up to 40 outputs, which can be either TTL, 300-mA transistor outputs, or 2-A relay contacts, depending on which components are installed. In addition to switching higher currents, the relays provide isolation and the ability to use arbitrary voltages. The output relays are configured as four groups of eight, with the normally open contacts wired to the edge connector. Each group has an independent common. In addition, one group of eight relays has common, normally open, and normally closed contacts all wired to the edge connector. The circuitry to implement a white-noise generator is also provided. Typically, the board uses approximately 2 A of +5 V, 100 mA of $\pm 12V$, and 100 mA of +5 V for each relay.

The controller has on-board diagnostics. The memory is tested each time the board is powered up, and an internal diagnostic mode can be enabled with a toggle switch. In this mode, each time an input occurs, the next consecutive output is turned on. By activating each input and observing the outputs, an "off-line" test of the integrity of the controller and experimental apparatus can be quickly performed.

STAND-ALONE ECBASIC CONTROLLERS

The stand-alone ECBASIC supports the typical simple BASIC commands. It has 32-bit integer arithmetic (which provides numbers to over two billion), one-letter variables, one-letter two-dimensional arrays, and a maximum subscript of 255. A complete description of ECBASIC is provided in the *ECBASIC Language Reference Manual* (available from the second author).

Table 1 depicts the program commands, monitor commands, and arithmetic operators available with the stand-alone ECBASIC. There are two ways programs can be entered into the stand-alone controller. As is typical of machines that run BASIC, line numbers followed by program commands can be entered from the terminal, or programs that have been SAVED on the tape can be entered with the LOAD command. Programs can be output to the terminal with LIST, to the printer with LLIST, and to the tape with SAVE. Programs are

Figure 1. The major functional units of the experiment controller. The elements depicted with dashed lines are implemented only in the stand-alone version.

Table 1
ECBASIC Commands and Operators

PROGRAM COMMANDS	MONITOR COMMANDS
REM	LOAD
DIM	LIST
INPUT [from terminal]	LLIST
PRINT [to terminal]	SAVE
LPRINT [to printer]	RUN
GOTO	NEW
FOR/NEXT	
READ/DATA/RESTORE	ARITHMETIC OPERATORS
GOSUB/RETURN	+ - * /
IF X THEN command	
[>,<,>=,<=,<,>]	
ON X GOTO Y,Z	32 bit integer
ON X GOSUB Y,Z	one letter variables
STOP	two dimensional arrays
END	maximum subscript 255

executed with the RUN command. The program buffer can be initialized with the NEW command.

Program commands are illustrated in the left column of Table 1. These commands can be entered following line numbers or, where appropriate, can be executed in immediate mode. REM designates a remark or comment statement and is not executed. DIM allocates space for an array. INPUT sets the specified variable(s) equal to the value(s) entered from the terminal. PRINT writes specified information to the terminal, whereas LPRINT writes the information to the printer. GOTO redirects program flow to a specified line number. The FOR/NEXT commands are used to define a loop that is to be executed a specified number of times. The READ/DATA/RESTORE commands are used to set up data within the program and read or draw upon that data as required. The RESTORE command specifies that the next data element is to be drawn from the beginning of the designated DATA statement. GOSUB/RETURN allows the execution of the same block of code from several places in a program. The return statement returns the program to the line number following the GOSUB, rather than to some constant line number. The IF X THEN command allows the conditional execution of a program command, depending on how the "X" expression is evaluated. If the expression is true, then the second half of the line is executed; otherwise, the next consecutive line following the IF THEN command is executed. ON X GOTO Y,Z and ON X GOSUB Y,Z redirect program flow to some line number, depending on how the X expression is evaluated. If X is evaluated as a 1, then the first line number (Y) in the following list is executed; if it is evaluated as a 2, then the second line number (Z) in the list is executed (and so forth). STOP terminates execution, and END is used following the highest line number.

Table 2 illustrates the functions and calls provided by ECBASIC. Functions can be used in any expression. The ABS(X) function will be evaluated as the absolute value of the expression within the parentheses. SGN(X) will return the sign of X. A positive number will be evaluated as a 0; a negative number will be evaluated as a 1. MOD(X,Y) returns the remainder of X divided by Y. PEEK(X) returns the contents of the memory location specified by X. FREE() returns the amount of free memory. RAND() returns a random number

between 0 and 1,000. RANGE(X,Y) returns a random number within the range of X and Y. PROB(X) returns true ("1") with a probability between 0 and 1, as specified by the argument.

TIME(X), or time since X, is an extremely powerful and convenient way to deal with real time. The time since the session started (in 10-msec ticks) is automatically maintained by ECBASIC and requires no explicit program steps to check and update the clock. There are no missing ticks and no cumulative timing errors. TIME(X) can be used in any expression and will be interpreted simply as the time elapsed since the time specified by its argument, with no further program overhead. For example, a variable can be set to the current time [B=TIME(0)], a process called "snapshotting" the clock. Later, program flow can be conditionally redirected depending on the time since B. For example, if TIME(B) is less than 1,000 ticks, then do one block of code (Line 90 and following); otherwise, do the block of code starting on the next consecutive line number following Line 60. The program can therefore execute code while waiting for a particular time.

Calls are used to carry out various activities. CALL TURNON(X,Y,Z) turns on the specified outputs. CALL PULSON(X,Y) turns on the single specified output for Y duration and halts program flow until completed. CALL TURNOFF(X,Y,Z) and CALL PULSOFF(X,Y) are analogous, except that they turn off rather than turn on the outputs. CALL SAVON saves the status of all the outputs, CALL CLRALL turns off all outputs, and CALL RESTON restores the saved condition of all the outputs. CALL POKE(X,Y) deposits byte Y in location X. CALL SLEEP(X) halts program execution for X ticks. Occasionally, it is more convenient to have automatically decrementing timers that can be started and stopped rather than to use the SLEEP(X) or TIME(X) function. CALL STICK(X,A) starts ticker X decrementing variable A from whatever value A had been set to initially. The decrementing occurs

Table 2
ECBASIC Functions and Calls

FUNCTIONS	
ABS(X)	[absolute value]
SGN(X)	[sign]
MOD(X,Y)	[remainder X/Y]
PEEK(X)	[examine memory location X]
FREE()	[amount of free memory]
RAND()	[random number between 0 - 1000]
RANGE(X,Y)	[RAND in range]
PROB(X)	[true p = 0 to 1000]
TIME(X)	[time since X]
	e.g.
	10 B = TIME(0)
	60 IF TIME(B)<1000 THEN GOTO 90
CALLS	
CALL TURNON(X,Y,Z)	[turn on X,Y,Z]
CALL PULSON(X,Y)	[turn on X for duration Y]
CALL TURNOFF(X,Y,Z)	[turn off X,Y,Z]
CALL PULSOFF(X,Y)	[turn off X for duration Y]
CALL SAVON	[store current condition of outputs]
CALL CLRALL	[turn off all outputs]
CALL RESTON	[restores saved output states]
CALL POKE(X,Y)	[at location X deposit byte Y]
CALL SLEEP(X)	[halt for X time]
CALL STICK(X,A)	[start ticker X decrementing variable A]
CALL HTICK(X)	[halt ticker X]
CALL INPUT(X,R)	[start input X incrementing variable R]
	e.g.
	10 INPUT(1,R)
	50 S = R
	60 IF R<S+100 THEN GOTO 90

automatically every 100 msec and does not require any explicit attention once started. CALL HTICK(X) halts the automatic decrementing of variable A.

CALL INPUT(X,R) is an instruction that provides for the convenient handling of inputs. After CALL INPUT(X,R) is executed, each input (from whatever transducer is attached to input X) automatically increments the specified variable, for example, R for response count. No program activity is necessary. At any time, some variable (such as S, or "start count") can be used to snapshot the current value of R (the total number of responses to occur up to that time). The current value of R and the value when R was sampled (S) can then be used in an expression such as IF R<S+100 THEN GOTO 90. The important point to note is that the value of R increases as responses occur, even though nothing in the program is explicitly incrementing the value of R. In this example, if 100 responses have occurred since S, the block of code following Line 90 will be executed; if not, then the block of code beginning on the next consecutive line number following 60 will be executed.

ECBASIC Program Examples

Table 3 provides the major logic for example programs that would implement several typical schedules of reinforcement. The first example provides a loop that would implement a fixed-ratio 100 (FR 100). On Line 10, S (start count) is used to snapshot the current value of P (pecks). Line 20 then determines if the current number of pecks (P) is less than 100 more than the start count (S). If 100 pecks have not occurred since Line 10, then Line 20 is executed again. When $P \geq (S+100)$, then Line 30 is executed. This line provides the reinforcer. It turns on output 3 (magazine and magazine light), halts program flow for 400 ticks

Table 3
Example ECBASIC Logic

```

FIXED-RATIO 100
10 S=P
20 IF P<(S+100) THEN GOTO 20
30 CALL PULSON(3,400)
40 GOTO 10

VARIABLE-INTERVAL t
10 CALL SLEEP(V(RANGE(1,25)))
20 S=P
30 IF P=S THEN GOTO 30
40 CALL PULSON(3,400)
50 GOTO 10

SECOND ORDER FR 5(FR 50:SP)
10 FOR I=1 TO 5
20 S=P
30 IF P<(S+50) THEN GOTO 30
40 CALL PULSON(2,25)
50 NEXT I
60 CALL PULSON(3,400)
70 GOTO 10

CHAIN VI t FR 100
10 CALL TURNON(5)
20 CALL SLEEP(V(RANGE(1,25)))
30 S=P
40 IF P=S THEN GOTO 40
50 CALL TURNON(5)
60 CALL TURNON(6)
70 S=P
80 IF P<(S+100) THEN GOTO 80
90 CALL TURNON(6)
100 CALL PULSON(3,400)
110 GOTO 10

```

(4 sec), and then turns output 3 back off again. The program is then directed to Line 10. The current total number of responses (P) is set into the start count (S), and a new FR is started.

The second example implements a variable-interval (VI) schedule. Suppose that earlier in the program an array called V was defined with 25 VI times (e.g., Fleshler & Hoffman, 1962). Line 10 directs the program to halt execution for a duration. The duration is specified by one of the elements in the array of VI values. The specific one is chosen at random by the RANGE (1,25) function. After that duration elapses, the program snapshots the current number of responses (P) with the start count (S). Then Line 30 is reexecuted until a response occurs, in that P will equal S until after a response occurs. Line 40 then presents a 4-sec reinforcer. The program then returns to Line 10, chooses another VI time at random, and waits for that duration.

The third example is a second-order schedule. In this schedule, five FR 50s are required (FOR I=1 to 5), each followed by a 250-msec stimulus [CALL PULSON (2,25)]. The last of the FR 50s is followed by food [CALL PULSON(3,400)]. The final example in this figure is a chain schedule. It simply concatenates a VI schedule and an FR 100 and adds appropriate keylights.

Table 4 provides a complete runnable program that implements Dews's (1962) interrupted fixed-interval (FI) schedule. In his procedure, each session contained 11 FI schedules. Each interval was partitioned into 10 bins, and alternate bins had the houselight out.

The sample program is set up to store the number of pecks that occur in each of the 10 bins for each of the 11 FIs. In addition, it stores the elapsed time for each interval. This data is stored in an 11 x 11 data array. That array is dimensioned on Line 50. Line 60 sets the pecks counter to zero. Line 70 specifies that variable P is to be incremented each time a response on Key 1 occurs. Line 80 sets up the loop that will carry out the 11 FI cycles. Line 90 turns on the keylight. Line 100 snapshots the time the interval started in the 11th data element for that cycle. Line 110 sets up the loop that will step through the 10 bins. Lines 120 and 130 determine whether the bin is odd or even and sets the houselight accordingly. Line 140 obtains a start value for the number of pecks and snapshots it into a handy temporary storage location (the data array element for this iteration, e.g., cycle 1 bin 1). The program flow then halts for 50 sec. It then subtracts the start value of the peck counter from the current value of the peck counter (i.e., it determines how many pecks occurred in the last 50 sec) and sets the proper data element to that value (e.g., cycle 1 bin 1 could be set equal to 10). Line 170 then returns to Line 110 to conduct the next bin. If 10 pecks had occurred in cycle 1 bin 1, then Line 140 would set the data element for cycle 1 bin 2 equal to 10. If following the subsequent 50 sec a total of 30 pecks had occurred since the beginning of the session, Line 160 would determine that the

Table 4
ECBASIC Implementation of Dews (1962)

```

10 REM DEWS(1962)   WRITTEN IN ECBASIC
20 REM OUTPUT 1= HOUSE LIGHT
30 REM OUTPUT 3= MAGAZINE
40 REM OUTPUT 5= RED KEY LIGHT
50 DIM D(11,11)
60 P=0
70 CALL INPUT(1,P)
80 FOR C= 1 TO 11
90   CALL TURNON(5)
100  D(C,1)=TIME(0)
110  FOR B= 1 TO 10
120   IF MOD(B,2)=1 THEN CALL TURNON(1)
130   IF MOD(B,2)=0 THEN CALL TURNON(3)
140   D(C,B)=P
150   CALL SLEEP(5000)
160   D(C,B)=P-D(C,B)
170  NEXT B
180  S=P
190  IF P=S THEN GOTO 190
200  D(C,11)=TIME(D(C,11))
210  CALL CLRALL
220  CALL PULSON(3,300)
230 NEXT C
240 FOR C= 1 TO 11
250   LPRINT "FI ";C
260   FOR B= 1 TO 10
270    LPRINT " "; D(C,B);
280   NEXT B
290   LPRINT "TIME"; D(C,11)
300   PRINT
310 NEXT C
320 END

```

final value for cycle 1 bin 2 would equal 20. This process would be continued until the 10th bin of the first interval was completed. Lines 180 and 190 would wait for the first peck. Line 200 would log the time since the beginning of the cycle for that interval in data element cycle 1 bin 11. Lines 210 and 220 would carry out the reinforcement cycle. Line 230 returns to Line 80 for the next cycle.

After the 11 intervals, the data would be dumped out. Line 240 sets up the loop to iterate through the 11 cycles. Line 250 prints out a header with the letters "FI" followed by the number of the cycle to be printed. Then Lines 260, 270, and 280 print out the 10 bins of that interval. Line 290 prints the word "TIME" followed by the elapsed time for that interval. Line 310 returns to Line 240 to dump the information for the next interval.

NETWORKED ECBASIC CONTROLLERS

As is illustrated in Figure 2, several stand-alone controllers could share the same terminal, printer, and audio tape by using simple toggle switches or a rotary switch to connect them to the desired controller. In this way, multiple experiments would have almost total hardware independence, without the cost of additional terminals and printers.

Alternatively, program development and storage can be done with a small computer, such as an Apple or a TRS-80. We made some minor modifications in the firmware of the stand-alone BASIC to enable limited networking. Figure 3 illustrates a network in which all communication of programs to, and data from, the controllers is through a standard asynchronous RS232 serial port of the main computer. Simply put, the computer acts as if it were the terminal connected to the controller.

Programs would be written in ECBASIC using the editor of the main computer, and subsequently would be stored on its disk. A program could then be loaded into

a particular controller by issuing that controller's enable code to the network serial port as if it were being printed to a terminal. The appropriate controller would then recognize further transmissions on the RS232 line as input, just as if it were being entered from a terminal. The controller would be loaded with the desired program by directing the appropriate disk file to the network serial port as if it were being printed to a terminal.

Data are retrieved from a controller by issuing the desired controller's enable code to the network port. If that controller has data, it will respond by sending data as if it were printing the data to a terminal. The main computer need only read the serial line and store the obtained information on the disk as if a terminal were generating the data. Actually, the same command that enables the controller to receive a program also dumps data from the controller. This is because the networkable ECBASIC interpreter halts execution when it encounters a PRINT command, and waits to receive its enable code from the main computer. After the controller receives its code on the serial line from the main computer, it continues execution. When the data-dumping program in the controller terminates, ECBASIC returns to the "READY" state during which it can receive normal ECBASIC commands, such as NEW to initialize the program buffer and line numbers followed by program commands to enter a program.

The main computer can issue three commands to the networkable ECBASIC controllers. The connect packet links the main computer to a specific controller. This packet first initiates the data-output portion of the

Figure 2. A schematic diagram illustrating how a terminal, printer, and audio tape could be shared by several controllers.

Figure 3. A schematic diagram illustrating how several controllers are attached to the same RS232 asynchronous serial line of the main computer in the networked ECBASIC and networked ECL version of the controller.

program in the controller. The controller then returns to the "READY" state. When in this state, any monitor or program command can be sent to that controller. The break packet is the second command. It unconditionally terminates execution of an ECBASIC program in the controller and returns ECBASIC to the "READY" state. The last command is the disconnect packet; it disconnects a controller from the main computer.

NETWORKED ECL CONTROLLERS

The third version of the controller was designed to provide a more powerful control language than BASIC, timing accuracies to 1 msec, and the ability to automatically record and store the time of each event that occurs throughout the experiment. Communication between the main computer and the controller is a true error-checking, packet-based network using an unmodified asynchronous serial line.

The experiment-control language, ECL, is compiled in order to provide faster execution than is available with BASIC. The language is compiled to an intermediate code that is then executed by firmware on the controller. This intermediate code approach was used for the same reason that many Pascal compilers produce "P" code; it made the language easier to implement. Automatic recording of every interevent time provides the capability to dynamically analyze the data. As a result, the data can be

analyzed with respect to interpretations that codify only after the experiment is completed. The controller was designed to communicate over the three wires of a standard RS232 asynchronous communication line so that unmodified handlers available with commercial time-sharing operating systems could be used to communicate with the controllers. This allows the experiment controllers to be used in a typical time-sharing environment of a general-purpose computer, such as a PDP-11 running RSX or a VAX running VMS or UNIX. Additionally, using a standard asynchronous serial line minimizes the wiring necessary to support remote experiments. Many controllers can be attached to the same three wires. A packet-based error-checking network was developed to assure minimal data loss by providing virtually error-free communication.

The networkable controller allows one large time-sharing machine to be used for all software development, database, and analysis, while the actual control of the experiment is implemented on inexpensive special-purpose computers. Programs are created by invoking the EDITOR of the main computer and entering the program from a terminal. The program is written in ECL and is stored on the disk. That source program is then compiled into a run module, which can be executed by the experiment controllers. The resulting runnable program is stored on the disk.

All communication between the main computer and the controllers is handled by a program that can run as a foreground job or as a task under a time-sharing operating system of the main computer. Controllers are loaded with a compiled ECL run module by simply specifying to the communication module which program is to go to which controller. The same ECL module can be loaded into several controllers if the subjects receive the same treatment, in that all controllers function identically. The communication module also automatically retrieves and stores the data from the controllers as their buffers fill. It also obtains status information such as "reinforcers remaining." Status information can be requested by any terminal on a time-sharing system.

The data obtained from the controllers are stored on the disk. If complete interevent time records were obtained, a FORTRAN library, called SELLIB, is available to provide the software tools to conveniently search for and collect any specifiable data from the data stream.

ECL Program Commands

The ECL programming language was designed to provide powerful and easy-to-use constructs to carry out typical experiments. As can be seen, many instructions are analogous to those available in ECBASIC. However, occasionally compromises were made so that the program would run faster or so that the code could be more easily compiled. ECL provides extensive support for event-driven programming; however, these features will not be presented in this paper. A detailed

description of the entire language is provided in the *ECL Language Reference Manual* (available from the second author).

ECL is a free-form language similar in form to Pascal. Statements need not occupy specific columns. As a result, programs can be formatted for maximum readability. All that is required is that each command line be terminated with a semicolon. In ECL, each variable or array must be explicitly defined, and all variables are global. Variable names can be up to six characters, and the maximum subscript is 255. All variables are 32-bit integers.

Input/output commands. Table 5 summarizes the input/output commands available in ECL; INPUT specifies which hardware input (1-8) is to increment which variable, whether the input is a contact make or break, the debounce time, and whether or not the response is to be automatically recorded. The variable specified in INPUT can subsequently be used in expressions. For example, while the pecks are less than some number past the starting value, one block of code can be executed, and when past that value, some other block of code can be executed.

The COMBINE command ties together outputs so that activating any one output activates them all. For example, the magazine and magazine light could be combined: Operating the magazine would subsequently also operate the magazine light.

CALL TURNON turns on the specified output. CALL PULSON turns on an output, halts program flow for a duration, and then turns the output off. For

example, reinforcement could be implemented with CALL PULSON (rein,4000). CALL TURNOF turns off the specified output. CALL PULSOF turns off an output for a duration.

CALL SAVON saves the status of all the outputs. CALL CLRALL turns off all outputs, and CALL RESTON restores the SAVED condition of all the outputs. These instructions are useful when turning off all outputs for a short duration, such as during reinforcement or time out. CALL WAITIN halts program flow for a specified time or until a specified input. It returns the initiating event as the value of a variable. For example, the instruction CALL WAITIN(a,timlmt,key1, key2) will wait until the time specified by "timlmt" or an input on "key1" or an input on "key2." When one of those events happens, program flow is continued. The value of "a" can then be used to direct program flow accordingly. The instruction CALL CONC is basically WAITIN optimized for running concurrent schedules. This instruction falls through when "key1" occurs after "time1," or "key2" after "time2," or "key3" after "time3." A changeover delay (COD) can be specified in the instruction. The value of "a" can then be used to direct program flow.

CALL OUTPUT puts a data item into the data stream to be automatically collected by the communication module in the main computer. For example, CALL OUTPUT(data[i]) can be used to pass an element of a data array to the main computer. CALL MARKER puts a marker into the data stream. For example, CALL MARKER(42) puts Marker 42 into the data stream. This marker can subsequently be used to indicate to the data-analysis program that the second link of the schedule had been entered at that point in the data stream.

Randomizing functions. ECL also provides random-number functions, all of which can be used in expressions. Table 6 illustrates these commands. RAND returns a random number between 0 and 1,000. RANGE returns a random number within the specified range. For example, a random set of waits from 1 to 10 sec could be implemented with SLEEP(RANGE(1,10)*1000). With PROB, a path can be set to occur with a specified probability. In the example in Table 6, half the time (500 designates $p = .005$) the block of code following the THEN will be executed, and half the time the code following the ELSE will be executed.

Timing commands. Several commands are provided for timekeeping; these are shown in Table 7. CALL SLEEP halts program flow for a specified time. For example, CALL SLEEP(1000) halts program flow 1 sec. Variables that can be set to automatically decrement every 100 msec are convenient to implement time bases that run only during a portion of the schedule. For example, a VI value that is to run during some parts of the schedule can be easily implemented with a ticker. CALL STICK starts a ticker decrementing a specified variable from its initial value. This process is automatic and does not require program intervention. The variable

Table 5
Input/Output Commands

INPUT	Specifies: Input / variable / record Make or break / debounce time Subsequently variable automatically increments. e.g., INPUT 1,1,+,pecks1; strtpk = pecks1; WHILE (pecks1 .LT. (strtpk+100));
COMBINE	Defines output group.
CALL TURNON CALL PULSON	Turns on output. Turns on output for a duration. e.g., CALL PULSON(rein,4000);
CALL TURNOF CALL PULSOF	Turns off output. Turns off output for a duration.
CALL SAVON CALL CLRALL CALL RESTON	Store current condition of outputs. Turn off all outputs. Restores saved output states.
CALL WAITIN	Waits for time or input. "a" = initiating event. e.g., CALL WAITIN(a,timlmt,key1,key2); IF (a .EQ. 0) THEN GO TO label1; IF (a .EQ. 1) THEN GO TO label2; IF (a .EQ. 2) THEN GO TO label3;
CALL CONC	Waits for first of time and input conjunction. Has specifiable COD. "a" = initiating event. e.g., CALL CONC (a,COD, key1, time1, key2, time2, key3, time3); IF (a .EQ. 1) THEN GO TO label1; IF (a .EQ. 2) THEN GO TO label2; IF (a .EQ. 3) THEN GO TO label3;
CALL OUTPUT	Puts data item in data stream. e.g., CALL OUTPUT(data[i]);
CALL MARKER	Puts a marker in data stream. e.g., CALL MARKER (42);

Table 6
Randomizing Functions

RAND	A random number between 0 and 1000
RANGE	A random number in a range e.g. SLEEP (RANGE(1,10)*1000);
PROB	A probability between 0 and 1000 e.g. IF (PROB(500)) THEN BEGIN; ----- END; ELSE BEGIN; ----- END;

Table 7
Timing Commands

CALL SLEEP	Waits for specified time e.g. CALL SLEEP(1000);
CALL STICK	Starts a ticker automatically decrementing a variable e.g. CALL STICK(1,vitime);
CALL HTICK	Halts the specified ticker e.g. CALL HTICK(1);
TIME	Time since specified variable e.g. strttm = TIME(0); IF(TIME(strttm) .LT. 1000) GO TO link1;

can subsequently be used to make decisions based on time. CALL HTICK halts the specified ticker. TIME is equivalent to the same instruction available in ECBASIC. TIME will return a time difference between the current time and any expression. For example, if the elapsed time since the start time is less than 1,000, a special block of code could be executed; otherwise, the next consecutive instructions would be executed. Real time cannot be lost and cumulative timing errors cannot occur with this instruction, in that it represents the total ticks since the beginning of the session. SLEEP halts program flow. In contrast, a time duration implemented with a ticker or with TIME need not "lock up" the program.

Program flow commands. ECL has a variety of program flow commands. They are illustrated in Table 8. LABEL provides a symbolic destination for program flow. GOTO redirects program flow to a label. SUBROUTINE defines a user subroutine as all the code between the subsequent BEGIN and END. CALL can be used to redirect program flow to a user subroutine. For example, CALL observ executes the code in the observe subroutine and then returns to the line following the CALL statement. WHILE reexecutes the same line, or executes all statements between its BEGIN and END as long as its condition is nonzero. FOR executes a block of statements a number of times. It is essentially equivalent to a FOR/NEXT loop in BASIC. For example, FOR I=1,10 executes the code following its semicolon, or the code within its BEGIN END boundaries, 10 times. The IF statement conditionally executes a block of statements. If the condition is true, the THEN block is

executed. If it is false, the next consecutive instruction is executed or the ELSE block is executed if an ELSE is listed. CONTINUE forces the next loop iteration in a FOR or WHILE loop. BREAK forces an exit from a FOR or WHILE loop. CALL STOP terminates the program.

Variables and operators. Table 9 shows examples of variable definitions and lists the available operators. Variables must be allocated or defined at the beginning of a program with either the VARIABLE or the WORD statement. They can be simply initialized or can be set to a value. DEFINE can be used to define a symbol as a constant. This clarifies programming by providing names for things that are actually numbers. Output 20 can be defined as red, and "red" subsequently can be turned on.

ECL has an expression evaluator; therefore, variables can be set equal to expressions at any time. Expressions can be any combination of variables, array elements, constants, functions, and arithmetic operators.

Typical arithmetic, logical, shift, comparison, and relational operators are available.

ECL Program Examples

Table 10 provides examples of the major logic necessary to implement typical schedules of reinforcement in ECL. The first three are the same as those implemented in ECBASIC (Table 3). The differences between Table 10 and Table 3, therefore, are the differences in the characteristics of the languages.

An FR 100 is implemented in essentially the same way as in ECBASIC. In the first line, the variable strtpk is used to snapshot the peck count. Then, as long as the

Table 8
Program Flow Commands

LABEL	Defines a symbolic destination for program flow
GO TO	Redirects program flow to LABEL e.g. GO TO link2; link2:CALL TURNON (red);
SUBROUTINE	Defines a user subroutine between BEGIN and END
CALL	Calls a user subroutine e.g. CALL observ; SUBROUTINE observ; BEGIN; ----- END;
WHILE	Executes a block of statements while non-zero e.g. WHILE (pecks.LT.100); WHILE (pecks.LT.100) BEGIN; ----- END;
FOR	Executes a block of statements a number of times e.g. FOR I=1,10; array[I]=0; FOR I=1,10; BEGIN; ----- END;
IF	Conditional execution of blocks of statements e.g. IF(a=b.EQ.c)THEN GOTO tmout; IF(TIME(X).GT.1000) THEN BEGIN; ----- END; ELSE BEGIN; ----- END;
CONTINUE	Force next loop iteration
BREAK	Force exit from loop
CALL STOP	Terminates program

Table 9
Variables and Operators

VARIABLE	Defines a variable or array		
WORD	Defines a variable or array e.g. VARIABLE a; WORD time = 4000; VARIABLE array[10];		
REINF	Defines a symbol as a constant. Can be used in expressions. e.g. DEFINE red = 20; CALL TURNON (red);		
VARIABLE = Expression;	Expression can be any combination of variables, array elements, constants, functions, arithmetic operators.		
+ add	& logical AND	< shift left	.EQ. 1 if =
- subtract	! logical OR	> shift right	.NE. 1 if ≠
* multiple	~ logical XOR		.LT. 1 if <
/ divide	! complement		.GT. 1 if >
			.LE. 1 if ≤
			.GE. 1 if ≥

pecks are less than the start count plus 100, the WHILE is reexecuted. When the condition becomes false, reinforcement is provided and the loop is restarted. The VI example halts program flow for a duration specified by the element chosen from the "vival" array. Then the program flow is halted again until an input on "key1" occurs (the zero time argument indicates that time is not to be considered an argument). This is followed by reinforcement and a return to the beginning of the loop. The second-order schedule uses a FOR loop to iterate five cycles of the code within the BEGIN and END statements. A concurrent schedule is illustrated in the last example. Program flow waits for "key1" after a VI 60-sec time duration or "key2" after a VI 90-sec time. A 1.5-sec COD is in effect. When either of the conditions is met, program flow continues. The tickers that automatically decrement the two VI values are halted (HTICK) while reinforcement occurs. Then the VI that

Table 10
Example ECL Logic

```

FIXED-RATIO 100
start: strtpk = pecks;
      WHILE (pecks .LT. (strtpk + 100));
      CALL PULSON (rein,4000);
      GOTO start;

VARIABLE-INTERVAL 1
start: SLEEP (vival[RANGE(1,25)]);
      CALL WAITIN (a,0,key1);
      CALL PULSON (rein,4000);
      GOTO start;

SECOND ORDER FR 5(FR 50:5P)
start: FOR I = 1,5;
      BEGIN;
        strtpk = pecks;
        WHILE (pecks .LT. (strtpk + 50));
        CALL PULSON (tone,250);
      END;
      CALL PULSON (rein,4000);
      GOTO start;

CONCURRENT VI 60 s VI 90 s
Start: CALL CONC (reply,1500,
                key1,v160v1,
                key2,v190v1);
      CALL HTICK (1);
      CALL HTICK (2);
      CALL PULSON (rein,4000);
      IF (reply .EQ. 1)
      THEN v160v1 = v160[RANGE(1,25)];
      ELSE v190v1 = v190[RANGE(1,25)];
      CALL STICK (1,v160v1);
      CALL STICK (2,v190v1);
      GOTO start;

```

was satisfied is identified by testing the value of "reply," and a new VI value is drawn to replace it. The tickers are then restarted (STICK), and the program returns to CONC.

Dews's (1962) interrupted FI schedule is implemented in ECL in Table 11. This program is complete and runs. All data are automatically recorded, and therefore data-storage instructions are absent. The four definition statements are purely for program clarity. They allow variable names to be used when the argument should actually be a number. The VARIABLE pecks,bin,a command specifies that "pecks," "bin," and "a" will be used as variables. The INPUT line specifies that the contact break (+) of hardware Input 1 is to have a 20-msec debounce and is to be used to automatically increment the variable "pecks." The record option specifies that the time of every peck is to be recorded and put into the data stream for automatic retrieval by the communication module in the main computer. The communication module requests the number of reinforcers (REINF) when it loads a program. This variable can then be used to terminate the experiment after the appropriate number of reinforcers. The WHILE command allows for the convenient termination of the program when the number of reinforcers entered by the experimenter at load time is reached. The main program is therefore contained within this BEGIN END loop. The keylight is then turned on, and that event is recorded. Then, a 10-bin loop is set up. On odd iterations, the houselight is turned off, and on even iterations, the houselight is turned on. Whichever event occurs is recorded. Program flow then halts for 50 sec, after which the next bin is implemented. After 10 bins occur, program flow is halted until a peck occurs. This is followed by 3 sec of food presentation, which is recorded. REINF is decremented, and program flow is returned to BEGIN for the next loop through the cycle.

COMMUNICATION SOFTWARE

Communication between the main or host computer and the experimental controllers is a true packet-based network with error checking. The main computer is the network master and initiates all network traffic. Messages are sent in packet form and either are addressed to the desired experimental controller or are simply replies to the main computer. Figure 4 illustrates the communication protocol.

Programs can be transferred from the main computer to the controller with a "LOAD" packet. The controller replies with an "ACKNOWLEDGE" (ACK) packet if everything occurred correctly or a "NEGATIVE ACKNOWLEDGE" (NAK) packet if an error occurred. A program in the controller can be initiated with "START," and again the controller replies with "ACK" or "NAK," depending on whether or not an error occurred. The main computer can issue "TRANSFER," which requests "DATA" from the controller. After receiving the re-

Table 11
ECL Implementation of Dews (1962)

```

DEF rein = 1; /* DEWS(1962) */
DEF hselte = 2;
DEF redkey = 3; /* written in ECL */
DEF key1 = 1;
VARIABLE pecks, bin, a;
INPUT 1,1,+,pecks,RECORD,,20;
WHILE (REINF .GT. 0)
  BEGIN;
    CALL TURNON (redkey,RECORD);
    FOR bin = 1,10;
      BEGIN;
        IF (MOD(bin,2) .EQ. 1)
          THEN CALL TURNON (hselte,RECORD);
        ELSE CALL TURNON (hselte,RECORD);
        CALL SLEEP (50000);
      END;
    CALL WAITIN (a,0,key1);
    CALL CLCALL;
    CALL PULSON (rein,3000,RECORD);
    REINF = REINF + 1;
  END;
CALL STOP;

```

Figure 4. The communication protocol between the main computer and the addressed controller on the serial line. Arrows indicate direction of transmission.

requested data, the main computer then replies with "ACKNOWLEDGE" if no error occurred. This in turn allows the controller to delete the data it has just transmitted. If an error had occurred, the buffer is not deleted and it is retransmitted. If no data were avail-

able, the controller replies to "TRANSFER" with "NO DATA." The main computer can issue "POLL," which requests a "STATUS" packet from the addressed unit. The communication module can terminate a program in the controller by issuing a "KILL" packet.

DATA-SELECTION UTILITY

The SELECT subroutine library provides an easy-to-use, pointer-based, search facility for post hoc analysis of interevent time data. These utilities are similar to those available in word processors for finding text elements with specifiable characteristics, in that the SELECT subroutines process the data stream in much the same way that word processors process a text stream. With these subroutines, a user can dynamically search for (and base calculations on) any specifiable event, set of events, or events conditional on other events contained in the archived data stream of an experiment. In addition to simply cumulating the number of pecks in particular stimuli, conditional events can be selected. For example, the mean latency to the third peck in the blue stimulus following a red stimulus that is followed by no responding to the subsequent green stimulus can be easily determined. The pointer is directed to move forward through the data stream until it encounters the code indicating the onset of a red stimulus; it then determines whether the next stimulus is blue; if it is, then the time to the third peck is stored as a temporary variable. If no pecks occurred in the following green stimulus, then that latency is used in the calculation of the mean latency. If at some later time this measure does not prove useful, then the original data could be reanalyzed to select some better measure. For example, the latency to the point at which 1% of the total responding in the interval occurred could be used in place of the latency to the third response.

REFERENCES

- DEWS, P. B. (1962). The effect of multiple S^{Δ} periods on responding on a fixed interval schedule. *Journal of the Experimental Analysis of Behavior*, 5, 369-374.
- FLESHLER, M., & HOFFMAN, H. S. (1962). A progression for generating variable-interval schedules. *Journal of the Experimental Analysis of Behavior*, 5, 529-530.