Behavior Research Methods, Instruments, & Computers
1989, 21 (2), 294-302

The MED-PC experimental apparatus programming system

THOMAS A. TATHAM
Temple University School of Medicine, Philadelphia, Pennsylvania

and

KARL R. ZURN
MED Associates, Incorporated, East Fairfield, Vermont

MED-PC is a software system that implements the MEDSTATE NOTATION dialect of state notation on IBM PC and compatible computers equipped with MED Associates interfacing. It provides a programming environment in which users can write short programs in a specialized language to control and record the events of operant and classical conditioning experiments. As many as eight experimental stations, each with up to 8 inputs and 32 outputs, running the same or different experimental procedures, may be active simultaneously. The system provides a standard set of run-time features, including mechanisms for displaying real-time data, simulation of responses, manipulation of array and variable contents, and writing of disk files. The system is based upon polling techniques, and is implemented as a translator that generates Pascal units, which are then linked to previously compiled Pascal routines.

MED-PC is a programming environment that implements the MEDSTATE NOTATION (MSN) dialect of state-notation language developed for the real-time control of operant and classical conditioning experiments. It runs on an IBM PC, AT, 386, or compatible computer with 640K RAM. An appropriately matched coprocessor and hard disk are highly recommended. Since the language is implemented with a translator written in Turbo Pascal, it will be relatively easy to port the system to the next generation of microprocessors. The system, which operates on a single computer, has the ability to control as many as eight independently functioning experimental stations (each running the same or different experimental procedures) simultaneously, with up to 8 inputs and 32 outputs per station.

There are three primary advantages derived from using state notation to describe and implement psychological experiments: (1) state notation produces an unambiguous description of experimental procedures, (2) the structure of the language is similar to the way psychologists describe experiments, and (3) many workers are familiar with state notation's syntax (Leslie, 1981; Snapper, 1973; Snapper, Kadden, & Inglis, 1982).

The concepts used in state-notation languages were first advanced in a paper by Mechner (1959), in which he presented a flow-chart system for describing behavioral experiments. Moore (1956) also developed certain key state-notation concepts (Snapper, 1973). Of the three main state-notation systems implemented on minicomputers (ACT by Millenson, 1971; SKED by Snapper, Knapp, Kushner, & Kadden, 1967; SCAT by Stadler, 1969), only SKED-11 is still readily available. SCAT is no longer available, and ACT is only available to a limited extent from BRS/LVE, Incorporated. Several other minicomputer versions of state notation have appeared, including those of Elsner (1982), Gilbert and Rice (1978, 1979), Takigawa and Mino (1981). These, however, have not seen wide use outside of the authors' laboratories. Butler (1980) and Orr (1984) each produced microcomputer versions of state notation, however, these codes were limited in their syntax and controlled only one operant conditioning chamber. Lucas (1986) implemented a relatively complete version of state notation that controlled four operant conditioning chambers, but this version was limited by providing a resolution of only 100 msec, and complete commercial hardware interfacing was unavailable. Our implementation of state notation attempts to provide an inexpensive microcomputer-based system to control up to eight chambers with complete, commercially available hardware interface. In addition, it has a user-definable temporal resolution of 10 to 50 msec, depending on the microcomputer used. Our system also provides an extended version of state-notation language. More than 50 such systems are presently in use in laboratories.

The authors gratefully acknowledge Donald A. Overton for early discussions on the feasibility of this project, Barbara Wanchisen for providing her laboratory as an alpha testing site for MED-PC's algorithms, Mount Union College for providing access to its psychology laboratory, and David Olton, Jay Kaufman, and Michael Davison for their patience, support, and suggestions for enhancements to MED-PC. Requests for reprints may be sent to Thomas A. Tatham, Department of Psychology, Weiss Hall, Temple University, 13th and C. B. Moore Ave., Philadelphia, PA 19122. Requests for product information and reprints should be sent to Karl R. Zurn, MED Associates Inc., Box 47, East Fairfield, VT 05448.

DESIGN OBJECTIVES

MED-PC was developed as a commercial product to satisfy the following criteria: (1) implementation on commercially available microcomputers; (2) implementation with commercially available interfacing; (3) a total cost for computer(s), interfacing, and software affordable to small academic departments; (4) rapid, flexible programming of novel and standard conditioning paradigms; (5) accommodation of user extensions; and (6) a reasonable development time for actually implementing the system. These criteria are generally similar to those described by other developers of conditioning software (Chayer-Farrell & Freedman, 1987; Kaplan, 1985a).

The first two criteria emphasize basing the system on commercially available hardware. Excellent "homebrew" hardware, most notably the ECBASIC controller, has been previously described in this journal (Palya, 1988; Walter & Palya, 1984). However, for many researchers, the labor and technical expertise required to construct and maintain custom equipment is beyond their means. Furthermore, many of these researchers already have access to commercial microcomputers and /or interfacing, and would like to avoid investing in alternative technology.

Specifically, we chose the PC (used loosely to refer to any computer that operates under the MS-DOS operating system with an 8088, 8086, 80186, 80286, or 80386 microprocessor) as the host computer because it is already available in many laboratories, its performance/cost ratio has been steadily increasing, and it seems likely that it will continue to be available for many years to come. After conceiving MED-PC, it was necessary to choose the appropriate interfacing. The first author chose MED Associate's hardware because of their excellent reputation for providing technical support and the moderate cost of their popular equipment. Collectively, a PC and interfacing to control four two-lever operant conditioning chambers can be obtained for less than $3,000, which satisfies Criterion 3.

Criteria 4 and 5 were satisfied by implementing a dialect of state notation. Most people, even those with minimal prior programming experience, can rapidly learn to program in state notation, and can quickly progress from writing simple, to very complex programs (Butler & Grisham, 1977). State notation provides programmers with high-level commands that insulate them from the low-level details of switching outputs, sensing inputs, and timing that can often intimidate novice programmers. In order to facilitate user extensions, we extended the syntax of state notation to accommodate user-written in-line statements and procedure calls embedded in the output section of MSN statements. User-written procedures must be nominally written in Pascal (upon which MED-PC is based), but code written in Assembler, C, or Prolog may, in turn, be linked to user-written Pascal procedures.

The goal of rapidly developing a system that met Criteria 1 through 5 influenced our decision to base the system on Pascal. Strictly speaking, MED-PC is not a compiler, but rather a translator that translates MSN source programs into Pascal code. The resulting code is then linked to a library of compiled routines that control inputs, outputs, screen displays, and timing. Implementing this system as a translator (as opposed to a true compiler), presumably reduced system development time. However, even implementing the system as a translator was very time-consuming; the first author spent approximately 1,500 h developing MED-PC prior to its commercial release.

A variety of languages are well suited for implementing real-time control systems, including Pascal, C, Forth, and Modula-2 (Balsam, Deich, O'Connor, & Scopatz, 1985). However, we chose Turbo Pascal because it produces tight, fast code, and the compiler (which users must purchase separately) is inexpensive (approximately $100). Pascal is also relatively easy to learn, and is taught at most academic institutions; users who wish to program their own extensions should find it easy either to become or to obtain a suitable programmer.

MED-PC FEATURES

The MEDSTATE Dialect of State Notation

MED-PC is based on the MSN dialect of state notation. MSN is a hybridization of state-notation dialects to which new language elements and features have been added. Figure 1 lists several MSN commands, including many language elements not commonly implemented in state notation.

Installation

The MED Associates interface chassis holds up to 16 input and/or output cards (each with eight channels), but the number may vary according to the user's needs. An installation program, INSTATE, is used to create a system configuration file name MEDRTM.PAS. The query-driven installation process, which takes about 5 min to complete, also partitions the input and output cards into a series of logical boxes. Individual boxes may have a maximum of 8 inputs and 32 outputs, subject to the limitation that the system may contain a maximum of 128 combined inputs and outputs. Boxes need not have identical configurations; one box may have 2 inputs and 6 outputs, while another box may have 3 inputs and 16 outputs. During installation, the user also selects the polling rate (determined by the computer's processor speed) and options that pertain to data storage.

A particularly interesting feature of the system is the SHOW command. SHOW allows each box to display the value of as many as six variables, along with their descriptive labels. This feature facilitates the monitoring of experimental sessions. Another useful feature is the addition of constants so that outputs and inputs may be referenced by a descriptive label, rather than by a number. For example, "5#R ^LEFTLEVER: ON ^FEEDER → S2" is considerably more self-documenting than "5#R 1: ON 3 → S2". All variables in MSN programs are im-

Command	Example	Comments on Example
	DATA DEFINITION COMMANDS	
DIM	DIM A=100	Declare an array named A with elements 0..100.
LIST	LIST B = 1",2",3"	Declare array B with B(0)=1", B(1)=2", B(3)=3"
^	^FEEDER = 1	Declare a constant named FEEDER with a value of 1.
#R	5#R1:ON ^FEEDER--->S2	After 5 responses on input 1 activate feeder and goto S2.
	INPUT COMMANDS	
#START	#START:ON ^LIGHTS--->S2	Activate lights after keyboard start command is issued.
#T	V #T ---> S2	Goto S2 after the amount of time specified by V.
#Z	#Z1--->S2	If synchronization pulse 1 is received goto S2.
"	2":OFF ^HOPPER ---> S2	After 2" deactivate hopper and goto state 2
'	60'--->STOPABORT	End session and save data after 60'.
!	60" ! 5#R1 ---> S2	Goto S2 after 60" OR 5 responses on input 1.
	OUTPUT COMMANDS	
ADD	#R1: ADD C --->S2	Increment variable C after every response on input 1.
IF	#R1:IF C=50 [@GO,@QUIT] @GO:--->S2 @QUIT:--->STOPABORT	If C equals 50 then continue session else save data and end session.
LIST	2":OFF A; LIST A=B(I);ON A--->SX	Every 2" turn off output A, assign a value to A sequentially drawn from array B, then turn on output A.
OFF	2":OFF ^FEEDER ---> S2	Deactivate the output named ^FEEDER.
ON	60" : ON ^FEEDER ---> S2	Activate the output named ^FEEDER.
RANDD	2":OFF A;RANDD A=B;ON A--->SX	Same as LIST example, but A is drawn pseudorandomly without replacement, from array B.
RANDI	2":OFF A;RANDI A=B(I);ON A--->SX	Same as RANDD example, but with replacement.
SET	#R1:SET D=E/F--->SX	Arithmetic assignment of E/F to D.
SHOW	#R1:ADD A;SHOW 2,RESP,A--->SX	Every response increments A and displays A's value in screen channel 1 with the label "RESP".
SUB	1":SUB X--->SX	Every 1" decrement the value of X by 1.
WITHPI	#R1:WITHPI=2500 [@RF,@NORF] @RF:ON ^FEEDER--->S2 @NORF:--->SX	After every response on input 1 pass through a probability gate set to 2500/10000; there is a 25% probability of operating the output named ^FEEDER.
Z	5#R1:ON ^FEEDER;Z^RF--->S2	Send synchronization pulse named ^RF.
~	1":~SOUNDON(100);~ --->SX	Execute user-defined Pascal procedure named SOUNDON.
	TRANSITIONS	
SX,S1-S32	2":OFF ^FEEDER--->S5	Goto state 5. SX indicates no transition.
STOPABORT AND STOPKILL	50#Z^RF--->STOPABORT	After 50 reinforcers (signaled by Z pulse named ^RF) end session, shut off all outputs and retain data. STOPKILL is similar to STOPABORT, but data are discarded.

Figure 1. Brief examples of MSN commands.

plemented in Pascal as real (decimal point) numbers. This extends the flexibility and usefulness of computations performed within MSN programs. The overhead normally associated with the use of real numbers is mitigated by using a math coprocessor. Perhaps the most important feature of MSN is the ability to declare up to 10,000 array elements per program; recording complete interresponse-time data for most experimental paradigms should therefore be feasible.

Steps in Producing Executable Programs

MSN source code is turned into executable (.EXE) programs via a sequence of steps beginning with the creation of a source file (with a .MPC file name extension). Turbo Pascal's integrated environment is particularly well suited to produce such files. Source files are then processed by a program named TRANS. At this point, a run-time file name is declared with a file name extension of .RTM; eventually the Pascal compiler will generate a file with the same name, but with an executable (.EXE) extension. TRANS permits specification of up to eight different state-notation programs that will be translated and compiled into the run-time (executable) program. TRANS then translates the specified MSN programs into new run-time programs.

TRANS logs all detected syntax errors to an error file; translation does not stop when errors are detected. The error file documents the name of the source program, the offending line number, the text of each offending line, and a description of the error. Many compilers and translators halt after detecting a single error, but TRANS's approach substantially accelerates program development.

When all bugs have been removed from the MSN programs and they have been successfully processed by TRANS, the run-time (.RTM) file is compiled using TPC (R), the command-line version of the Turbo Pascal compiler. In addition to compiling and linking the output files of TRANS into an executable file, TPC links these files with the installation data file (named MEDRTM.PAS) created by INSTATE and with optional user-written Pascal procedures contained in a file named USER.PAS. Upon completion of these steps, a program is created which can be executed from the DOS command line.

General Run-Time Features

All run-time programs provide a standard set of features and commands designed to maximize the amount of information displayed, minimize the likelihood of losing data, and provide maximum flexibility to monitor and modify the progress of experimental sessions. Particular emphasis has been placed on error detection, which begins from the moment run-time files are executed from DOS. The first error detection routine determines whether a powered-up interface is present; if an interface is not found, the operator is asked to indicate whether the session should be terminated or should proceed in emulation mode (which permits debugging of programs without an interface). After an interface is connected or emulation mode is selected, a standard display is presented. The display is comparable to the first and last lines of the screen portrayed in Figure 2.

The top line of the screen lists the keyboard commands available. Commands are invoked by typing the first letter of the command name. Most commands generate additional prompts in the menu area between the command line and the first station-status line (which begin with "B"). The bottom line is a status line that shows the current date, time, cumulative timing errors (speed warnings), name of the run-time file, and bytes of available memory.

Experimental procedures are assigned to specific boxes with the "Load" command. The operator then indicates which MSN procedure should be loaded, followed by the subject, experiment, and group identification numbers. Boxes may be loaded without interfering with the timing and processing of events in currently active boxes. After a box is loaded, a status line, similar to the second non-

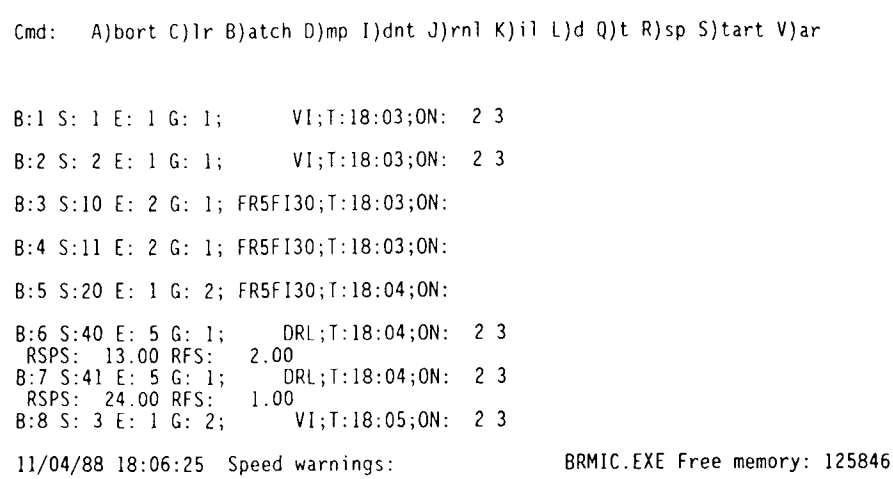

Figure 2. A representative MED-PC run-time screen.

blank line of Figure 2, is placed in the central portion of the screen. It indicates, from left to right, that Box 1 contains Subject 1, a member of Experiment 1, Group 1. The MSN program running in Box 1 is named VI, the box was loaded at 6:03 p.m., and Outputs 2 and 3 are turned on. Immediately beneath the status line for each box is a line reserved for the output of SHOW commands. For example, Box 6 has a SHOW statement labeled RSPS with a value of 13.00.

A variety of other keyboard commands are provided. "Identifier" changes the subject, experiment, and group numbers of a currently executing box. "Abort" and "Kill" are keyboard versions of the MSN STOPABORT and STOPKILL commands described in Figure 1. Inputs may be simulated from the keyboard with the "Response" command. "Start" generates a signal detected by the MSN #START command.

The "Variables" command allows the user to view and alter the contents of arrays and variables. This feature facilitates debugging and setting of experimental parameters at run-time. The "Journal" command provides information on the history of all loads, aborts, kills, and data dumps since the initial loading of the run-time program. In addition, a "Batch" facility can be used to record keystrokes to a disk file for subsequent playback; this feature is especially useful when a large number of parameters are routinely set with the "Variables" command.

Data Recording

Data from sessions terminated at the keyboard or by transition to STOPABORT may be written to disk with the "Dump" command. Writing a disk file is an extremely time-consuming task, so the "Dump" command will only write to disk if no boxes are currently active (an error message is printed if any boxes are active). Data from specific stations may be abandoned, but the operator must verify such requests; this is a safeguard against accidental data loss. Although data recording and analysis practices vary widely among laboratories, MED-PC accommodates this variability by providing two schemes for naming and segregating data, and three formats for structuring their contents. One segregating system places all data from all subjects into a single file, named according to the date of the session. The alternative format places data from individual subjects into separate files that grow larger across sessions; under this scheme, files are named according to subject, experiment, and group identifiers.

Regardless of the growth scheme used, any of three file structures may be produced. The first structure is a fully annotated file in which background information (e.g., date and time of each session) is included along with explanatory labels. The values of all variables are listed in alphabetical order, one variable per line, after which the contents of all array elements are listed in five columns. This format produces a virtually self-documenting printout, but it consumes a large amount of disk space.

The second type of file format is a nonannotated structure that includes virtually all of the information provided in the annotated format. However, in the nonannotated structure, all nonnumerical data are removed, including labels and blank spaces. For example, a session's start time is printed by placing the hours on one line and the minutes on the following line. This format conserves space at the expense of legibility, and assumes that a computer program will be used for data analysis; it would be difficult to visually inspect data written in this format. This format, however, requires less disk space than the fully annotated structure, and is easier to analyze with computer software.

The third and most efficient file structuring option produces a stripped file with a structure similar to the nonannotated file. However, this structure includes only background information and the contents of a single data array which must be named C; all other arrays and simple variables are omitted from the dump. This format should be particularly useful when collecting very large data sets.

Safeguards against accidental data loss include an optional disk identification/verification system. This system assures that data are dumped onto a given disk only if the disk holds a file named "ID"; this file must contain subject, experiment, and group numbers that correspond to those that are being dumped. If "ID" is missing, or contains inappropriate data, then the operator may either override verification or attempt to insert the appropriate disk. Verification prevents data from being fragmented among multiple disks and files when using the filing system in which data from successive sessions are appended to a growing file segregated by subject, experiment, and group number.

Further safeguards against data loss include determining the amount of free space left on data disks prior to every dump. If enough space remains for the present dump but not enough for another dump of the same size, a warning message is generated. If insufficient space remains for even the present dump, the system requires the operator to insert another disk.

TECHNICAL ASPECTS

Timing

Two basic methods may be used to determine the flow of execution of real-time programs. The first method is the interrupt-driven approach, in which high-priority events suspend the processing of lower-priority tasks. After executing the code that caused the interruption, the system returns control to the original task at the point at which it was interrupted. For example, a typical system might allow responses to interrupt screen updates in order to determine if schedule contingencies require feeder operation. In theory, interrupt-driven software has the desirable feature of allowing the user to react to important events as soon as they occur. This approach works

properly, though, only if interrupts do not occur in very rapid succession; if they occur too rapidly, some interruptions must be queued until the preceding interrupts have been serviced. Under these circumstances, the latency to service a given event is indeterminate and potentially very long. Interrupt-driven systems, unless they are very simple ones in which periodic clock ticks generate interrupts, tend to be difficult to write, maintain, and modify. Furthermore, interrupts actually degrade system performance because of the processing overhead they require (e.g., pushing and popping the stack and the processor's flags when entering and exiting the interrupt routines).

An alternative approach to determine the handling of real-time programs is polling, in which very small sections of code are executed in rapid succession. There are a number of variations to this approach, but the one implemented in MED-PC will be described. Our approach defines two sections of code: (1) high-priority sections that control and sense experimental events, and (2) low-priority sections that perform functions such as screen updating and handling of keyboard input.

High-priority sections are serviced on a fixed, periodic time schedule. The frequency with which active high-priority sections are serviced is referred to as the system resolution, and typically has a value of 50 msec on 8088-equipped PCs and 25 msec on 80286 PCs. Conceptually, experimental sessions are divided into 50-msec "time slices" (assuming an 8088 system). At the beginning of each time slice, a new "sweep" is initiated. At the beginning of a sweep, the status of all inputs is recorded and the system clock is read. Active experimental chambers are then sequentially serviced, during which the system updates response counters, adjusts output channels, tracks the progress toward satisfying schedule contingencies, executes user-written procedures, and so forth. After servicing all chambers, the system consults the millisecond timer in the interface to determine the duration of the sweep. Sweeps longer than 50 msec result in a speed warning on the screen and immediate initiation of the next sweep. However, the overwhelming majority of sweeps require substantially less than 50 msec. Low-priority events are serviced if at least 40 msec remain before the scheduled starting time of the next sweep, or if low-priority events have not been serviced for at least 500 msec. The latter provision is necessary to prevent keyboard lockups. After low-priority events are serviced, the system continuously monitors the timer until it is time to initiate the next sweep.

Programmers sometimes avoid polling techniques because of concerns about the possibility of missing responses of short duration. This concern is well founded if the polling software is relatively slow and the interval between responses is less than the system's resolution. MED Associates' input cards drastically reduce the likelihood that responses will be missed due to short duration, however, because onboard memory locations latch indefinitely until they are sampled. The other concern, that of interrespone times being shorter than system resolution, is alleviated by having the system poll inputs at a rate that exceeds the maximal local response rates typically encountered in conditioning paradigms. Experimenters who require extraordinarily rapid polling may match their choice of PC to their requirements simply by acquiring an especially fast computer system.

```
1   \FR5FI30
2   S.S.1,
3   S1,
4      5#R1:On 1--->S2
5   S2,
6      .1":Off 1--->S1
7   S.S.2,
8   S1,
9      30"--->S2
10  S2,
11     1#R2:On 1--->S3
12  S3,
13     .1":Off 1--->S1
```

Figure 3. A simple MSN program for a concurrent fixed-ratio 5 (left manipulandum), fixed-interval 30-sec (right manipulandum) schedule. Line numbers are not part of the syntax of MSN, but have been added to the left margin for clarity.

Translation of MEDSTATE NOTATION to Pascal

Programs written in MSN consist of independently functioning code segments known as "state sets." Each state set contains several states, each of which has one or more statements. Figure 3 shows the listing for a concurrent fixed-ratio 5 (FR 5), fixed-interval 30-sec (FI 30) schedule. Lines 2–6 constitute State Set 1 and dictate that program execution begins with line 4. After five responses on Manipulandum 1, Output 1 is activated and control is transferred to State 2. After 0.1 sec, Output 1 is deactivated and control is returned to State 1. The code of State Set 2 (lines 7–13) executes in parallel to State Set 1, but is totally independent of events in State Set 1. State Set 2 should be recognizable as an FI 30 schedule on Manipulandum 2.

Although state notation's parallelism is alien to most programmers who use conventional languages, there is a surprisingly direct correlation between state sets and certain features of structured languages. The key to translating MSN to Pascal resides in exploiting the structural similarities between state sets and Pascal case statements. Case statements are conditional control structures similar to, but more powerful than, "if" statements in BASIC or FORTRAN. For example, the following Pascal fragment would write "One" on the screen:

```
A := 1;
Case A of
  1:Write('One');
  2:Write('Two');
  3:Write('Three');
end;
```

Changing "A := 1" to "A := 3" would place "Three"

```
Procedure FR5FI30(Box:integer);
Begin
 FR5FI30Record[Box] Do
  {StateSet 1 Follows}
  Case StateSet[Box][1] of
   1:begin{State 1}
      If Response[Box][1] Then
       ResponseCount[Box][1][1] = ResponseCount[Box][1][1] + 1;
       If ResponseCount[Box][1][1] > 5 Then
       begin
        TurnOn(Box,1);
        ResponseCount[Box][1][1] := 0;
        Timer[Box][1] := CurrentTime + TimeValue[Box][1][2];
        StateSet[Box][1] := 2;
       end;
     end;{State 1}
   2:begin{State 2}
      If CurrentTime >= Timer[Box][1] Then
      begin
       TurnOff(Box,1);
       Timer[Box][1] := CurrentTime + TimeValue[Box][1][1];
       StateSet[Box][1] := 1;
      end;
     end;{State 2}
  end;{StateSet 1}

  {StateSet 2 Follows}
  Case StateSet[Box][2] of
   1:begin{State 1}
      If CurrentTime >= Timer[Box][2] Then
      begin
       Timer[Box][2] := CurrentTime + TimeValue[Box][2][2];
       StateSet[Box][2] := 2;
      end;
     end;{State 1}
   2:begin{State 2}
      If Response[Box][2] Then
       ResponseCount[Box][2][2] = ResponseCount[Box][2][2] + 1;
       If ResponseCount[Box][2][2] > 1 Then
       begin
        TurnOn(Box,1);
        ResponseCount[Box][2][2] := 0;
        Timer[Box][2] := CurrentTime + TimeValue[Box][2][3];
        StateSet[Box][1] := 3;
       end;
     end;{State 2}
   3:begin{State 3}
      If CurrentTime >= Timer[Box][2] Then
      begin
       TurnOff(Box,1);
       Timer[Box][2] := CurrentTime + TimeValue[Box][2][1];
       StateSet[Box][3] := 1;
      end;
     end;{State 3}
  end;{StateSet 2}
 end;{With}
End;{Procedure}
                                        {MAIN LOOP}
Begin
 Repeat
  DoLowPriorityTasks;
  GetTime:
  For Box := 1 to 8 Do
  begin
   Case ProcedureToRun[Box] of
    1:FR5FI30(Box);
    2:Some_Other_Schedule(Box);
   end;
  end;
 Until QuitSignal = True;
End.
```

Figure 4. A Pascal code fragment similar to the one which TRANS would generate for the MSN code listed in Figure 3. Note the correspondence between state sets and case statements.

on the screen. The following code is an equivalent BASIC fragment:

```
10 A = 1
20 If A = 1 Then Print "One"
30 If A = 2 Then Print "Two"
40 If A = 3 Then Print "Three"
```

A case statement differs from a series of "if" statements in that a maximum of 1 alternative may be executed on each pass through a case statement; in the preceding case statement, changing the third line from "1:Write('One');" to "A := 2" would not place "Two" on the screen, unless the case statement was executed a second time without resetting A to 1.

Translation of MSN to Pascal is achieved by treating each state set as a case statement and each state as a case statement alternative. Figure 4 shows simplified Pascal code that corresponds to the concurrent FR 5, FI 30 MSN code in Figure 3. The Pascal code is conceptually similar, but not identical, to the one that TRANS generates. A two-dimensional array named StateSet controls the execution of each state set. For example, StateSet[Box][1] contains the current state of State Set 1 for each box. Transitions between states are accomplished by altering the value of this variable. For example, when program execution begins, State Set 1 is in State 1 (by default). After a response on Manipulandum 1, Output 1 is turned on and transition to State 2 is effected by setting the value of StateSet[Box][1] equal to 2.

Each array in Figure 4 is subscripted by a variable name "Box," which enables multiple boxes to share procedures. For example, the main loop of the example (at the bottom of Figure 4) specifies that Boxes 1 through 8 will be sequentially serviced. Assuming that all boxes are running the FR 5, FI 30 program, then FR 5, FI 30 would execute only those statements appropriate to the current status of Box 1 by accessing the values stored in Element 1 of its arrays (StateSet, Response, ResponseCount, etc.).

This system of storing data in arrays and records for each station facilitates the sharing of single procedures among multiple stations; code does not need to be redundantly generated for each box. Furthermore, this is the key to assigning procedures to stations at run time. In the main loop in Figure 4 is a case statement controlled by an array name "ProcedureToRun." This variable stores the procedure number assigned to a given box when it was loaded by the experimenter. Thus, Box 1 would execute FR 5, FI 30 if the experimenter set ProcedureToRun[1] equal to 1 via keyboard input; alternatively, Some_Other_Schedule would execute if ProcedureToRun[1] had been set to 2. In reality, the code produced by TRANS is conceptually similar, albeit more complex than the code supplied here. It is hoped that this explanation will serve as a useful starting point for other programmers.

CONCLUSION

Future Developments

We are currently in the process of testing and developing an enhanced version of MED-PC. Enhancements are expected to include an expanded system for displaying data via SHOW commands. Presently, up to six data values per box may be displayed, and all boxes may display their data simultaneously. The new system will permit each box to display up to 40 values in a window at the bottom of the screen. Data for a given box will be displayed for 5 sec, followed by data for the next box, and so on. The order and duration of data presentation will be user-controllable.

We are also considering new language constructs, including: (1) interbox Z pulses to facilitate synchronization of events across boxes, (2) mechanisms for sharing variables among boxes, (3) standard commands for placing text on the screen, (4) an input command named #F which will treat function keys as inputs (e.g., #F10→S2), (5) parenthetical mathematical expressions and computed array subscripts, and (6) faster program execution.

Availability

MED-PC and associated hardware is available from MED Associates, Incorporated, Box 47, East Fairfield, VT 05448, 802-872-3825.

REFERENCES

BALSAM, P. D., DEICH, J., O'CONNOR, K., & SCOPATZ, R. (1985). Microcomputers and conditioning research. *Behavior Research Methods, Instruments, & Computers*, **17**, 537-545.

BUTLER, F. E. (1980). MicroSKED. *Behavior Research Methods & Instrumentation*, **12**, 152-154.

BUTLER, F. E., & GRISHAM, M. G. (1977). SKED-controlled experimentation in an undergraduate instructional laboratory. *Behavior Research Methods & Instrumentation*, **9**, 219-221.

CHAYER-FARRELL, L., & FREEDMAN, N. L. (1987). CORE: Computer-controlled operant reinforcement. *Behavior Research Methods, Instruments, & Computers*, **19**, 319-326.

ELSNER, J. (1982). PASTOR: A new schedule programming language. *Behavior Research Methods & Instrumentation*, **14**, 254-263.

GILBERT, S. G., & RICE, D. C. (1978). NOVA SKED: A behavioral-notation system for Data General minicomputers. *Behavior Research Methods & Instrumentation*, **10**, 705-709.

GILBERT, S. G., & RICE, D. C. (1979). NOVA SKED II: A behavioral notation language utilizing the Data General Corporation real-time disk operating system. *Behavior Research Methods & Instrumentation*, **11**, 71-73.

KAPLAN, H. L. (1985a). Design decisions in a Pascal-based operant conditioning system. *Behavior Research Methods, Instruments, & Computers*, **17**, 307-318.

KAPLAN, H. L. (1985b). When do professional psychologists need professional programmers' tools? *Behavior Research Methods, Instruments, & Computers*, **17**, 546-550.

LESLIE, J. C. (1981). State notation programming languages in psychology. *International Journal of Man-Machine Studies*, **14**, 341-354.

LUCAS, G. A. (1986). *The conman control manual* [Computer program manual]. Bloomington, IN: Spyder Systems.

MECHNER, F. (1959). A notation system for the description of behavioral procedures. *Journal of Experimental Analysis of Behavior*, **2**, 133-150.

MILLENSON, J. R. (1971). A programming language for on-line control of psychological experiments. *Behavioral Science*, **16**, 248-256.

MOORE, E. F. (1956). Gedanken-experiment on sequential machines. In *Automation studies*. Princeton, NJ: Princeton University Press.

ORR, J. (1984). Going FORTH in the laboratory. *Behavior Research Methods, Instruments, & Computers*, **16**, 193-198.

PALYA, W. L. (1988). An introduction to the Walter/Palya controller and ECBASIC. *Behavior Research Methods, Instruments, & Computers*, **20**, 81-87.

SNAPPER, A. G. (1973). Use of a notation system for digital control and recording. *Behavior Research Methods & Instrumentation*, **5**, 124-129.

SNAPPER, A. G., KADDEN, R. M., & INGLIS, G. B. (1982). State notation of behavioral procedures. *Behavior Research Methods & Instrumentation*, **14**, 329-342.

SNAPPER, A. G., KNAPP, J. Z., KUSHNER, H., & KADDEN, R. M. (1967, June). *A notation system for behavior experiments*. Paper presented at the meeting of the Digital Equipment Users Society, New York, NY.

STADLER, S. J. (1969). On the varieties of computer experience. *Behavior Research Methods & Instrumentation*, **1**, 267-269.

TAKIGAWA, T., & MINO, T. (1981). TYMES: A high-level language for process control and data manipulation in the behavior laboratory. *Behavior Research Methods & Instrumentation*, **13**, 741-746.

WALTER, D. E., & PALYA, W. L. (1984). An inexpensive experiment controller for stand-alone applications or distributed processing networks. *Behavior Research Methods, Instruments, & Computers*, **16**, 125-134.