MED-PC®

SOF-735

PROGRAMMER'S MANUAL

DOC-003
Rev. 3.4

Copyright ©2012
All Rights Reserved

notes

Table of Contents

CHAPTER 1 | INTRODUCTION

This manual has been designed to aid all users of the state notation language, MedState Notation (MSN), used with MED-PC®.

The novel user will find that each of the following chapters introduces commands used in MedState Notation and then presents example programs using those commands. The chapters build on one another, so it is recommended that the manual be read from cover to cover, taking the time to try each Tutorial. There should be no concerns about having to set aside large blocks of time to read the manual; the manual has been written such that each chapter is quite brief. To get the most from each chapter it is recommended that the reader type in each program in the Tutorial to test their new knowledge. Also, be sure to save the files with the names suggested, as each Tutorial builds on the previous one. This will make it easier to transition from one Tutorial to the next, and creates the habit of going back to old code for ideas and/or shortcuts in programming.

The intermediate user may find the chapters of some use, but if already familiar with most of the commands, it is possible to use to the Tutorials at the end of each chapter to brush up on MedState Notation.

The experienced user may want to look at Appendix A for any refreshers needed for codes.

There is also a chapter on how to write and use Macros.

Software Backup

It is highly advised that a backup of all programs and/or data be created on a regular basis.

Before Getting Started

In order to get the most out of the Tutorials at the end of each chapter, it is recommended that MED-PC be installed on the hard drive and configured for the existing hardware prior to reading this manual. If necessary, consult the MED-PC User's Manual for step-by-step directions.

CHAPTER 2 | BASIC CONCEPTS OF PROGRAMMING IN MED-PC®

State Sets

MedState Notation procedures are organized into blocks of code called State Sets. There may be as many as 32 State Sets within a single procedure, with each State Set functioning autonomously and with apparent simultaneity. Within MedState, it is represented by the **S.S.#,** where # is a number between one (1) and thirty-two (32).

State Sets do not need to be numbered consecutively or be in ascending order, as they are processed in the order in which they appear in a procedure. Each State Set MUST have a unique number and may not contain decimal points, constants or variables. The periods following each "S" and the comma following the number are required.

States

The basic unit of a State Set is the State. At any given moment, a procedure can be thought of as being in a given State. When a procedure begins to execute, it is always in the first (i.e. topmost) State. States are indicated by **S#,** where # is an integer between one (1) and thirty-two (32) with the same restrictions on numbering as indicated for State Set numbering.

Statements - General Description

Statements are within States and are made up of commands. Comparing the basic elements of a program (e.g., State Set, State, and Statements) to the components of a book, the State Sets are the chapters, the States are the paragraphs within those chapters, the Statements are the sentences, and the commands are the words.

The Components of a Statement

A statement is composed of three components: an **Input Section**, an **Output Section**, and a **Transition**. The input section consists of the commands to the left of the colon ":", the output section is between the colon and the arrow "--->", and the transition is to the right of the arrow.

A statement may be thought of as an IF - THEN - GOTO statement. For example, the following statement means, IF "Input" occurs THEN "Output" and GOTO "Next:"

```
Input: Output ---> Next
```

Actual code may look like this:

```
#R1: ADD A; ON 5 ---> S3
```

Typing Conventions

Case Is Irrelevant

Upper and lower case letters may be freely intermixed and used in any manner that best clarifies source code for a procedure.

Spacing And Blank Lines

Spacing and blank lines are ignored. This feature may be used to make source code statements as clear and as easy to read as possible.

Rules for Comments

Comments are notes placed into the code that are not translated into PASCAL. Comments may be placed on their own lines or following the end of any line of MedState Notation code. Comments always begin with a '\' backslash. Please note that this is the same character as the DOS path separator, not the arithmetic division symbol, (/). Comments may not occur in the middle of a statement.

Legal examples:

```
\ This is a comment before State Set 1 (S.S.1)
S.S.1,
\ This is a comment before State 1
S1,
   Input: Output ---> Next
```

Illegal examples:

```
Input  \ This is the House Light : Output ---> Next
```

Integers

Integers are whole or counting numbers, such as 0, 5, 112 and 3000, which do not contain decimal points. A few commands logically require the use of integers, and MED-PC automatically converts numbers to integer format where necessary, by rounding them to the nearest whole number. For example, "ON 1.9" or "ON 2.1" are illogical, but will not cause difficulties because MED-PC will automatically convert both to "ON 2." The only place where proper use of integers is required is in declaring constants and in numbering State Sets and States.

Constants

Constants are a convenient means of substituting words for frequently used integers. Constants must be declared prior to the first State Set and may be up to 55 characters in length. They must be preceded by a caret "^" and can be comprised of any combination of letters and numbers. Spaces are ignored. An example declaring "^Feeder" follows

```
^Feeder = 1

S.S.1,
S1,
  Input: ON ^Feeder ---> S2

S2,
  2": OFF ^Feeder ---> S1
```

Constants must be declared as having an integer value, and time may not be assigned a constant. For example, **^Feeder = 1.1** and **^FeederDur = 2** are illegal. As constants are represented as integer (whole) numbers, it is illegal to attempt to assign the value of a variable to a constant during program execution. Up to 1000 constants may be declared in a single procedure and constants are restricted to holding values within the range of –2,147,483,647 to 9,223,372,036,854,775,807.

The use of constants cannot be emphasized enough. Constants tremendously improve readability of a procedure and can substantially reduce debugging time. It is good practice to define and use constants to refer to all inputs and outputs.

Tutorial 1: Writing the first program

The basic terminology and concepts down to write a simple program will replace the vague notions of "Input," Output," and "Next" from the previous sections and replace them with concrete example of code. This exercise will cover how to arrange State Sets, States and Statements as well as the proper use of Statements, Constants, Inputs, and Outputs. For this example, assume that the operant chamber is equipped with a House Light, although any output device can be used.

The first step is to open TRANS IV, then click **File| New**[1].

Figure 2.1 - Open Trans IV

---
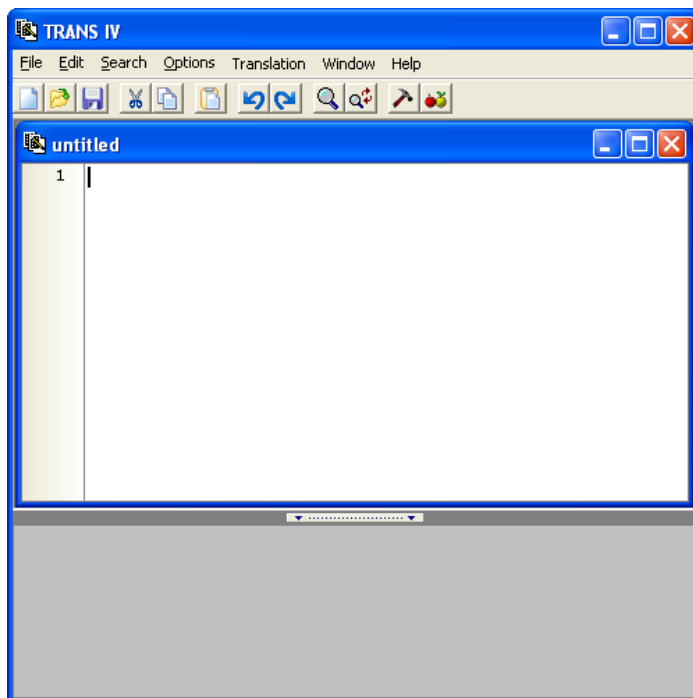
[1]   Note that any ASCII text editor may be used to type the initial code. If a text editor other than the one supplied with TRANS is used, however, you must save the text as unformatted ASCII or DOS text and it must be saved with the extension *.MPC (where * is your filename.).

Next enter the comments. It does not matter what is contained in the comment section, it is only there for convenience.

Figure 2.2 - Enter Comments
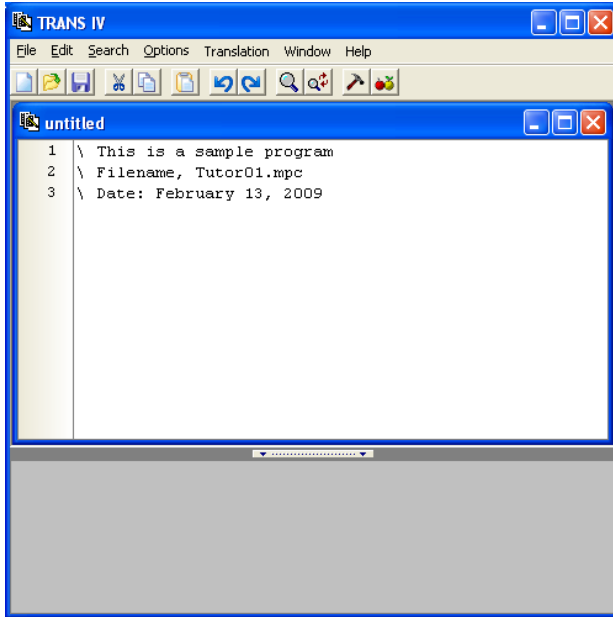
Next, define the constants. This is a simple program with only one input, time, and one output, the House Light. Therefore, define one constant for the House Light and call it "House."

Figure 2.3 - Define Constants
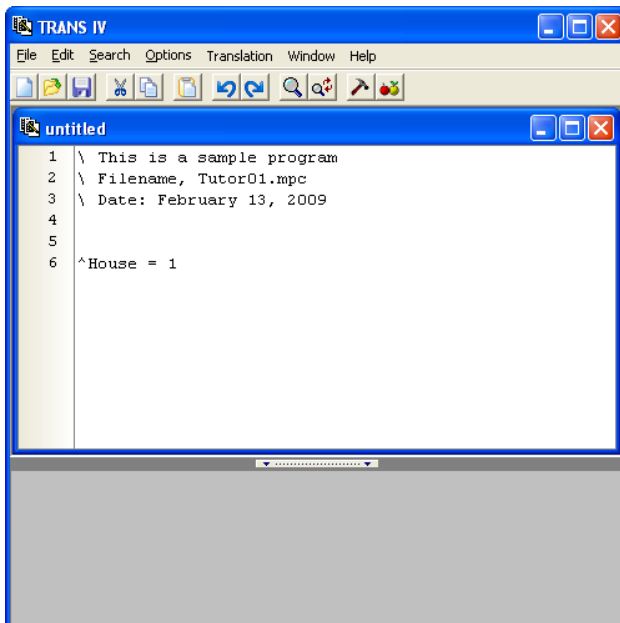
As previously mentioned, time will be the input. In MedState Notation, minutes are represented by a single quotation mark (**'**) and seconds are represented by a double quotation mark (**"**). In this example, the first input will be one second (1") and the second input will be five seconds (5"):

Figure 2.4 - Enter Time
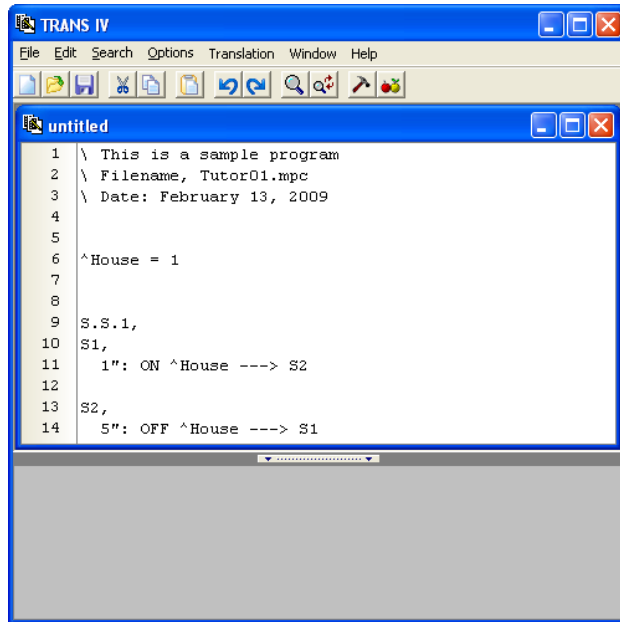
Save this file as Tutor01.mpc in the default directory[2]. Then click **Translation | Translate and Compile**." Highlight the filename **Tutor01.mpc** and click **MAKE**. There should now be a letter **M** listed a tab space before file name as shown in Figure 2.5.

Figure 2.5 - Select Files to Translate

---
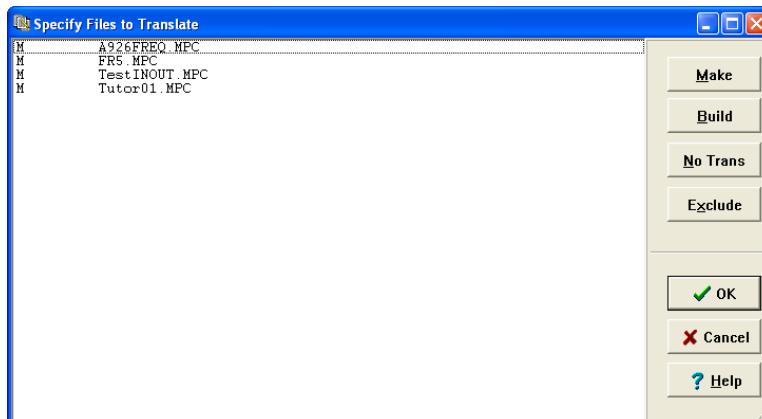
[2] Note: If another text editor is being used, save the file and close the text editor, then open TRANS and follow the above directions.

Once MED-PC is opened, select **File | Open Session** and select Tutor01.mpc from the Procedure pulldown menu. Click **OK** to open.

Figure 2.6 - Open Session

After one second, the house light should come on for five seconds, turn itself off for one second, and repeat this cycle until the session is closed.

To close the session, click **File | Close Sessions**. The screen shown below will appear.

Figure 2.7 - Close Experimental Sessions Menu

Select the checkbox next to **Box 1** and the **Stop, Abandon Data (StopKill)** radio button. Note that the light turning on and off continues when in this window. This is because the STOPKILL command is not sent until the OK button is clicked.

This concludes the first tutorial.

CHAPTER 3 | #R, SX, ADD, AND SHOW COMMANDS

In the previous chapter, time was the only input. Although time is used quite often an input, what may be of more interest to researchers is the input from a test animal on a response lever, lickometer, nose poke, etc. Also, remember there was not much being displayed on the MED-PC window as the first program was being run. This chapter will explain how to write code for the recording of mechanical inputs, as well as how to display the information on the screen.

#R

#R is, in the simplest terms, the code for the response of a test animal on a MED Associates input device (e.g. a lever). In order for #R to have any meaning in a program, the device that the response is on must be specified, as well as the number of times the response must happen. So the syntax is:

```
[Number of times]#R[Device that is collecting input]: OUTPUT ---> NEXT
```

So real code may look like:

```
5#R^LeftLever: ---> S4
```

Which means, "After five presses of the Left Lever, make the transition to State 4."

Or:

```
#R^RightLever: ON ^Pellet ---> S2
```

Which means "After one press of the Right Lever (MedState Notation has a default of one for #R), turn on the pellet dispenser and make a transition back to S2."

Null Transition (SX)

Sometimes it is desirable to have a transition that does not reset the input conditions for the entire state. MedState Notation can do this with a command called SX, which is also known as the Null Transition. In code it would come after the transition arrow:

Syntax: `INPUT: OUTPUT ---> SX`

The following two examples will help explain the power of the Null Transition.

Example 1:

```
S2,  \ 1 min ITI.  Count Responses during ITI.
  1': ---> S3
  #R1: ADD C ---> SX  \ A Response on Input 1 does not
                      \ reset the 1 minute timer
```

In Example 1 the program times 1 minute and counts all responses that happen on Input 1. Responses on Input 1 do not affect the 1 minute timer because of the transition to SX.

Example 2:

```
S2,  \ 1 min ITI.  Restart the ITI timer if animal presses the lever.
```

```
    1': ---> S3
    #R1: ADD C ---> S2  \ A Response on Input 1 does
                        \ reset the 1 minute timer
```

In Example 2 the program times 1 minute and counts all responses that happen on Input 1. But in this example a Response on Input 1 also resets the 1-minute timer because of the transition to S2. The animal is being punished for responding during the ITI.

ADD

As the name suggests, ADD is a mathematical command that will increment a variable by one. It is an output command, so it will always follow the colon in the code.

Syntax: `INPUT: ADD X ---> NEXT`

Where: X = the variable to which the value 1 will be added.

Real code may look like this:

```
    S1,
      1": ADD C ---> S2
```

In this example, after one second, the value one will be added to the variable C and then the transition will be made to S2.

SHOW

The SHOW command indicates to the user that a MED-PC program is running. The bottom portion of the screen may be used to display data for each active Box in any of the 200 available positions (numbered 1 - 200).

Syntax: `INPUT: SHOW P,Label,X ---> NEXT`

```
  Where:                     P =    Position 1-200 (must be defined).
                             Label = A user-defined name for that position.
                             X =    The variable value listed in position 1-200.
```

Real code may look like this:

```
    #R2: ADD A; SHOW 1,Center Key,A ---> SX
```

Where after one response on input 2, one will be added to the variable A and finally the value of variable A will be displayed on the screen in position 1 (and to the left of the value will be the label Center Key) before making the null transition.

Tutorial 2: Expanding the first program

In this exercise the program written in the first Tutorial will be expanded upon. The goal of this program is for a count to appear on the screen each time the left lever is pressed.

Open the text editor (TRANS) and type in the following:

```
\ This is a sample program
\ Filename, Tutor02.mpc
\ Date: February 13, 2009


^House     = 1
^LeftLever = 1

S.S.1,
S1,
  0.01": ON ^House ---> S2
```

Note that the House Light is an output whereas the Left Lever is an input; therefore both constants can be defined as 1. Refer back to Chapter 2 if there are any questions about the program thus far.

Next, add the code for the responses, the count and the display by adding a new State within State Set 1.

```
\ This is a sample program
\ Filename, Tutor02.mpc
\ Date: February 13, 2009


^House     = 1
^LeftLever = 1


S.S.1,
S1,
  0.01": ON ^House ---> S2

S2,
  #R^LeftLever: ADD C; SHOW 1,Count,C ---> SX
```

Save the program as Tutor02.mpc in the default directory, translate and compile the program and then open it in MED-PC. Refer back to Tutorial 1 for instructions on how to accomplish these tasks.

If the program is running, the House Light will turn on immediately after loading the program. Press the Left Lever and notice that the count increments on the screen.

Do not be alarmed if the count is not going up geometrically (…11, 12, 13, …), but sporadically (11, 16, 18). The screen update is a low priority function. If the responses are rapid, the computer is keeping an accurate count of the data, but the screen is only updated when the system gets a chance.

This concludes the second tutorial. The session may be ended in the same manner as Tutorial 1.

CHAPTER 4 | CONTROLLING THE BEGINNING AND END OF A PROGRAM

Thus far, all programs that have been written have begun running the second they were loaded. However, the experiment may require more than one Box should be loaded and run simultaneously, or perhaps the program should be ready to go before loading a test subject in the chamber. In either case, starting the program upon loading is not desirable. This chapter will deal with this problem, as well as how to stop the program without issuing the stop session command.

#START

The #START command give the user the ability to load a procedure but hold procedure initiation until a signal is given by the experimenter. This is useful when loading several Boxes because this enables the experimenter to place multiple subjects in experimental chambers and then start their sessions simultaneously.

Syntax: `#START: OUTPUT ---> NEXT`

Real code may look like:

```
#START: ON ^House ---> S2
```

This means, "Wait until a START command has been issued and when it has, turn on the House Light and make the transition to State 2."

The START command is explained further in Tutorial 3.

STOPABORT and STOPKILL

These two commands cause the program that is running to immediately stop executing. Any outputs currently turned on are turned off immediately (i.e., whether the program is in the middle of a procedure or not, everything stops). In addition, the Box's status lines on the monitor are cleared. The difference between STOPABORT and STOPKILL is that data collected up to the STOPABORT command can still be salvaged by saving and/or printing it (i.e., it is still in memory) whereas STOPKILL wipes the data from the memory. These commands are special transitions.

Syntax: `INPUT: OUTPUT ---> STOPABORT`

`                INPUT: OUTPUT ---> STOPKILL`

Real code may look like this:

```
2': ---> STOPABORT
```

Or

```
2': ---> STOPKILL
```

This is saying after 2 minutes, make the transition to stop but leave the data in memory (or stop and wipe the memory clean).

STOPABORTFLUSH

Like STOPABORT, STOPABORTFLUSH is a special transition that turns off all outputs and stops procedure execution. Instead of keeping the data in memory, however, STOPABORTFLUSH will save the data to the hard drive and then wipe the memory clean.

Syntax: `INPUT: OUTPUT ---> STOPABORTFLUSH`

Real code may look like this:

```
60': ---> STOPABORTFLUSH
```

Where after 60 minutes, the program will stop, the Box status will clear, the data will be saved to disk and the memory will be wiped clean.

Multiple Commands

MedState Notation allows for the stringing together of multiple commands. In order to do this, semicolons are used to separate the parameters of a command from the following command.

Syntax: `INPUT: OUTPUT#1; OUTPUT#2 ---> NEXT`

Real code may look like this:

```
#R^LeftLever: ADD C; SHOW 1,Count,C ---> SX
```

Tutorial 3: Expanding the last program to control itself

In this exercise the program written in the second Tutorial will be expanded upon. The goal of this program is to demonstrate how to issue a start command, have the program stop on its own and what to do once the program stops.

Open the text editor (TRANS) and type in the following:

```
\ This is a sample program
\ Filename, Tutor03.mpc
\ Date: February 13, 2009


^House     = 1
^LeftLever = 1


S.S.1,
S1,
```

Next, add the code for the responses, the count and the display by adding a new State within State Set 1.

```
\ This is a sample program
\ Filename, Tutor03.mpc
\ Date: February 13, 2009


^House     = 1
^LeftLever = 1


S.S.1,
S1,
  <This we will fill in>

S2,
  #R^LeftLever: ADD C; SHOW 1,Count,C ---> SX
```

Add the start command under State Set 1, State 1:

```
#START: ON ^House ---> S2
```

Add a new State Set so the program will stop on its own by adding the following at the bottom of the new program:

```
S.S.2,
S1,
  #START: ---> S2

S2,
  1': ---> STOPABORT
```

The final product should look like this:

```
\ This is a sample program
\ Filename, Tutor03.mpc
\ Date: February 13, 2009


^House     = 1
^LeftLever = 1


S.S.1,
S1,
   #START: ON ^House ---> S2

S2,
   #R^LeftLever: ADD C; SHOW 1,Count,C ---> SX


S.S.2,
S1,
   #START: ---> S2

S2,
   1': ---> STOPABORT
```

The program is now ready to be saved as Tutor03.mpc in the default directory.

Translate and compile the program and then open it in MED-PC. Notice that, unlike before, the house light is not on. But look at the upper left hand corner of the window -- it shows that the program "Tutor03" was loaded at xx:xx (the computer clocks current time). This shows that the Box was indeed loaded, and is now awaiting a start command.

Figure 4.1 –Box Loaded and Waiting for Start Command
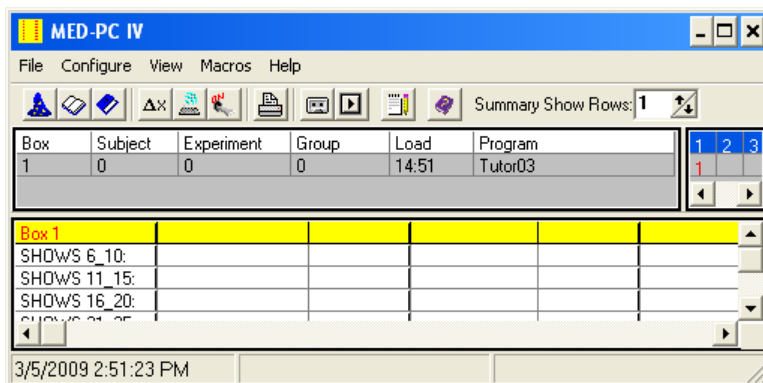
Issue a START command by selecting **Configure | Signals**. The screen shown below will appear.

Figure 4.2 - Send Signals to Boxes
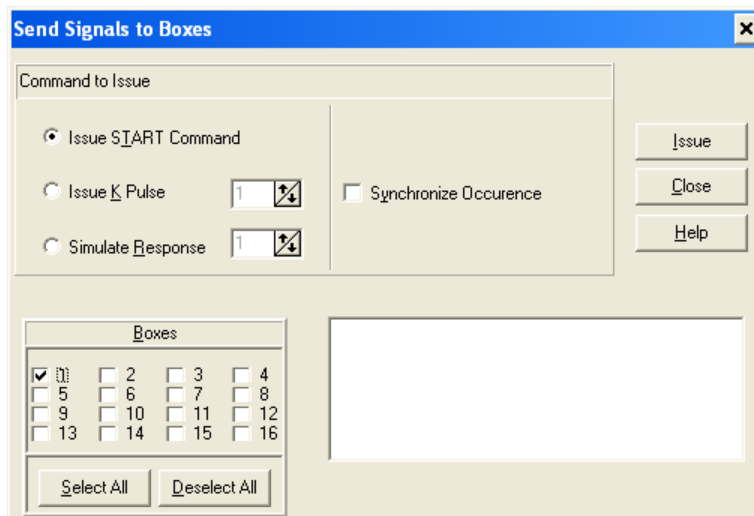
Select the **Issue START command** radio button and the checkbox next to **BOX 1**, as shown in Figure 4.2, then click the **Issue** button. The program is now running.

Depress the Left Lever a few times to get a few counts on the screen. After one minute, the House Light will go off, the counter will stop incrementing, and the information next to Box 1 will be gone and the word, "Closed" will appear, demonstrating that the program was successfully STOPABORTed.

Keep in mind, the data is still in memory and if MED-PC is closed the following message will appear: **Boxes still running, data not printed, macro recorder is on or data not dumped. Close MED-PC?** Select **No**, then select **File | Print** and the screen shown below will appear. Check **BOX 1** and click **OK**.

Figure 4.3 – Boxes to Print
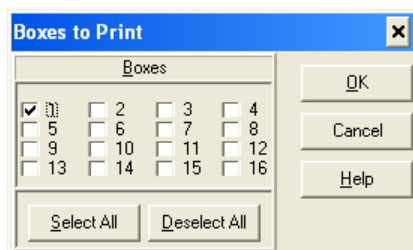
The data from the "Experiment" will be printed. Save this data or close without saving.

This concludes the third tutorial.

CHAPTER 5 | CREATING AN FR SCHEDULE PROTOCOL

Z-Pulses (#Z)

MedState Notation utilizes Z-Pulses to communicate between State Sets. Procedures composed of multiple State Sets are more readable and easier to modify than single State Set procedures and Z-Pulses provide a convenient means for communicating among State Sets.

For example, in order to flash on the house light whenever the FR-5 contingency has not been satisfied and turn it off during feeder operation, there are several options. A program may be written that had all of this in one State Set, but it would have the potential to be prone to programming errors. It would be easier to program the flasher as a separate State Set and then turn it off and on as needed (i.e., when the test animal has or has not met the conditions defined).

Like State Sets and States, Z-Pulses must be numbered, but the highest Z-pulse may not be greater than 32[3]. Unlike any of the other commands which fit nicely in the INPUT: OUTPUT ---> NEXT format as either an INPUT, an OUTPUT or a NEXT, Z-Pulses are unique in that each Z-pulse acts as either an input or an output.

Syntax: `INPUT: ZN ---> NEXT`

`#ZN: OUTPUT ---> NEXT`

Where: N =   An integer between 1 and 32 and is the same in both examples. Also note that when used as an output, the syntax is **ZN** but when used as an input, the syntax is **#ZN**.

In real code, it may look more like this:

```
S.S.1,
S1,
  #START: ON ^HouseLight; Z1 ---> S2


S.S.2,
S1,
  #Z1 ---> S2
```

Rules For Comments Revisited

Comments have been included at the beginning of each program thus far, however comments may also be placed at the END of a line code (comments may not occur in the middle of a statement). This tactic will be used in future Tutorials to explain the code further (please note that the comments in the Tutorials are optional).

Syntax: `Input: Output ---> Next  \ Comment`

---

[3]   The numbering of Z-Pulses does not have to be sequential – they are processed in the order they are read. However it is recommended the numbers be sequential in order to minimize potential confusion.

Tutorial 4: Writing an FR-5 Program

In this exercise, parts of the program written for the last Tutorial will be used. The goal of this Tutorial will be to write a program that works on a Fixed Ratio Schedule.

Open the text editor (TRANS) and type in the following:

```
\ This is an FR-5
\ Filename, Tutor04.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


S.S.1,  \ Main Control Logic for "FR"
S1,
  #START: ON ^House ---> S2
```

Add the code for the responses (remember, this is an FR-5), and a means of counting the responses in another State Set. The Z-pulse (Z1) records the rewards in another State Set, and this will be programmed later. In order to do this, add a new State within State Set 1.

```
S2,
  5#R^LeftLever: ON ^Reward; Z1 ---> SX
```

Program the response count and set it up to display on the screen:

```
S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,     \ This will not put label the "Responses"
        \ and its value "A" on screen until the
        \ START command is issued.
  #START: SHOW 1,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 1,Responses,A ---> SX
```

Now it is time to insert the code for the reward counter and timer using the Z-pulse generated in State Set 1. Note that the Z-pulse has a # sign in front of it. This demonstrates that it is an input, as opposed to an output (as it was in State Set 1):

```
S.S.3,  \ Reward Counter and Timer
S1,
  #Z1: ADD B; SHOW 2,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1
```

The final bit of code is for the session timer. It is identical to the session timer used in the previous Tutorial.

```
S.S.4,  \ Session Timer
S1,
  #START: ---> S2

S2,
  1': ---> STOPABORT
```

Since that was the last State Set, the final product looks like this:

```
\ This is an FR-5
\ Filename, Tutor04.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


S.S.1,  \ Main Control Logic for "FR"
S1,
  #START: ON ^House ---> S2

S2,
  5#R^LeftLever: ON ^Reward; Z1 ---> SX


S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 1,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 1,Responses,A ---> SX


S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 2,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1
```

```
S.S.4,  \ Session Timer
S1,
  #START: ---> S2

S2,
  1': ---> STOPABORT
```

The program is now ready to be saved as Tutor04.mpc in the default directory.

Translate and compile this program and then open it in MED-PC. The upper left hand corner of the window should show that the program "Tutor04" was loaded at xx:xx (the computer clocks current time).

Now issue the START command, then press the Left Lever repeatedly to get responses and reward counts on the screen. When one minute has passed and the program has shut down, the data is hanging in limbo. Save, Print or Delete the data.

This concludes the fourth tutorial.

CHAPTER 6 | ESTABLISHING DEFAULT VALUES FOR VARIABLES

SET

SET is used to perform any of four basic mathematical operations involving two or more operands[4]. The four operators permitted are multiplication (*), division (/), addition (+), and subtractions (-). Although always outputs, two forms of this command are possible as indicated by syntax A and syntax B.

Syntax A: INPUT: SET P1 = P2 Operator P3 ---> NEXT

Syntax B: INPUT: SET P1 = P2 ---> NEXT

Where: P1 =   Variable or array element.

P2 =   Number, variable, or array element.

P3 =   Number, variable, or array element.

Operator = A mathematical operation (e.g., *, /, +, or -).

It is important to point out that the stringing of elements with in the program is permissible with each operation separated by a comma. Variables may also be set to seconds or minutes (i.e., P2 or P3 may be followed by " or ' to assign a time value to a variable). Assigning a new value to a constant, however, is not permissible.

Real code may look like this:


```
1': SET A = 5 * A, B = C(K)  ---> SX
#R3: SET A = 5 * (A + B) + C ---> SX
#START: SET A = A * 1"
```


In the past, complicated math expressions had to be broken into pieces. MedState Notation has now been extended so that complex expressions (e.g., 1 + [(2 * 10) / 4] - 3) may now be written directly

(e.g., SET A = 1 + ((2 * 10) / 4) - 3).

---

[4]   Any mathematical function provided by Turbo Pascal can also be inserted within a MedState Notation statement using In-Line Pascal.

Variable Time Inputs (#T)

Time may be explicitly defined in terms of minutes (10' = ten minutes) or seconds with a whole or decimal number (3.5" = three and one half seconds). Variable time inputs using the #T command are also possible. Regardless of whether the time values are explicit or variable, time always serves as an input in MedState Notation. When it is desirable to change the value of a time variable, #T is preceded by a variable containing a specified amount of time.

Syntax: X#T: OUTPUT ---> NEXT

Where: X is a predefined variable; in this example it is set to a value of 1:

Example:

```
S.S.1,
S1,
   #START: ON 1; SET X = 1" ---> S2

S2,
   X#T: OFF 1 ---> S1
```

Please note, as with explicit time values, only one time command per state may be present, so the following example of code is illegal:

```
S1,
   X#T: ---> SX
   1": ---> SX
```

Tutorial 5: Creating an FI Schedule

Open Tutor04.mpc and make the following changes (changes noted in bold):

```
\ This is an FI
\ Filename, Tutor05.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


S.S.1,  \ Main Control Logic for "FI"
S2,                                              \ Changed S1 to S2
  #START: ON ^House; <WILL BE ADDING CODE HERE> ---> S3

S4,                                              \ Changed S2 to S4
  #R^LeftLever: ON ^Reward; Z1 ---> SX           \ Delete 5 Before #R


S.S.2,
S1,
  #START: SHOW 2,Responses,A ---> S2             \ Was SHOW 1

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A --->SX


S.S.3,
S1,
  #Z1: ADD B; SHOW 3,Rewards,B ---> S2           \ Was SHOW 2

S2,
  0.05": OFF ^Reward ---> S1


S.S.4,
S1,
  #START: ---> S2

S2,
  1': ---> STOPABORT
```

Next, add the "SET" code. In this program the variable "X" will be the fixed interval. The default will be 10, but later in this Tutorial the interval will be changed without changing the code. All of the "SET" code will be inserted into State Set 1:

```
S1,
  0.01": SET X = 10 ---> S2

S2,      \ Converts time into MED-PC clock ticks
         \ (Interrupts -- see User's Manual for additional
         \ information on runtime system).
  #START: ON ^House; SET X = X * 1" ---> S3
  1": SHOW 1,FI =,X ---> SX
```

Next add the time command #T with Variable X as State 3 of State Set 1:

```
S3,
  X#T: ---> S4
```

The final product should look like this:

```
\ This is an FI
\ Filename, Tutor05.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


S.S.1,  \ Main Control Logic for "FI"
S1,
  0.01": SET X = 10 ---> S2

S2,
  #START: ON ^House; SET X = X * 1" ---> S3
  1": SHOW 1,FI =,X ---> SX

S3,
  X#T: ---> S4

S4,
  #R^LeftLever: ON ^Reward; Z1 ---> S3


S.S.2,  \ This is the State Set that contains the Reward
        \ Count and Display
S1,
  #START: SHOW 2,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A ---> SX


S.S.3,  \ Reward Timer
S1,
  #Z1: ADD B; SHOW 3,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1
```

```
S.S.4,  \ Session Timer
S1,
   #START: ---> S2

S2,
   1': ---> STOPABORT
```

Save this file as Tutor05.mpc, translate and compile this program and then open it in MED-PC. Load the file, issue the START command, press the Left Lever repeatedly to get responses and reward counts on the screen. When one minute has passed and the program has shut down, the data is hanging in limbo. Print, save or delete the data, but DO NOT exit MED-PC. Instead, open the session again. The program is going to be changed to an FI-15.

Select **Configure** | **Change Variables** and the screen shown below will appear.

Figure 6.1 – Displaying Variables from Box 1

Enable Box 1 under Display Data from Box, as shown above, in order to display the variables.

The variable data for the displayed box can be changed from this screen. All Boxes can be changed by clicking "Select All", or select Boxes can be changed by clicking the appropriate number(s) in the "Additional Boxes to Update" section of the window.

For this tutorial, only one variable (the interval) on one Box (Box 1) needs to be changed. The code written for this program uses the variable "X" as the interval value, so click in the text Box next to the letter "X." In the lower right hand corner of the screen, there is a field titled **X from Box 1** whose default value is 10.

To change this program to an FI-15, replace 10 with 15 and click **Issue**. Note that the FI value displayed is now 15 on the runtime screen. Run or close the program.

From the Window's Desktop, reopen MED-PC and reload "Tutor05." Note that the runtime screen reads FI = 10. This is because the "Change Variables" screen does not change the code,

only the value of the variable for the procedure currently being run. Once MED-PC is closed the changes goes away.

It can be changed back upon reopening MED-PC or even changed multiple times in one session, depending on how the code is written. In the current example, changes made after the #START command is issued would result in an error unless the value changed is in "MED clock ticks."

CHAPTER 7 | IF STATEMENTS

Introducing the "IF" Statement

Until this point, the programs that have been written have all followed a pattern of, "do one thing until completed, then another, then another (e.g., the House Light blinking on and off) or time is up (e.g., Tutorial 5)."

By utilizing the "IF" command the programs can come to a proverbial fork in the road and the path they take is contingent on whether or not a criteria established are met. There are many variations to the IF command and this chapter will deal with three of them. As a result, there will be three versions of the Tutorial at the end of this chapter to illustrate how they would all work in the code.

An overview of "IF"

IF is an output command that compares the values of two numeric parameters, a numeric parameter and a variable, or two variables. The basic syntax of "IF" regardless of function, is[5]:

```
Syntax:      INPUT: IF P1 Operator P2 [@Label1, @Label2]

                    @Label1: Output ---> NEXT 1

                    @Label2: Output ---> NEXT 2
```

Where: P1 =   Constant, Number, variable, or array element.

P2 =   Constant, Number, variable, or array element.

Operator = Is one of six comparisons operators that are permitted: Equals (=), Less Than (<), Less Than or Equal To (<=), Greater Than (>), Greater Than or Equal To (>=), or Not Equal To (<>);

Label1 & Label2 = Any text label. Note, the @ must be present before the "Label" but the label itself is purely subjective.

If the comparison evaluates as TRUE, then the following statement (on the next line, i.e. @Label1) is executed. If the comparison is false, then the statement two lines down (i.e. @Label2) is executed.

---

[5] Please note, there are three syntaxes that can be used to write an IF statement. This chapter will show how to use the most complete syntax that can be used in any situation. Refer to Appendix A: | MedState Notation Commands for examples of how to use the other, abbreviated syntaxes.

IF as a session timer

Up to this point, a crude version of a session timer has been used. Since the programs were not very complex, this was not a problem. However, when there are a lot of things going on in a program, it is be possible for the screen data to disagree with the saved data (the saved data would be higher and correct, but the screen may not have time to update) or the program may stop in the middle of an event (like issuing a reward). Neither of these are ideal situations.

Assume that the experiment is running for sixty minutes, real code may look like this:

```
S.S.5,
S1,
  1": ADD N; IF N/60 >= M [@TrueEnd, @FalseContinue]
             @End: Z5 ---> S2
             @Cont: ---> SX

S2,
  3": ---> STOPABORT
```

This is adding the variable N every second and then converting the new value N to minutes. Since the session timer (represented by variable M) is set for 1 hour, or sixty minutes, this IF statement is set to stop the program at an hour or any fraction above it. If the session has been running for less than an hour, the system waits (coded for by @Cont: ---> SX). When an hour has passed, the program transitions to S2 where it waits three seconds before shutting down (therefore the screen can be updated, the reward can be given, etc.). A Z-pulse has been added that can be used where a function is terminated immediately (e.g., a response contingent statement or counter).

Nested IF commands

IF statements are not limited to only one set of options, they can also be nested. The syntax is nearly the same as a non-nested if, with the exception being that the nested commands must be organized sequentially:

```
1": IF A >= X [@True, @False]
      @True: IF B >= X [@True, @False]
               @True: IF C >= X [@True, @False]
                        @True:  ---> S2
                        @False: ---> S3
               @False: ---> S3
      @False: ---> S3
```

In the above example, all three variables (A, B, and C) must be greater than or equal to X for the statement to transition to S2. Any False outcome results in a transition to S3.

Compound IF commands

IF statements may also be constructed so that several logical conditions are evaluated in a single expression by placing each set of logical criteria in parentheses and connecting each set with AND, OR, NOT, AND NOT, or OR NOT. Parentheses must be used to denote the order in which expressions are evaluated if multiple expressions are strung together (note that this is like the way that parentheses control execution of algebraic expressions in SET statements). The syntax would look like this:

```
INPUT: IF (A = 1) AND (B = 2) [@True, @False]
        @True:  ---> S2
        @False: ---> SX
```

Real Code may look like this:

```
S.S.5,
S1,
  1": IF (R = 100) OR (M = 60) [@True, @False]
        @True:  ---> S2
        @False: ---> SX

S2,
  3": ---> STOPABORT
```

Tutorial 6A: Using A Single "IF" Command as a Session Timer

Open Tutor04.mpc and make the following changes noted in bold:

```
\ This is an FR schedule
\ Filename, Tutor06A.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  M = Minutes
\  X = Fixed Ratio
\  N = Session Timer


S.S.1,  \ Main Control Logic for "FR"
S1,
  1": SET M = 1, X = 5 ---> S2

S2,
  #START: ON ^House ---> S3

S3,
  X#R^LeftLever: ON ^Reward; Z1 ---> SX
```

```
S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 2,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A ---> SX


S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 3,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1
```

The position of the displays for the response and the reward counters have been shifted over one so that another counter can be added before them.

Also, a set of statements has been added in the beginning of the program that "define" the constants. Since these statements are preceded by a backslash, they are not part of the program, they are included for convenience. Once this program is translated and compiled, the values of M (Duration of the program) and X (the Fixed Ratio value) can be changed.

Now add the Session Timer as State Set 4. The code is:

```
S.S.4,  \ Session Timer
S1,
  #START: SHOW 1,Session,N ---> S2

S2,
  1": ADD N; SHOW 1,Session,N/60;
      IF N/60 >= M [@True, @False]
         @True: ---> S3     \ Therefore, when the Session Timer
                            \ >= M, time to Stop.
         @False: ---> SX    \ But if Session Timer < M, go no
                            \ where

S3,
  3": ---> STOPABORT
```

The program is ready to be saved (as Tutor06A.mpc), translated, and compiled. Open MED-PC and run the program. After seeing that it times out after a minute, but acts as a FR-5 when it is running, reload the Box with Tutor06A and change variables X to 10 and M to 1.5 before issuing the start command. Notice now that it runs as an FR-10 for a minute and a half.

Tutor06B.mpc

The primary purpose of this program is to see how a nested "IF" statement works. State 2 of State Set 4 contains the nested "IF" statement. First, a variable was SET in State Set 1 to allow something to nest. The comments indicate that variable Q will be the maximum number of rewards the animal will be allowed.

This program already had a Z-pulse to signal the Reward Timer. The new Z-pulse terminates certain program functions after the Session Timer runs out or the animal has enough rewards. Without the Z-pulse, the animal could get another reward and the response counter could still be counting after the procedure is "terminated" because of the three-second-time delay in S3 (needed to allow the screen to update before stopping).

```
\ NOTE: CHANGES IN BOLD ARE CHANGES FROM Tutor06A.MPC
\
\ This is an FR-5
\ Filename, Tutor06B.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Constants
\  A = Number of Responses
\  B = Number of Rewards
\  M = Minutes
\  X = Fixed Ratio
\  N = Session Timer
\  Q = Maximum Rewards


S.S.1,  \ Main Control Logic for "FR"
S1,
  1": SET M = 1, X = 5, Q = 5 ---> S2

S2,
  #START: ON ^House ---> S3

S3,
  X#R^LeftLever: ON ^Reward; Z1 ---> SX
  #Z2: ---> S1


S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 2,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A ---> SX
  #Z2: ---> S1
```

```
S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 3,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1


S.S.4,
S1,
  #START: SHOW 1,Session,N ---> S2

S2,
  1": ADD N; SHOW 1,Session, N/60;
      IF N/60 < M [@True, @False]
                \ As long as Session Time < M we will Continue
         @True: IF B >= Q [@2True, @2False]
                           \ If Animal has Enough Rewards we will Stop
                @2True: Z2 --->S3
                @2False: ---> SX
             \ Z-Pulse Added for when Elapsed Time = M.  It will
             \ send both S.S.1, S3 and S.S.2, S2 back to S1.
         @False: Z2 ---> S3

S3,
  3": ---> STOPABORT
```

Tutor06C.mpc

The primary purpose of this program is to see how a compound "IF" statement works. State 2 of State Set 4 contains the compound "IF" statement and simplifies the information in Tutor06B.mpc's into four lines (from six).

```
\ NOTE: CHANGES IN BOLD ARE CHANGES FROM Tutor06B.MPC
\
\ This is an FR-5
\ Filename, Tutor06C.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  M = Minutes
\  X = Fixed Ratio
\  N = Session Timer
\  Q = Maximum Rewards
```

```
S.S.1,  \ Main Control Logic for "FR"
S1,
  1": SET M = 1, X = 5, Q = 5 ---> S2

S2,
  #START: ON ^House ---> S3

S3,
  X#R^LeftLever: ON ^Reward; Z1 ---> SX
  #Z2: ---> S1


S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 2,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A ---> SX
  #Z2: ---> S1


S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 3,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1


S.S.4,
S1,
  #START: SHOW 1,Session,N ---> S2

S2,
  1": ADD N; SHOW 1,Session,N/60;
      IF (N/60 >= M) OR (B >= Q) [@True, @False]
        @True: Z2---> S3
        @False: ---> SX

S3,
  3": ---> STOPABORT
```

CHAPTER 8 | AN INTRODUCTION TO ARRAYS, PART ONE

The variables used so far have all been simple, non-array variables. Any variable (A to Z) can also be designated as an array with 2 to 1,000,001 elements. Although there are certain restrictions when using arrays, overall this is a very powerful means of collecting data and controlling the program. This chapter will deal using arrays to collect data.

The General Concept Behind Arrays

Once a variable has been assigned or defined as an array, the elements within that array are identified with subscripts of that variable where the first element is always numbered 0 (zero), and each successive element is consecutively numbered. The individual elements of an array are always accessed through subscripts.

In other words, the first piece of data in array "A" would be placed in element 0 and would be referenced by A(0), while the third piece of data would be placed in element 2 and referenced by A(2). If properly defined, this could continue up to the 1,000,001 piece of data that would be placed in element 1,000,000 and referenced by A(1000000).

The limit is 1,000,001 elements per Box (i.e., one array of 1,000,001 or two arrays of 500,000 each, one array of 500,000 plus 5 arrays of 100,000 each, etc.), so A(1000000) would have to be the last piece of data for not only the array, but for the Box and be the only variable defined.

DIM Command

The size of the array must be stated before the first State Set. As with all MSN variables, the values of array elements are always equal to 0 until explicitly changed. In a case where the program should fill in the array with data through the course of an experiment (i.e., the array is to be created empty), the DIM (dimensional) command is very useful.

Syntax: DIM = X

Where: X = The number of elements to define.

Comments: This command doesn't fit into the INPUT: OUTPUT ---> NEXT format, it must always be placed somewhere before S.S.1

Remember, arrays start with element zero, so an array with 25 elements to fill with data is written as:

Example:

```
DIM = 24
```

Using An Array To Record IRT's

An especially useful application of an array is the recording of Inter-Response Time (IRT) data. If the array is dimensioned smaller than the number of IRT's, an error message from MED-PC will appear when an attempt is made to use an array element that does not exist.

Sealing An Array

Arrays should be sealed so that only data recorded is displayed and all superfluous elements are excluded from the data file. This is done with the MedState Notation -987.987 command in conjunction with the "SET" command (e.g., SET X(250) = -987.987). The real code would look like this:

```
S2,
   #R^LeftLever: SET C(I) = T, T = 0; ADD I;
                 IF B >= Q [@True, @False]
                     @True: ---> S1
                     @False: SET C(I) = -987.987 ---> SX
```

In this example (which will be seen again in the Tutorial), every response adds to array C, and then the array is tested. If the statement is true, transition takes place to the first statement the collection of IRT's without terminating the procedure (Note: the program will end when the session timer tells it to do so). If the statement is false, however, the seal of the array is moved over one spot. The advantage, of course, being that the array is always "sealed" in case of a true statement or a premature stop, but the seal can always be moved.

Tutorial 7A: Using the DIM command

Open Tutor04.mpc and make the following changes marked in bold, save it as Tutor07A.mpc and translate/compile it.

```
\ This is an FR-5
\ Filename, Tutor07A.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  C = IRT Data Array


DIM C = 49


S.S.1,  \ Main Control Logic for "FR"
S1,
   #START: ON ^House ---> S2

S2,
   5#R^LeftLever: ON ^Reward; Z1 ---> SX
```

```
S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 1,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 1,Responses,A ---> SX


S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 2,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1


S.S.4,  \ Time Increment in 0.1 Second Intervals.
S1,
  #START: ---> S2

S2,
  0.1": SET T = T + 0.1 ---> SX


S.S.5,  \ Recording IRT's
S1,
  #START: ---> S2

S2,
  #R^LeftLever: SET C(I) = T, T = 0; ADD I ---> SX


S.S.6,  \ Session Timer
S1,
  #START: ---> S2

S2,
  1' ---> STOPABORT
```

After opening MED-PC, load and start the program. When testing it, do not press the lever more than 50 times (remember, DIM = 49). When the Box times out after one minute, select **File | Print**. The following will appear at the bottom of the print out:

```
C:
    0:       6.400        0.800        0.400        0.500        0.300
    5:       3.200        0.500        0.300        0.300        0.900
   10:       0.300        0.400        0.600        0.300        0.200
   15:       0.200        0.500        0.200        0.400        0.800
   20:       1.000        1.200        2.300        0.600        1.000
   25:       6.400        0.800        0.400        0.500        0.300
   30:       3.200        0.500        0.300        0.300        0.900
   35:       0.300        0.400        0.600        0.300        0.200
   40:       0.200        0.500        0.200        0.000        0.000
   45:       0.000        0.000        0.000        0.000        0.000
```

This is the data array. The numbers preceding the colon indicate the subscript of the first number per row. Each number that follows is the next subscript. Therefore, looking at the 0: row, C(0) = 6.400, C(1) = 0.800, C(2) = 0.400, etc.

Tutorial 7B: Sealing the Array

In this Tutorial, some changes are going to be made to Tutor07A.mpc. The changes to be made
are noted in bold.

```
\ NOTE: CHANGES IN BOLD ARE CHANGES FROM Tutor07A.MPC
\
\ This is an FR-5
\ Filename, Tutor07B.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1


\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  C = Data Array
\  M = Max Time in Minutes
\  N = Session Timer
\  Q = Max Rewards


DIM C = 999


S.S.1,  \ Main Control Logic for "FR"
S1,
  #START: ON ^House; SET Q = 5, M = 1 ---> S2

S2,
  5#R^LeftLever: ON ^Reward; Z1 ---> SX
  #Z2: ---> S1


S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 2,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A ---> SX
  #Z2: ---> S1


S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 3,Reward,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1
```

```
  S.S.4,  \ Time Increment in 0.1 Second Intervals.
  S1,
    #START: ---> S2

  S2,
    0.1": SET T = T + 0.1 ---> SX

  S.S.5,  \ Recording IRT's
  S1,
    #START: ---> S2

  S2,
    #R^LeftLever: SET C(I) = T, T = 0; ADD I;
                  SET C(I) = -987.987 ---> SX
    #Z2: ---> S1


  S.S.6,  \ Session Timer
  S1,
    #START: SHOW 1,Session,N/60 ---> S2

  S2,
    1": ADD N; SHOW 1,Session,N/60;
        IF N/60 < M [@True, @False]
           @True: IF Q <= B [@MaxRewards, @Continue]
                     @Max: Z2---> S3
                     @Cont: ---> SX
           @False: Z2 ---> S3

  S3,
    3": ---> STOPABORT
```

Save this program as Tutor07B.mpc then translate/compile it. Open MED-PC and load/run it. This program will now stop in one of two ways, when the animal has received the Max Rewards (defined by Q) or the program Max Time has been reached (defined by M), whichever happens first. Although the array is much larger number than necessary (1000 elements), the array will seal itself no matter how the program stops. When the program is done running, make a print out. Unlike the last program that had zeros in the array where there were no responses, this printout only shows the data collected. Below is an example of the data from an IRT = 42 :

```
C:
    0:        3.700        0.300        0.200        0.300        0.500
    5:        0.800        0.200        0.100        0.700        0.100
   10:        0.200        0.200        0.100        0.100        0.100
   15:        0.100        0.100        0.100        0.100        0.100
   20:        8.900        0.500        0.600        0.600        0.500
   25:        0.500        0.800        0.500        0.500        0.800
   30:        3.700        1.300        1.100        4.500        0.800
   35:        0.000       18.100        0.500        0.500        1.400
   40:        0.700        0.900
```

This was an example of the programming timing out as opposed to getting the maximum numbers of rewards. Unlike the print out from the Tutor07A.mpc program, this printout does not contain all of the excessive zeros due to the insertion of the -987.987 command.

CHAPTER 9 | ARRAY COMMANDS AS OUTPUTS

An introduction to the LIST, RANDI, AND RANDD commands

The previous chapter dealt with transforming variables into arrays by assigning them a dimensional value in order to collect and sort data. Arrays can be used for more than just data collection; they can also be used as control variables in a program. This chapter will deal with how to set up and use arrays in outputs.

LIST (as a definer of arrays)

Unlike DIM, which only allows the user to set up the shell of an array that must be filled in (or sealed), LIST allows the user to set up an array with assigned values. LIST is best used in conjunction with an output function (this will be demonstrated a bit later). When used to define, the syntax of LIST is:

```
LIST X = V1, V2, V3,...
```

Where X is any available variable and V1, V2, and V3 are all values assigned to the new array X.

LIST (as an output)

The second use of the LIST command is found in the output section of statements. This must be used in conjunction with LIST as a definer. The basic idea behind these two commands is that LIST first defines the array at the beginning of a program. Later in the program when drawing from this array, the LIST as an output will draw each number, one at a time and sequentially, until the program is done or the numbers have all been used. If the latter occurs, LIST simply starts again at the beginning of the list. The following would successively display the numbers 1, 2, and 3 on the screen, and demonstrates the two uses of LIST:

```
LIST G = 1,2,3

S.S.1,
S1,
  1": LIST F = G(H); SHOW 1,FVAL,F ---> SX
```

RANDI

RANDI is similar to LIST (as an output) and is used in to automatically select data from an array set up with LIST (as a definer). The difference between LIST and RANDI is that while LIST pulls its values from the array sequentially, RANDI pulls them randomly with replacement. The following example shows how the program above could use the RANDI command:

```
LIST G = 1,2,3

S.S.1,
S1,
  1": RANDI F = G; SHOW 1,FVAL,F ---> SX
```

Note that a subscript variable is not specified for the array variable, as was the case with the LIST command. The subscript is selected randomly as a function of RANDI. Also, unlike the LIST program that would present the data 1, 2, 3, 1, 2, 3,…, this program might cause the numbers 2, 2, 1, 3, 2, 1, 3 to be successively displayed on the screen (so the average, if allowed to run over time would be 1 = 2 = 3).

RANDD

RANDD is closely related to RANDI with the difference lying in that RANDD randomly selects from an array, but without replacement. Substituting RANDD in the examples of code used above might cause the numbers 1, 3, 2, 2, 1, 3, 2, 3, 1 to be successively displayed on the screen (any one number cannot be reused until the other two numbers have been selected).

Tutorial 8: Using the List as a Definer & RANDD to Set Up a VR Schedule

Open Tutor07B.mpc, make the changes shown below in bold and save as Tutor08.mpc. Translate/compile then open MED-PC and test the program.

```
\ This is a VR-10
\ Filename, Tutor08.mpc
\ Date: February 13, 2009

\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  C = Data Array
\  D = Variable Ratio Array
\  M = Max Time in Minutes
\  N = Session Timer
\  Q = Maximum Reward
DIM C = 999

LIST D = 1, 5, 10, 15, 19
```

```
S.S.1,  \ Main Control Logic for "VR"
S1,
  #START: ON ^House; SET Q = 10, M = 1 ---> S2

S2,
  1": RANDD X = D; SHOW 2,VR =,X ---> S3

S3,
  X#R^LeftLever: ON ^Reward; Z1 ---> S2
  #Z2: ---> S1


S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 3,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 3,Responses,A ---> SX
  #Z2: ---> S1


S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 4,Rewards,B ---> S2

S2,
  0.05": OFF ^Reward ---> S1


S.S.4,  \ Time Increment in 0.1 Second Intervals.
S1,
  #START: ---> S2

S2,
  0.1": SET T = T + 0.1 ---> SX

S.S.5,  \ Recording IRT's
S1,
  #START: ---> S2

S2,
  #R^LeftLever: SET C(I) = T, T = 0; ADD I;
                SET C(I) = -987.987 ---> SX
  #Z2: ---> S1

S.S.6,  \ Session Timer
S1,
  #START: SHOW 1,Session,N/60 ---> S2

S2,
  1": ADD N; SHOW 1,Session,N/60;
      IF N/60 < M [@True, @False]
         @True: IF Q <= B [@MaxRewards, @Continue]
                   @Max: Z2 ---> S3
                   @Cont: ---> SX
         @False: Z2 ---> S3

S3,
  3": ---> STOPABORT
```

As the program is run, notice that the value of VR is shown on the runtime screen. As long as at least five responses are generated the VR will equal 1, 5, 10, 15, 19 (not in that order) before repeating any of those numbers. This is because of the RANDD command. To test this, change RANDD to RANDI to make the selection random with replacement or change RANDD to LIST to get the numbers to come out sequentially.

CHAPTER 10 | VAR_ALIAS COMMAND

VAR_ALIAS

The VAR_ALIAS command allows for the creation of Named Variables. For example, the VAR_ALIAS command may be used to set the value of a variable named "Maximum Reward," rather than an obscure variable, such as "Q." Note that variable aliases do not have any use within the body of MSN programs and are simply directives placed before the first State Set that establish meaningful aliases (essentially synonyms) for program variables.

Syntax: VAR_ALIAS A = B

Where: A = A descriptive label for the variable or array element.

                B = The variable or array element.

Comments: This command doesn't fit into the INPUT: OUTPUT ---> NEXT format, it must always be placed somewhere before S.S.1)

Real code may look like this:

```
VAR_ALIAS SoftCR Data Array (Yes=1  No=0) = A(6)


DIM A = 6


S.S.1,
S1,
  0.01": SET A(6) = 1 ---> S2

S2,
  #START: ---> S3
```

The above code allows the user to whether or not to record SoftCR data before issuing the START command.

Tutorial 9: Using the VAR_ALIAS Command

Open Tutor08.mpc, make the changes are shown below in bold. Save as Tutor09.mpc and translate/compile.

```
\ This is a VR-10
\ Filename, Tutor09.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  C = Data Array
\  D = Variable Ratio Array
\  M = Max Time in Minutes
\  N = Session Timer
\  Q = Maximum Reward


VAR_ALIAS Session Time             = M  \ Default = 1 minute
VAR_ALIAS Maximum Number of Rewards = Q  \ Default = 10


DIM C = 999

LIST D = 1, 5, 10, 15, 19


S.S.1,  \ Main Control Logic for "VR"
S1,
  0.001": SET Q = 10, M = 1 ---> S2

S2,
  #START: ON ^House ---> S3

S3,
  1": RANDD X = D; SHOW 2,VR =,X ---> S3

S4,
  X#R^LeftLever: ON ^Reward; Z1 ---> S2
  #Z2: ---> S1
```

```
S.S.2,  \ This is the State Set that contains the Response
         \ Count and Display
S1,
   #START: SHOW 3,Responses,A ---> S2


S2,
   #R^LeftLever: ADD A; SHOW 3,Responses,A ---> SX
   #Z2: ---> S1



S.S.3,  \ Reward Timer, Count, and Display
S1,
   #Z1: ADD B; SHOW 4,Rewards,B ---> S2

S2,
   0.05": OFF ^Reward ---> S1



S.S.4,  \ Time Increment in 0.1 Second Intervals.
S1,
   #START: ---> S2

S2,
   0.1": SET T = T + 0.1 ---> SX



S.S.5,  \ Recording IRT's
S1,
   #START: ---> S2

S2,
   #R^LeftLever: SET C(I) = T, T = 0; ADD I;
                 SET C(I) = -987.987 ---> SX
   #Z2: ---> S1



S.S.6,  \ Session Timer
S1,
   #START: SHOW 1,Session,N/60 ---> S2

S2,
   1": ADD N; SHOW 1,Session,N/60;
       IF N/60 < M [@True, @False]
          @True: IF Q <= B [@MaxRewards, @Continue]
                    @Max: Z2 ---> S3
                    @Cont: ---> SX
          @False: Z2 ---> S3

S3,
   3": ---> STOPABORT
```

Start MED-PC and use the Experiment Loading Wizard to load the Tutor09.mpc program into Box 1. If the Wizard does not start automatically when MED-PC is started, then start it manually by selecting **File** | **Wizard for Loading Boxes**.

When the Alter Session Parameters screen of the Experiment Loading Wizard, shown below, is reached the "Session Time" and the "Maximum Number of Rewards" can be changed, which will effectively be changing the value of the variables M and Q respectively.

Figure 10.1 – Alter Session Parameters Screen
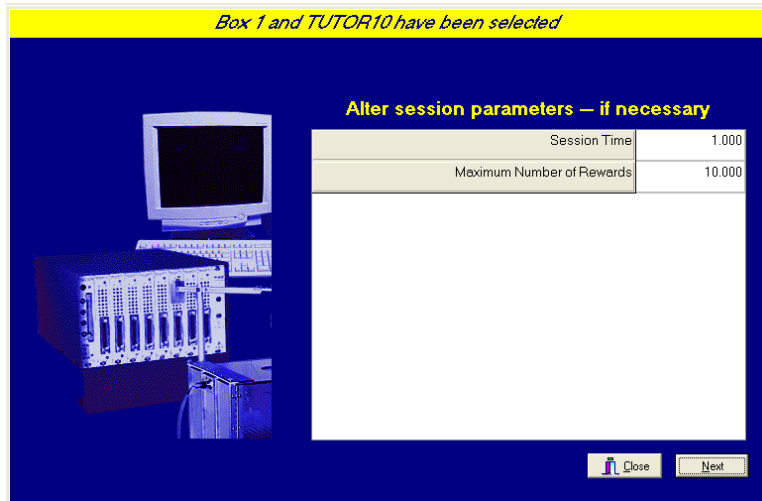
The Named Variables can be changed once MED-PC is started by selecting **Configure | Change Variables**. Select the appropriate Box and either change the variables M and Q directly or click on the **Named Vars** button. This button will produce the screen shown below where the values of the Named Variables can be changed. Click **Issue** to confirm any changes.

Figure 10.2 - Displaying Named Variables for Box 1
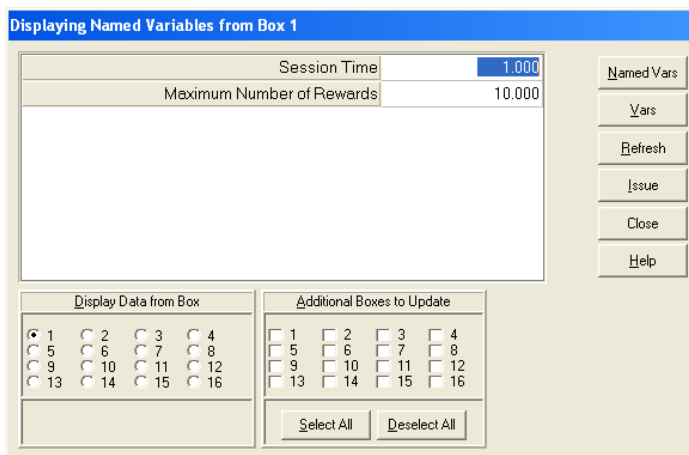
This concludes Tutorial 9. Appendix A contains a more in-depth description of the commands presented here.

CHAPTER 11 | THE DATA HAS BEEN COLLECTED, NOW WHAT?

Print and Disk Commands

Zeros appear next to unassigned variables, which can result in cluttered data file. MED-PC allows the user to establish which data are to be printed and/or saved. There are times when the printing (PRINT) and data saving (DISK) commands overlap. When this occurs, only the syntax of the PRINT command will be shown, but it will be explained that DISK can be used in place of PRINT.

All of these commands, unless explicitly stated, are written as stand-alone statements and must precede the first State Set (Like the DIM command from Chapter 8 or the VAR_ALIAS command from Chapter 10).

Setting the Orientation of Printouts (PRINTORIENTATION)

This command is used to override system defaults with respect to whether a given printout occurs in Landscape (sideways) or Portrait (standard) orientation.

Syntax: PRINTORIENTATION = direction

Where: Direction is "Landscape" or "Portrait."

Setting the # of Columns on Printouts (PRINTCOLUMNS) [on Data files (DISKCOLUMNS)]

PRINTCOLUMNS controls the number of columns in which the contents of arrays are printed. The use of this command will override any defaults set within the MED-PC menu system (which is five columns).   This command functions in combination with PRINTORIENTATION, PRINTPOINTS, and PRINTFORMAT. If the total line space available is exceeded, the column function may be automatically truncated.

Syntax: PRINTCOLUMNS = X

Where: "X" is the number of columns.

The DISKCOLUMNS command will do the same thing but to the data file. Its syntax is identical.

Controlling Font Size on Printouts (PRINTPOINTS)

PRINTPOINTS controls the size of the font used to print data from the Box in which this command is issued. The use of this command will override any defaults set within the MED-PC menu system.

Syntax: PRINTPOINTS = X

Where: X is the number of points (12 is the default).

Controlling the Printouts/Data Files (PRINTFORMAT)/(DISKFORMAT)

By default, MED-PC automatically sets aside 12 spaces for each number to be printed. It breaks down the 12 spaces into 8 reserved for the integer part of the number (to the left of the decimal), 1 for the decimal and 3 spaces for numbers right of the decimal. An example of a number printed in 12.3 format (the meaning of 12.3 will be detailed below) is, "12345678.123."

In many instances, it is useful to print data in other formats, particularly when trying to increase the amount of data printed per page. Placing a PRINTFORMAT statement before the first State Set of the procedure allows the user to control the printed format of numbers.

Syntax: PRINTFORMAT = P1.P2

Where: P1 =   The number indicates the total number of spaces to be occupied by the number including the decimal point.

                P2 =   The number indicating the number of spaces to be set aside for the decimal portion of the number.

PRINTFORMAT Examples

```
PRINTFORMAT = 5.1  \ Print in five space, with 3 to left of decimal
                   \ 1 to right as in 123.1

PRINTFORMAT = 7.2  \ e.g., 1234.12

PRINTFORMAT = 6.0  \ e.g., 123456
```

The use of a PRINTFORMAT statement has no effect upon the internal representation of numbers. If multiple PRINTFORMAT statements are used in the same .MPC procedure, then only the last one is implemented.

If the digits to the left of the decimal point exceed the total number of spaces set aside by the PRINTFORMAT statement, then the general formatting rules are temporarily set aside and the number is printed in as many spaces as are needed to represent the integer portion of the number. This may result in the printed line "spilling" onto the next line on the page. If the decimal portion of a number exceeds the space allocated, the number printed is rounded to the nearest value.

To save the data in the same format replace the word PRINT with DISK and follow all of the same rules.

Controlling the Selection of Variables or Arrays on Printouts/Data Files (PRINTVARS)/(DISKVARS)

It is often desirable to print only a subset of the variables and arrays in a procedure. This is particularly true when many of the variables are used internally by the procedure and do not contain data. Additionally, when collecting hundreds or thousands of data points per session, it would be convenient to be able to print a few key indices to the printer after every session, and still be able to save the detailed counters to disk file for later analysis.

The above objectives may be accomplished using the PRINTVARS command. This command may be used to declare a list of variables that will be printed whenever a PRINT command is issued. The PRINTVARS command affects printing irrespective of whether the command to print was issued from within a state table or by a keyboard command. The PRINTVARS command in no way affects the variables that will be written to disk (but a parallel command, DISKVARS, is provided).

As seen in previous Tutorials, by default all variables and arrays (A-Z) are printed. To print selected variables, place a PRINTVARS directive before the first State Set of the procedure. PRINTVARS must come before the first State Set.

Syntax: PRINTVARS = P1, P2, ..., P26

Where: P1...P26 are the variables or arrays A through Z to be printed.

Real Code may look like this:

```
PRINTVARS = A, B, C, D, F, G, Z
```

Condensed vs. Full Headers (PRINTOPTIONS)

PRINTOPTIONS provides control over the appearance of the headers that appear at the beginning of printouts, as well as when to print the data. The headers include information such as the time the experiment was loaded and the name of the program used to control the experiment.

There are two options for the appearance of headers: FULLHEADERS vs. CONDENSEDHEADERS.

Options for when to print the data include FORMFEEDS or NOFORMFEEDS. If FORMFEEDS is selected, the printout will occur when data is queued to be printed (i.e., when a Box is done running). If NOFORMFEEDS is selected, MED-PC will print only when a page of data is in the queue. If PRINTOPTIONS is not explicitly specified, the default printout is to print a condensed header and no form feed. When specified, commas separate multiple options, and any option not specified will stay at its default value.

Syntax: PRINTOPTIONS = P1, P2

Where: P1 =   FULLHEADERS or CONDENSEDHEADER.

                    P2 =   FORMFEED OR NOFORMFEED.

Printing Data (PRINT)

PRINT is the only command in this section that is not placed before State Set 1. It is an output command that may be used to generate printouts from within the code. Just like with printing from the menu bar, unless specified differently (using the aforementioned commands), PRINT will print everything.

All printing is done through the Windows Print Manager. In the event that the printer is offline or out of paper or there is some other problem, Windows will present a Dialog Box indicating the nature of the problem. It is generally best to correct the problem and then select "RETRY." Data will not generally be lost under such circumstances.

Syntax: INPUT: PRINT ---> NEXT

Real code may look like this:

```
S2,
  30': PRINT ---> STOPABORTFLUSH
```

Tutorial 10: Bringing it all Together

Open Tutor09.mpc, make the changes noted in bold and save it as Tutor10.mpc:

```
\ This is a VR-10
\ Filename, Tutor10.mpc
\ Date: February 13, 2009


\ This section is for Inputs
^LeftLever = 1

\ This section is for Outputs
^House  = 1
^Reward = 2  \ In this code, this is a Pellet Dispenser.


\ Defined Variables
\  A = Number of Responses
\  B = Number of Rewards
\  C = Data Array
\  D = Output Array
\  M = Max Time in Minutes
\  N = Session Timer
\  Q = Maximum Reward

VAR_ALIAS Session Time               = M  \ Default = 1 minute
VAR_ALIAS Maximum Number of Rewards = Q  \ Default = 10

DIM C = 999

LIST D = 1, 5, 10, 15, 19

PRINTORIENTATION = LANDSCAPE
PRINTCOLUMNS     = 4
PRINTOPTIONS     = FULLHEADERS, FORMFEEDS
PRINTVARS        = A, B, C


S.S.1,  \ Main Control Logic for "VR"
```

```
S1,
  0.001": SET Q = 10, M = 1 ---> S2

S2,
  #START: ON ^House ---> S3

S3,
  1": RANDD X = D; SHOW 2,VR =,X ---> S3

S4,
  X#R^LeftLever: ON ^Reward; Z1 ---> S2
  #Z2: ---> S1

S.S.2,  \ This is the State Set that contains the Response
        \ Count and Display
S1,
  #START: SHOW 3,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 3,Responses,A ---> SX
  #Z2: ---> S1

S.S.3,  \ Reward Timer, Count, and Display
S1,
  #Z1: ADD B; SHOW 4,Rewards,B ---> S2

S2,
  0.05":OFF ^Reward ---> S1


S.S.4,  \ Time Increment in 0.1 Second Intervals.
S1,
  #START: ---> S2

S2,
  0.1": SET T = T + 0.1 ---> SX


S.S.5,  \ Recording IRT's
S1,
  #START: ---> S2

S2,
  #R^LeftLever: SET C(I) = T, T = 0; ADD I;
                SET C(I) = -987.987 ---> SX
  #Z2: ---> S1


S.S.6,  \ Session Timer
S1,
  #START: SHOW 1,Session,N/60 ---> S2

S2,
  1": ADD N; SHOW 1,Session, N/60;
      IF N/60 < M [@True, @False]
          @True: IF Q <= B [@MaxRewards, @Continue]
                   @Max: Z2 ---> S3
                   @Cont: ---> SX
          @False: Z2 ---> S3

S3,
  5": PRINT ---> STOPABORTFLUSH
```

Translate and compile the program, open it in MED-PC and run. After one minute a print out of the data specified should be produced in a landscape format. The data file, on the other hand, will contain all variables and arrays in the default format. This is the first program written with a STOPABORTFLUSH command in it. This command is used so a data file would be made to compare to the printout. The data file was saved using MED-PC's automatic naming system. If necessary, consult the MED-PC User's Manual for an explanation of how and where MED-PC saves files.

This concludes Tutorial 10.

CHAPTER 12 | UNDERSTANDING HOW MED-PC WORKS

Time-Based Interrupts

MED-PC is an interrupt-based system. Most of the time, MED-PC is occupied by performing non-critical, non-experimental operations such as responding to user keystrokes, displaying the output of SHOW commands, writing to disk and the printer, etc. Periodically, these activities are "interrupted" and attention is shifted to active Boxes. These interrupts occur immediately; regardless of what actions the computer is performing. Even in the middle of writing to disk, the occurrence of an interrupt immediately shifts attention to the experimental Boxes.

The frequency with which interrupts occur (and Boxes are serviced) is equal to the system resolution value declared during the "Hardware Configuration." For example, if the resolution is set to 10 ms, the Boxes are serviced every 10 ms. In the discussions that follow, it will be assumed that a system with 10 ms resolution is applicable. These timed-based interrupts are generated by crystal-controlled interrupt hardware on the interface card that plugs into the chassis of the PC.

Noting and Reacting to Inputs

As soon as an interrupt occurs, any ongoing activity is suspended and processing of all active Boxes commences. Before any individual Boxes are processed, the status of all inputs is read and recorded. Thus, if a #R1 (response) has occurred in Box 1 and a #R1 has occurred in Box 3, these events will be noted and made available to the respective Boxes when the Boxes are serviced (soon to commence). Any keyboard #R's presented since the initiation of the last processing sweep will be merged with any that were read from the input cards.

For example, if a keyboard #R1 was recently generated for Box 3 and a hardware #R2 for Box 3 was also recorded, both #R1 and #R2 will be presented to Box 3 during the present sweep. Only one instance of a given response for a single Box may occur during a single sweep. Thus, if #R1 was issued from both the keyboard and was present on the interface for the same Box since the last interrupt, only one instance of the #R1 will be presented to the Box. A statement of the form "2#R1: ---> S2" would require a response on another sweep in order for a transition to S2 to occur. Similarly, if the subject responded twice on the same input between the occurrences of two interrupts, only one response would be counted.

However, responses will not be missed for the following reasons:

- Responses are latched in the hardware buffer until read (contact does not have to be coincident with the interrupt).

- A resolution of 10 milliseconds is far faster than subjects can respond.

- The minimum response time is not dependent on resolution, but rather the response frequency of the hardware. MED Associates SmartCtrl Cards use a response frequency of 100 Hz whereas Standard and SuperPort modules range from 7 Hz to 400 Hz.

- Two responses occurring less than 10 ms apart will actually be resolved if one occurs before a given interrupt (processing sweep) and the other occurs after the interrupt.

Further discussion of theoretical vs. practical timing resolutions is provided at the end of this section.

Order Of Processing Of Boxes

After preliminary events have occurred (i.e., recording of inputs) individual Boxes are serviced in sequential order, beginning with the first active (loaded) Box. Please note that inactive (unloaded) Boxes receive no processing.

Order Of Processing Of Events Within A Box

Once processing of a Box's State Sets begins, the "First" State Set of the procedure is serviced. Next, the remaining State Sets are processed in the order in which they appear in the .MPC procedure file or "State Table," not by the assigned numerical label. Processing then proceeds to the next active Box. For example in the following code, S.S.2 will be processed prior to S.S.1:

```
S.S.2,
S1,
  #R1: ADD A ---> SX


S.S.1,
S1,
  #R2: ADD B ---> SX
```

Processing of States

The State within each State Set is processed depending on the "current" State of any given State Set. When loaded, the first State listed will always be the "current" State of any procedure until the input requirements of that State are met and a transition occurs. Again, this is independent of the numbering of the States. In the State Set that follows, S10 will be the current state when the procedure is loaded and will remain "current" until #R1 occurs:

```
S.S.3,
S10,
  #R1 ---> S5

S5,
  1": ADD A ---> S10
```

Processing of Statements within a State

As indicated above, processing of an .MPC procedure file starts with the first State Set listed as being the current State. Processing of statements within a State also proceeds from the top down with the caveat that those statements associated with external inputs (i.e. #R, #K, ", ', #T, and #START) are processed before those statements associated with internal inputs (i.e., Z-Pulses). Within each current State, processing continues until a statement is encountered in which the stated input condition has been met or until the last statement has been reached.

In the following example, S1 of S.S.5 begins by ignoring the Z-pulse in line 3. Assuming a #R1 has occurred prior to the initiation of the current sweep, the internal variable that tracks the total number of responses on R1 in S.S.5, S1 is incremented. The current State remains S1 since two #R1s are required to cause a transition to S2. Additionally, processing of the State proceeds downward to line 5 because the input requirement was not satisfied and no transition (either to SX, the same State or to a different State) has occurred. If an #R2 occurs, the #R2 input count will be incremented, but, since three #R2s are required, processing continues within S1. Line 8 is then processed, and if Z1 is issued (i.e., if 1 second has elapsed), a second "sweep" of the procedure begins in which only #Z-pulse inputs are processed. Now, processing of S.S.5, S1 issues a transition to S3 (not shown). If S.S.5, S1 is re-entered all counters are reset to zero.

Example:

```
S.S.5,               \ Line 1
S1,                  \ Line 2
  #Z1:  ---> S3      \ Line 3
  2#R1: ---> S2      \ Line 4
  3#R2: ---> S4      \ Line 5


S.S.6,               \ Line 6
S1,                  \ Line 7
  1": Z1 ---> SX     \ Line 8
```

A Review of the General Principles

Although it may take a few minutes to read this review, all of this is occurring in the FIRST MILLISECOND of each 10 ms interrupt.

1. External inputs are processed.

   A. External inputs refer to:

      i.   #R   (Used to input a response via interface modules)

      ii. #K   (Used to input a "signal" from another Box)

      iii. "   (Used with a numerical value to time in seconds, e.g., 5")

      iv. '   (Used with a numerical value to time in minutes, e.g., 2')

      v.   #T   (Used with a variable to define a timed input)

      vi. #START   (A user issued command)

2. Z-Pulses are ignored during the processing of "external" inputs.

3. Processing is done in a top-down fashion.

   A. The first State Set listed is processed first, followed by the subsequent State Sets in the order LISTED; State Set numbers (1 to 32) do not determine processing order.

   B. Within a State Set, the "current" State is processed.

      i. The current State at the beginning of program execution is that which is physically first in the State Set.

      ii. The current State is changed as the result of transitions that occurs when a statement's input requirements are satisfied. A state change resulting from an external input becomes the "current" state for the processing of Z pulses.

   C. Statements within a State Set are processed in a top-down fashion, with the proviso, that Z-Pulses are ignored during the processing of external inputs.

4. Processing of a State stops as soon as a statement's input requirements are satisfied.

   A. As inputs (K's, R's, and START's) are encountered, the counters associated with them are incremented as appropriate. As soon as the input side of a statement is satisfied, any subsequent counted inputs do not have their counters incremented.

      i. This is true irrespective of whether the satisfied statement is performing a transition to SX, the same State or a different State.

      ii. Although there are no counters associated with time-based inputs, the effect of a time-based transition is analogous to that of the "counted" inputs in that satisfaction of a time-based input also halts further processing of the present State.

5. All Z-Pulses issued in the output section of statements during the processing of external inputs are noted and held for use as input during the Z-pulse processing phase.

A. Only one instance of a given Z-pulse is recorded during processing of external inputs, but more than one different Z-pulse may be counted. If, for example, a time-based statement issues two #Z1's in its output section, the effect of doing so does not differ from issuing a single #Z1. Similarly, issuing #Z1 from multiple States is equivalent to issuing a single #Z1.

6. After the completion of processing of external inputs, one or more passes is made through the State Table, provided that at least one #Z-pulse was issued during the external-processing phase.

7. The current State of a given State Set during Z-pulse processing is that State it was left in at the conclusion of external-input processing. Thus, if the current State of a State Set changes from State 1 to State 2 during external-processing, #Z-Pulses will be processed in State 2 during Z-pulse processing.

8. During Z-pulse processing, only Z-Pulses serve as input, all other inputs are ignored.

9. Processing priority rules for stacked Z-Pulses are analogous to those for external inputs.

10. Any new Z-Pulses issued during Z-pulse processing are held until the bottom of the State Table is reached, at which time a new Z-pulse processing pass will be initiated.

A. During any given Z-pulse processing pass, only Z-Pulses generated during the immediately preceding pass will be presented during the present pass.

B. Up to 9 consecutive Z-pulse processing passes may occur. If a tenth pass is required to resolve the actions of the State Table, processing of the State Table will be terminated and the on screen error indicator will be activated, with a corresponding entry made in the Journal. This is done to avoid the occurrence of "endless loops" which could indefinitely delay processing of events in other Boxes. The Box will, however, be processed at the beginning of the next processing sweep.

C. Within a State Set, a new State entered during a given Z processing pass becomes the current State when (and if) a subsequent Z processing pass occurs. Thus, if transition from S1 to S2 occurs as the result of a Z-pulse, and another processing pass is occasioned by the generation of Z-Pulses during the earlier pass; S2 will be the current state in which further Z-Pulses may be detected. The final State that results from transitions during Z-pulse processing will remain; of course, the "current" State when external inputs are processed upon occurrence of the next interrupt.

Examples

The following code results in a SHOW display, as soon as K1 is issued, of "A Value" with a value of "1" in position 1 and "C Value" incrementing in position 3. "B Value" is never displayed and subsequent occurrences of K1 are not reflected in "A Value" since S.S.1 remains in S3 after processing is complete.

EXAMPLE A:

This illustrates that processing of Z-Pulses continues, and progressions through more than one State may occur during Z-pulse processing.

```
S.S.1,
S1,
  #Z1: Z2 ---> S2
  #K1: ADD A; SHOW 1,A Value,A ---> SX

S2,
  #Z2: ---> S3  \ #Z2 detected in same Clock Tick in
                \ which it is issued.
  0.01": ADD B; SHOW 2,B Value,B ---> SX
                \ Never executed
                \ #Z2 always occurs immediately

S3,
  1": ADD C; SHOW 3,C Value,C ---> SX


S.S.2,
S1,
  #K1: Z1 ---> SX
```

EXAMPLE B:

The following code changes the second statement in State Set 1 so that transition occurs to S1 (replacing SX). The result, however, is exactly the same as in Example A.

```
S.S.1,
S1,
  #Z1: Z2 ---> S2
  #K1: ADD A; SHOW 1,A Value,A ---> S1

S2,
  #Z2: ---> S3
  0.01": ADD B; SHOW 2,B Value,B ---> SX

S3,
  1": ADD C; SHOW 3,C Value,C ---> SX


S.S.2,
S1,
  #K1: Z1 ---> SX
```

EXAMPLE C:

In this code, issuing K1 places S.S.1, S1 into S4. Hence, when the Z1 is issued in S.S.2, S.S.1 S1 is no longer active – S4 becomes the active state. As a result, B and C are not displayed when K1 is issued.

```
S.S.1,
S1,
  #Z1: Z2 ---> S2
  #K1: ADD A; SHOW 1,A Value,A ---> S4

S2,
  #Z2: ---> S3
  0.01": ADD B; SHOW 2,B Value,B ---> SX

S3,
  1": ADD C; SHOW 3,C Value,C ---> SX

S4,
  1": ADD D; SHOW 4,D Value,D ---> SX


S.S.2,
S1,
  #K1: Z1---> SX
```

EXAMPLE D:

In the following code, #K1 causes a transition to S5 in S.S.1. In S.S.2, it generates #Z1. During the Z-pulse-processing phase, S.S.1 is in S5 so the Z1 required for transition is received and "In State 5: 1" is displayed on the screen, upon issuance of the first #K1 from the keyboard. A second #K1 will display a 2, a third, 3, etc.

```
S.S.1,
S1,
  #Z1: Z2 ---> S2
  #K1: ADD A; SHOW 1,A Value,A ---> S5

S2,
  #Z2: ---> S3
  0.01": ADD B; SHOW 2,B Value,B ---> SX

S3,
  1": ADD C; SHOW 3,C Value,C ---> SX

S5,
  #Z1: ADD E; SHOW 5,In State 5,E ---> SX


S.S.2,
S1,
  #K1: Z1 ---> SX
```

Additional Commentary on Time Based Inputs

Time-based inputs (", ', and #T) are subject to the same rules and considerations with respect to order of processing and the consequences of stacking as is true of the other external inputs (#K, #R, and #START), but a few points may not be readily apparent.

1. Timers continue to time even when stacked with other external inputs, but cannot cause a transition (to SX, to the same State, or to another State) unless they are processed prior to other inputs for which an input actually occurred. Once a timer has elapsed, it will continue to be eligible to cause a transition, provided that the current State has not changed, until it is the first statement capable of causing a transition.

2. It is never a good idea to stack timers with durations equal to the system resolution above other external inputs, for the other inputs will never receive processing because the timer will always cause a transition.

3. Timers with durations equal to the system resolution may be stacked beneath other inputs. Note, intervals specified for less than the resolution interval (the interrupt sweep interval) will be processed as though they were equal to the resolution interval.

4. In the following situation, imagine a system resolution of 10 milliseconds and that an #R1 occurs and the 10" time duration times out on the same interrupt sweep (within the 10 millisecond window between interrupts). In this unlikely occurrence, A will be added and transition to S2 will occur during the subsequent interrupt, or 10.01" after entry into the state, provided that another #R1 does not occur within 10 milliseconds of the first response which is probably physically impossible. Stated differently, transition to S2 will not occur until a clock tick occurs in which no #R1 has occurred and the elapsed time is >= 10" since entry into S1.

```
S.S.1,
S1,
  #R1: ADD A ---> SX
  10": ---> S2
```

Accuracy of the MED-PC System

The MED Associates, Inc. millisecond timer, which generates the interrupt signal, features a precise microsecond crystal with an accuracy of 0.001%. Keep in mind that all PC's are sequential processors and therefore can only do one thing at a time, although by virtue of their speed they appear to do multiple tasks simultaneously. The actual processing speed of any system depends upon the number and complexity of the procedures run, but the resolution of timed events is determined by the interrupt interval (resolution) set during the running of the Hardware Configuration Utility.

Some users may wish to think of this as one clock tick. Theoretically, it is possible to err in timing by one clock tick -- as is the case with all timing systems. In practical applications however this becomes a concern only when several conditions are met (i.e., the processing time

for all Boxes is both inconsistent and at times approaches the resolution value, and time durations have been set equal to the resolution value). The MED-PC system provides a speed warning system that monitors the average processing or sweep time. This does not inhibit the performance of the system in any way, but alerts the user to possible procedure shortcomings before they become a problem. Remember, this is an interrupt-based system. Interrupts are extremely precise and all external inputs (both responses and timed events) are serviced prior to further processing.

At the risk of making MED-PC appear less accurate than it really is, the following illustration demonstrates a "worst case" situation for MED-PC. Consider a computer that just barely keeps up with the 10-millisecond resolution set during installation and 16 Boxes are running the same procedure. This procedure is designed to turn an output alternately turned on and off every 10 ms. The result is that Box 1 performs exactly as expected, but the timing in Box 16 is somewhat less precise. This is because Box 1 is always serviced immediately after initiation of a sweep. In contrast, the processing of Box 16 is dependent on the amount of time it takes to process the preceding 15 Boxes on each sweep. A scenario in which a sweep is initiated illustrates the potential problems that this could pose, and Boxes 1 through 15 require 9 ms to process. Box 16's output would be toggled on 9 ms after initiation of the sweep and the next sweep would occur 1 ms later. If on the second sweep, Boxes 1 through 15 required very little time to process, such that Box 16 is serviced 1 ms after initiation of the sweep, its output would be turned off 2 ms after being turned on (rather than the 10 ms separation specified by the procedure).

NOTE: This is not a cumulative source of errors and would never be any greater than the resolution value. An extremely important point to bear in mind is that this discussion reflects a very unlikely situation in which the system oscillates from moment-to-moment between the Boxes requiring substantial time to process and the Boxes requiring minimal processing time. The actual behavior of a MED-PC system can be expected to be closer to one in which any given Box is processed at intervals approximating the nominal resolution value because the demands of the Boxes average out, with some Boxes active on one sweep and others active on the next. This is especially true when the Boxes have timers with minimum durations of at least twice the resolution value. The error, in practice, will also usually be much smaller than +/- the resolution value provided that the average sweep duration (displayed on screen by the "A:" indicator on the status line) is appreciably less than the resolution value.

CHAPTER 13 | MACROS

What are Macros and Why Should I Use Them?

The macro feature is one of the key aspects of the system. Macros are text files that automate sequences of operator actions. A common use for macros is to automate the loading of a set of boxes, along with the setting of key session parameters. Virtually any command that can be issued from the keyboard can also be issued from a macro file.

Creating Macros

To create a macro, turn on the macro recorder, use the menu system to carry out whatever tasks should be automated, turn off the macro recorder and save the macro to a file.

Turning On/Off the Macro Recorder

The macro recorder may be turned on by either clicking the cassette icon on the tool bar or by selecting the **Macros | Turn On Macro Recorder**. During recording, the text "Recording Macro" appears at the bottom of the main window of MED-PC.

The macro recorder may be turned off by either clicking the cassette icon on the tool bar or by selecting the **Macros | Turn Off Macro Recorder**. Selecting this option presents a file dialog that allows the user to specify the file name and directory for the macro.

Insert Macro Playback Delay…

This option is used to insert a time delay into a macro so that the macro playback will pause for the specified time duration. To insert a macro playback delay select **Macros | Insert Macro Playback Delay** (this menu option is only available while recording a macro). The duration of the delay must be specified in milliseconds. No delay will occur while the macro is being recorded.

Syntax: DELAY A

Where: A =   The number of milliseconds for which macro playback should be delayed.

The following example will pause the macro for 1 second:

```
DELAY 1000
```

Example of When to Use the DELAY Command

This command can be useful when it is necessary to wait for a program to complete some action before the macro continues. For example, an MSN program might be written so that it immediately sets default values for variables. It would be convenient to be able to use a macro to load the program, allow the defaults to be set and then over-ride some of the values. Without a time delay between loading the program and over-riding the defaults, it would be possible to change the variables in the macro and to then have the MSN program change the values back to their defaults. Consider the following MSN program:

```
\ FR Program
\ A = FR Size

S.S.1,
S1,
  0.01": SET A = 10 ---> S2

S2,
  A#R1: ON 1 ---> S3

S3,
  0.1": OFF 1 ---> S2
```

This program arranges a simple FR. Ten milliseconds after loading, "A" is set to 10. In S2, "A" responses on input 1 turns on output 1 (presumably connected to a pellet dispenser) and transitions to S3. S3 turns the output off after 100 milliseconds and returns to S2 for another ratio run. Now consider the following macro:

```
LOAD BOX  1 SUBJ 1 EXPT FR Demo GROUP 2 PROGRAM A2
SET A VALUE 20 MAINBOX 1 BOXES 1
```

If this macro is executed on a fast computer, it is possible that the Box will load and "A" will be set to 20 in less than 10 milliseconds. If that happens, then 10 milliseconds after the program loads, S1 of the program will set "A" back to 10.

The following macro avoids this problem by introducing a delay of 20 milliseconds after the Box is loaded.

```
LOAD BOX  1 SUBJ 1 EXPT FR Demo GROUP 2 PROGRAM A2
DELAY 20
SET A VALUE 20 MAINBOX 1 BOXES 1
```

Editing Macros

Existing macros may be edited by selecting **Macros | Editor** from the main menu. A simple text editor will then appear, and existing macros may be opened for editing using the editor's File menu.

One of the best ways to edit existing statements within a macro is to use the editor's powerful editing templates by positioning the cursor on the line that needs to be edited and then selecting **Insert | Edit Command at Cursor**.   The screen shown below will appear.

Figure 13.1 – Macro Editor - Load
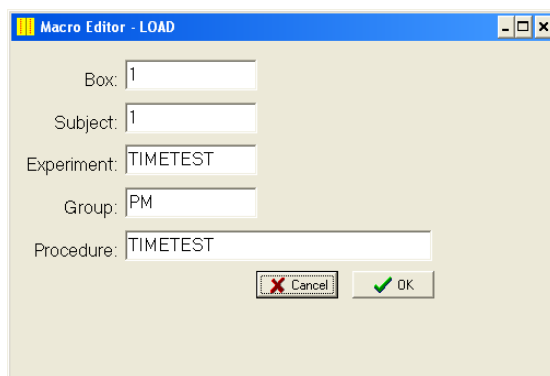
The parameters for the LOAD command may be edited using this dialog, and any changes made within the dialog will be reflected in the text of the macro after clicking the **OK** button.

To add a new command to an existing macro, position the cursor where the new command is desired and select the **Insert** menu option. The menu then presents a list of commands that may be inserted. Choosing a command then presents a dialog box (similar to the example above) to simplify creating the command.

Figure 13.2 – Insert Menu
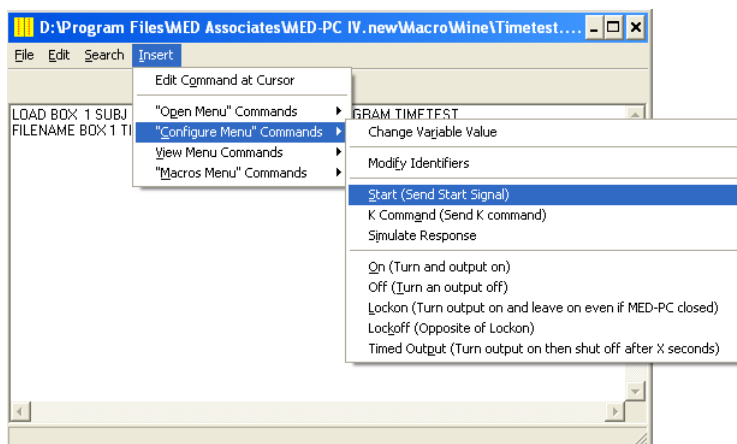
Playing Macros

Playback of an existing macro may be initiated by selecting **Macros | Play Macro**. Selecting this command displays a file dialog. An existing macro may be played by either typing in the name of the macro or by clicking on the appropriate name.

Macros need not reside in the default macro directory. It may be more convenient to store macros for various projects in distinct directories.

Getting the Most Out of Macros

- Use macros to load experiments and set parameters that do not change from session to session.

- Use the PLAYMACRO command to have one macro play another. Rather than placing the commands to load all Boxes in a single macro, record separate macros for each subject. Then create a macro to load the set of subjects -- this macro merely consists of a series of PLAYMACRO commands that call each of the individual macros. This greatly simplifies maintenance of macros and increases flexibility.

- In many experiments, the contingencies change in some systematic fashion from session-to-session. This lends itself to creating one macro to load the Boxes that the operator would always run as the first macro for setting up the session. A second macro could then be run that sets session parameters appropriate to the contingencies. For example, the first macro that is always run might be named "Squad1" and either "Left Lever" or "Right Lever" would be loaded, depending on which lever is designated active for the session. For more complex experiments, it may be desirable to create a whole series of contingency-containing macros named according to the date or phase of the experiment.

- Include reminders using the SHOWMESSAGE command.

- Rather than entering parameters via the "Configuration | Change Variables" dialog, present custom dialogs that contain meaningful queries that automatically plug the results into the appropriate macro command (such as SET). Custom dialogs that prompt for inputs may be added to macros using the INPUTBOX, NUMERICINPUTBOX, and TEXTINPUTBOX macro commands.

Tutorial 11: Creating a Macro

In this tutorial a macro will be created that will load Tutor10.mpc program into Box 1.

1. Open MED-PC and then start the macro recorder by selecting **Macros | Turn On Macro Recorder**;

2. Select **File | Open Session**;

3. Select the **Tutor10.mpc** procedure, enter **1** for the Subject and Group, **Testing Macro Recorder** for the Experiment and click the **OK** button;

4. Close the **Open Experimental Session** window by clicking the **Close** button;

5. Now select **Macro | Enter Macro Playback Delay…** menu option, enter **1000** into the text field and click **OK**. This will delay the running of the next macro command by 1 second;

6. Select **Configure | Change Variables**;

7. Select **Box 1** and the click the Named Vars button;

8. Change the "Session Time" to **5** minutes and the "Maximum Number of Rewards" to **15** and click on the **Issue** button;

9. Close the "Displaying Named Variable" window by clicking on the **Close** button;

10. Select **Configure | Signals**;

11. Select **Box 1** and the **Issue START Command** option and then click on the **Issue** button;

12. Close the "Send Signals to Boxes" window by clicking the **Close** button;

13. Select **Macros | Turn Off Macro Recorder**;

14. Enter the name "Tutor10.mac" for a file name and click the **Save** button.

The macro has been successfully created. The text of the macro can be viewed by selecting **Macro | Editor** and then opening the file that was just created. The text should look as follows:

```
LOAD BOX  1 SUBJ 1 EXPT Testing Macro Recorder GROUP 1 PROGRAM TUTOR10
DELAY 1000
SET "Session Time" VALUE        5.000 MAINBOX 1 BOXES  1
SET "Maximum Number of Rewards" VALUE       15.000 MAINBOX 1 BOXES  1
START BOXES  1
```

See Appendix B for a complete list of all available Macro commands and their syntax.

Appendix A | MedState Notation Commands

This appendix lists all MedState Notation commands and is presented in detail, with syntax, comments, examples and discussion. They have been grouped as follows, Input Section Commands, Output Section Commands, Mathematical Commands, Statistical Commands, Decision Functions, Array Functions, Data Handling Commands, Miscellaneous Commands, Special Identifiers, and Commands that Come Before the First State Set.

Input Section Commands

#START

#START is used to hold a State Set in a given State until the "START" command is issued. #START may appear in any State Set and may appear more than once. This command is useful for allowing the operator to load procedures and place subjects in chambers before initiating the experimental session as well as allowing the setting of variable values.

Syntax: P1#START:

Where: P1 = Number, constant, variable, array element, or special identifier.

Comments: P1, since it is unclear when specifying a count would be useful, it is generally not
               stated and defaults to 1

Examples and Discussion:

```
#START: ---> S2       \ Go to State 2 following #START
#START: ON 1 ---> S2  \ Turn on 1 and go to State 2 following #START
5#START: ---> S2      \ Wait for 5 #START commands before proceeding
                      \ to State 2
```
Remember, MED-PC procedures actually begin to execute as soon as they are loaded. #START does not initiate the procedure, it is merely a mechanism for holding a State Set in a given State until the operator provides keyboard input. Also, #START and program Start Dates and Times in MED-PC printouts and data files are unrelated.

#R

#R is satisfied by "Responses" resulting either from keyboard simulation or from satisfying the electrical requirement of an input on the MED Associates interface[6]. It has two parameters, P2 that specifies a logical input number, and P1 that specifies the number of responses that must occur to progress to the output section of the statement.

Syntax: P1#RP2

Where: P1 & P2 = number, constant, variable, mathematical expression, array element, or special identifier.

Comments: P1 defaults to 1 if not stated

             P2 must be a legal input in the range 1..80 defined in the MPC2INST.DTA file

Examples and Discussion:

```
5#R3: ON 1 ---> SX
```

In the preceding example, Output 1 will be turned on after five responses have occurred on Input 3. If P1 is omitted, as in "#R3: ON 1 ---> SX", P1 defaults to 1 and the first response will turn on Output 1. P1 is reset upon entry to a state. #R is unrelated to the variable "R."

A common misconception is that an external input may be detected in only one State Set at a time. In actuality, a single #R, #K, or #START: may be detected and processed in multiple State Sets, without penalty of efficiency or processing speed. In the following example, an occurrence of #R1 would place three SHOW's on the screen:

```
S.S.1,
S1,
  #R1: ADD A; SHOW 1,First,A ---> SX


S.S.2,
S1,
  #R1: ADD B; SHOW 2,Second,B ---> SX


S.S.3,
S1,
  #R1: ADD C; SHOW 3,Third,C ---> SX
```

---

[6]   A variety of modular and stand-alone interface cards are available. The most common input is a 28 VDC ground signal or a TTL 5VDC sinking logic signal.

Explicit Time Inputs: " (Seconds) and ' (Minutes)

Time, specified in terms of seconds or minutes, may be used as an input condition.

Syntax: P1" or P2'

Where: P1 & P2 = number or constant

Comments   " = Seconds

                        ' = Minutes

Legal Examples:

```
a) 1":   ON 1 ---> SX
b) 0.5": ON 1 ---> SX
c) 5.2': ON 1 ---> SX
d) 0.5': ON 1 ---> SX
e) ^Dur = 2

   S.S.1,
   S1,
     ^Dur": ---> S2
```

Illegal Example:

```
f) A": ON 1 ---> SX
```

As shown in the examples above, time may be specified as decimal quantities unless P1 is a constant. When a fractional minute is specified, (example c above), the decimal portion corresponds to fractional minutes, not seconds. When specifying time values it is most precise to use values that are multiples of the temporal resolution declared during the installation procedure (typically 10 milliseconds). Time values falling between two multiples will be treated as the higher of the multiples. For example, MED-PC handles 0.245" as 0.25". Time values are reset upon entry to a state.

A fundamental rule is that only one time expression may occur within a single state. The following is illegal and will produce a translator error message:

```
S.S.1,
S1,
  1": ON 1  ---> S2
  2': OFF 2 ---> S2
```

Variable Time Inputs: #T

Time values may also serve as inputs without an explicit declaration. This may be accomplished by using #T preceded by a variable containing a specified amount of time.

Syntax: P1#T

Where: P1 = a variable, array element, or mathematical expression.

LEGAL EXAMPLES:

Example A:

```
S.S.1,
S1,
  1": ON 1; SET X = 1" ---> S2

S2,
  X#T: OFF 1 ---> S1
```

Example B:

```
LIST X = 1", 2", 3"


S.S.1,
S1,
  1": ON 1 ---> S2

S2,
  X(0)#T: OFF 1 ---> S1
```

In Example A, X is set equal to one second and then used in State 2 as the parameter to #T. The effect of the procedure segment is to repeatedly turn output 1 on for 1" and off for 1". Example B manipulates output 1 in a similar fashion, but demonstrates the use of an array element as the parameter. A common misconception is that array variables may be set to time values only through a list declaration. The following examples, in which B(0) = 5" and B(1) = 10" are equivalent:

```
LIST B = 5", 10"

#START: SET B(0) = 5", B(1) = 10" ---> S2
```

As with explicit time values, only one time command per state may be present.

```
S.S.1,  \ Illegal Example
S1,
  X#T: ---> SX
  1": ---> SX


S.S.1,  \ Illegal Example
S1,
  X#T: ---> SX
  Y#T: ---> SX
```

A useful trick for time variables is to set the numeric value only and convert this value to internal time units only when the procedure is started. This makes it possible for the user to change the value of "A" in the following example without understanding time conversions. For example:

```
S.S.1,
S1,
  1": SET A = 5 ---> S2

S2,
  #START: SET A = A * 1" ---> S3
  1": SHOW 1,A_Value,A ---> SX

S3,
  A#T: ON 1 ---> S4

S4,
  1": OFF 1 ---> S3
```

Time Inputs Less Than the Resolution Value

The minimum amount of time that an MSN procedure can time is equal to the resolution parameter declared during installation. A system set up with 10 ms resolution can not time events less than 0.01". On a system with 10 ms resolution, the following code would increment A every 10 ms. If the resolution was 25 ms, then A would increment every 25 ms. Using time inputs less than the resolution value causes no particular stress to MED-PC run-time programs -- just be advised that they become equal to the resolution value.

```
S.S.1,
S1,
  0.001": ADD A ---> SX  \ Increments A every 10ms not every 1ms
```

A possibly less obvious situation arises when programs are run without an interface and the PC's internal clock (as opposed to the MED crystal timer) is used for timing; since the PC's timer cannot time in units less than 66 ms, this value becomes the effective resolution. If timing seems to be slow or inaccurate while debugging a procedure without an interface, test the program with an interface.

Internal Representation of Time

Time values in seconds (") or minutes (') are represented internally as the numeric value multiplied by the result of (1000 / the systems resolution value in seconds or minutes). For example, on a system with 10 ms resolution, the variable A in the following expression would be internally set to 100 by MED-PC:

```
#R1: SET A = 1" ---> SX
```

Examining A with the VARS command or in a MED-PC printout would show that A=100. A useful trick for displaying time values with SHOW commands is to divide the value by 1" (or 1', depending on the units of the time value) prior to display. For example:

```
LIST B = 1", 2", 3"


S.S.1,
S1,
  #START: RANDI A = B; SHOW 1,TIME,A/1" ---> SX
```

#Z (Z pulses as inputs)

Z-Pulses are used to communicate between State Sets and are generated in the output section of a MSN statement (See the output commands section of this Appendix). When placed in the input section of a MSN statement they perform in a manner analogous to responses (#R). #Z is unrelated to the variable "Z."

Syntax: P1#ZP2

Where: P1 & P2 =   Number, constant, variable, mathematical expression, array
                            element, or special identifier.

Comments: P1 defaults to 1 if not stated

                    P2 must be in the range 1...32

                    P1 is reset upon entry to the state.

Example:

```
S.S.1,
S1,
  #Z1: ON 1 ---> S2
```

#K

This command may be found in either the input or the output section of a statement. It may be used to communicate with procedures via the keyboard and or to allow Boxes to communicate with one another. (See the description of K-Pulses in the section of this chapter covering output commands). When placed in the input section of a MSN statement they perform in a manner analogous to responses (#R). Indeed, the syntax rules for #K are identical to those for #R. One common use for #K is in yoked experiment using the special identifier "BOX" as the P2 parameter. #K is unrelated to the variable "K."

Syntax: P1#KP2

Where: P1 & P2 =   Number, constant, variable, mathematical expression, array element, or special identifier.

Comments: P1 defaults to 1 if not stated

                      P2 must be in the range 1...100

Examples and Discussion:

Example A: Yoked Aversive Stimulus

```
S.S.1,
S1,
  #K(BOX-1): ON ^Shock ---> S2

S2,
  2": OFF ^Shock ---> S1
```

Example B: Free or Shaping Pellet

```
S.S.1,
S1,
  #START: ---> S2

S2,
  #K100 ! A#R1: ON 1 ---> S3  \ Reinforce After Every "A" Responses
                              \ or a Free Reinforcer if Operator
                              \ Issues #K100

S3,
  0.1": OFF 1; ADD C;
        IF C >= B [@End, @Continue]
            @End:  ---> STOPABORT  \ Session Ends After B Reinforces
            @Cont: ---> SX
```

Example C:

```
S.S.1,
S1,
  #R: ON ^Pellet ---> S2
  #K1: ---> STOPABORT

S2,
```

```
       0.05": OFF ^Pellet ---> S1
```

! (OR)

It is often desirable to permit several conditions to cause transfer to the same output section. For example, to allow presses on either a left lever or a right lever to produce reinforcement, one could write the following code:

```
S.S.1,
S1,
   #R1: ON 1 ---> S2
   #R2: ON 1 ---> S2

S2,
   0.1": OFF 1 ---> S1
```

A more desirable way to code this, however, is to use a logical OR, indicated by placing an exclamation point "!" between two or more input commands. For example, the following code indicates that either #R1 or #R2 is acceptable:

```
S.S.1,
S1,
   #R1 ! #R2: ON 1 ---> S2

S2,
   0.1": OFF 1 ---> S1
```

Output Section Commands

Overview

Output section commands fall between the colon and arrow of a statement. Multiple output commands may be separated by semicolons (;) while multiple parameters for the same command are separated by commas (,). An example of separating output commands is:

```
S1,
  #R1: ON 1, 5; ADD X ---> SX
```

Turning Outputs On and Off

ON

ON is used to turn outputs on. Turning on an already active output has no effect.

Syntax: ON P1

Where: P1 =   number, constant, variable, array element, or mathematical expression.

Comments: Comma separation permissible

Examples:

Activate a stimulus light (output 1) and grain feeder (output 2) for 4 seconds:

```
^Feeder = 2

S.S.1,
S1,
  #R1: ON 1, ^Feeder ---> S2

S2,
  4": OFF 1, ^Feeder ---> S1
```

OFF

OFF turns off the specified output. It is the opposite of ON.

Syntax: OFF P1

Where: P1 =   Number, constant, variable, array element, or mathematical expression.

Comments: Comma separation permissible

Example:

```
S1,
  #R1: OFF 1, 2, ^Feeder, A(K), X ---> S2
```

LOCKON and LOCKOFF

LOCKON and LOCKOFF commands supplement ON and OFF, and like ON and OFF, may be issued from the menu system or from the output section of an MSN statement, but with somewhat different effects. They are more powerful versions of ON and OFF in the sense that an output turned on by ON may be shutoff by either OFF or LOCKOFF, but an output turned on by LOCKON may only be shut off by LOCKOFF. Essentially, until a LOCKOFF is issued, OFF will not deactivate an output activated by LOCKON. In contrast, LOCKOFF will shut off an output, irrespective of whether it was activated by ON or LOCKON.

If an output has been turned on by ON and a LOCKON for the same output is issued, the output is upgraded from ON status to LOCKON status. Outputs activated by LOCKON remain on even after a Box is unloaded by STOPKILL or STOPABORT and are not even turned off when the system is shutdown and the CPU is turned off. The only way to shut off an output activated by LOCKON is by executing a LOCKOFF.

LOCKON and LOCKOFF are especially useful in conjunction with an output that must be left on even when Boxes wired to it are not running. An example of this requirement arises when chamber ventilation fans are controlled by MED-PC; subjects still need fresh air even when their Box is not running. Another application of LOCKON might be a light attached to the door of a running room. The output that drives the light could be shared by all Boxes and LOCKON'd by each Box as it loads. By using LOCKON, the light is not shutoff every time a Box terminates (because of automatic shutoff outputs activated by ON). When all sessions have finished, the user could execute a keyboard macro to LOCKOFF the relevant output to extinguish the light. For further discussion of output sharing, refer to the next section, "Overlapping Inputs and Outputs."

Syntax: LOCKON P1 or LOCKOFF P1

Where: P1 =   Number, constant, variable, array element, or mathematical expression.

Comments: Comma separation permissible

Overlapping Inputs and Outputs

It is an acceptable practice to have a given bit on an input or output card assigned to more than one logical Box. For example, all Boxes might have output one mapped to output card 780, Bit #1. This kind of output sharing is used in one of the beta testing labs, where every Box turns on output 1 upon loading. This output is mapped to port 780, Bit 1 in all Boxes and is connected to a relay that turns on all of the chamber ventilation fans.

It is also okay for a Box to attempt to turn on an output it doesn't have. For example, some operant chambers may have lever extension solenoids but some of the others do not. The chambers with solenoids are wired to six outputs, whereas the remaining Boxes each have four outputs. Never the less, a single procedure is used to run both types of chambers. Irrespective

of which type of chambers is actually being run, outputs five and six are turned on to extend or retract the levers (if present).

Input sharing is also permissible and might be used as a panic button to cause all Boxes to shut off simultaneously via a push-button wired to a single shared input. Entering a keyboard response (#R) that is not matched by a hardware input has no effect.

WARNING! Assigning a procedure to a Box that does not have a hardware input corresponding to an input that the procedure attempts to read would result in a stream of continuous responses being produced.

ALERTON

ALERTON produces a 500 Hz tone via the PC's speaker. The tone is alternately on and off for 500 ms. If multiple Boxes issue ALERTON, a single alternating tone is produced; one cannot identify the number of Boxes that have issued the ALERTON.

ALERTOFF

ALERTOFF cancels the tone. A single ALERTOFF cancels the tone until the next ALERTON. Thus, several Boxes could issue an ALERTON, but a single ALERTOFF would entirely eliminate the tone.

A common use for ALERTON will be to signal the end of a session as follows:

```
S.S.1,
S1,
  10#R1: ON 1 ---> S2

S2,
  2": OFF 1 ---> S1


S.S.2,
S1,
  30': ALERTON ---> S2

S2,
  #K1: ALERTOFF ---> SX
```

ALERTON may only be turned on from within an MSN procedure, but ALERTOFF may either be issued from within a procedure or by using a functionally equivalent command from the runtime menu system. It is also possible to produce beeps whenever all Boxes are shut off by enabling the "Tone Alert When Done" toggle within the runtime menu system. The beeps produced by this menu selection are functionally equivalent to those produced by ALERTON, and may be canceled either from the menu or by ALERTOFF.

Even if inline Pascal code is used to produce tones, it is still possible to simultaneously use the ALERTON/ALERTOFF features. Note that ALERTON is easier to use than inline Pascal-produced beeps because it is unnecessary to handle timing or other details of producing beeps. Additionally, the beeping will persist even when all Boxes are unloaded.

The duration of beeps produced by ALERTON may show considerable variation and may temporarily be suspended during some especially time-intensive menu tasks. This is especially likely to happen while writing files to disk. As soon as disk writing is finished, the beeping will resume.

Coordinating Events across State Sets

Z-Pulses

A construct known as a Z-pulse may be used to communicate among State Sets. Z-Pulses are generated in the output section of statements, but may also function as inputs to other statements (refer back to the Input command section for more details on this use of Z-Pulses). Z-Pulses may have values falling between 1 and 32. They can be an invaluable means for coordinating the action of multiple State Sets.

Syntax: ZP1

Where: P1 =   Number, constant, variable, array element, or mathematical expression
                       with value in the range 1...32.

Warning: It is the programmer's responsibility to ensure that P1 is in range 1...32. If this
                range is not observed, particularly if violated with a constant, variable array
                element, or mathematical expression, no warning message will be generated
                and unpredictable problems will result when programs are running.

Examples and Discussion:

Example A:

The following example demonstrates the use of Z-Pulses to coordinate an FR 10 on input 1, an FI 30" on input 2 and reinforcement. When either schedule is satisfied, a Z1 is generated in the relevant output section. For example, when the FR 10 is completed, a Z1 is generated. The Z1 then serves as an input to S2 or 3 of S.S.2 (depending on whether or not 30" have elapsed), driving S.S.2 to S4. Simultaneously, the Z1 also serves as an input in S2 of S.S.3, causing an output channel to turn on and transition to S3. After 2", a Z2 is generated, which then serves as an input to S3 of S.S.1 and S4 of S.S.2, driving them both back to S2.

```
S.S.1,  \ FR 10 on Input 1
S1,
  #START: ---> S2

S2,
  #Z1: ---> S3
  10#R1: Z1 ---> S3

S3,
  #Z2: ---> S2
```

```
S.S.2,  \ FI 30" on Input 2
S1,
  #START: ---> S2

S2,
  #Z1: ---> S4
  30": ---> S3

S3,
  #Z1: ---> S4
  #R2: Z1 ---> S4

S4,
  #Z2: ---> S2


S.S.3,  \ Reinforcer State Set
S1,
  #START: ---> S2

S2,
  #Z1: ON 2 ---> S3

S3,
  2": OFF 2; Z2 ---> S2
```

Z-Pulses can be tremendously useful when used wisely. When a Z-pulse is generated, it is not processed immediately. Instead, a record of all Z-Pulses generated during a pass through the State Sets of a procedure is recorded and then a second pass is immediately made through the State Sets. If more Z-Pulses are generated, then yet another pass through the State Sets occurs. Each pass, of course, requires processing time. A situation to avoid is having one Z-pulse lead to the immediate generation of another Z-pulse, then another, etc. Below is an example of poor programming utilizing Z-Pulses, which could produce a loop that the runtime system will treat as an error after 10 iterations.

Example B:

```
S.S.1,
S1,
  #START ! #Z3: Z1 ---> S2

S2,
  #Z1: ON 1; Z2 ---> S3

S3,
  #Z2: OFF 1; Z3 ---> S1
```

While it is unlikely that one will ever write a series of statements as aberrant as the set above, it is still important to avoid generating chains of Z-Pulses wherever possible. The following example is inefficient, for a #Z1 is used as input in S1 of S.S.2 to immediately generate a Z2, which is then used in S.S.3 to produce a reinforcer. Although the following code will not cause a system "lock up," it could result in some system performance falling on a slower computer.

Example C:

```
S.S.1,  \ FR 1
S1,
  #START: ---> S2

S2,
  #R1: Z1 ---> S3

S3,
  #Z3: ---> S2


S.S.2,  \ Record Data
S1,
  #Z1: ADD A; SHOW 1,Reinforcers,A; Z2 ---> SX


S.S.3,  \ Deliver Reinforcer
S1,
  #Z2: ON 1 ---> S2

S2,
  2": OFF 1; Z3 ---> S1
```

Z-pulse Depth Checking

If a very long (yet finite) sequence Z-pulses is generated, program performance may suffer greatly. Generally speaking do not use Z-Pulses to repeat other input statements (#R, #K, or #START). It is more efficient to repeat the input statement in multiple State Sets. In those instances where a substance such as a latency counter is activated with #START for the first trial and a Z-pulse in subsequent trials, the input may be OR'd or simply disregard the previous rule.

Example A:

```
S.S.1,
S1,
  #Z1 ! 1": Z1 ---> SX  \ Will Cause a MED-PC Runtime Error
```
Example B:

A limit of nine consecutive Z-Pulses is now considered acceptable. For example: the following is acceptable:

```
S.S.1,
S1,
  #Z1: Z2 ---> SX


S.S.2,
S1,
  #Z2: Z3 ---> SX


S.S.3,
S1,
  #Z3: Z4 ---> SX


S.S.4,
S1,
```

```
   #Z4: Z5 ---> SX
S.S.5,
S1,
   #Z5: Z6 ---> SX


S.S.6,
S1,
   #Z6: Z7 ---> SX


S.S.7,
S1,
   #Z7: Z8 ---> SX


S.S.8,
S1,
   #Z8: Z9 ---> SX


S.S.9,
S1,
   #Z9: ---> SX
```

Adding a tenth Z-pulse would cause an error condition. When the tenth Z-pulse in a chain is generated, the "ERROR" indicator on the runtime screen appears and flashes. Additionally, an entry is made in the journal, and the chain of Z-pulse is terminated; Z-pulse chains longer than nine are not tolerated. Once Z-pulse chains longer than two or three are used in a procedure, it is very possible that Z-Pulses are being substituted for clear procedure logic (comparable to using lots of GOTOs in a BASIC program).

Coordinating Events across Boxes

K-Pulses

The K-pulse bears considerable similarity to the Z-pulse in that both may be issued in the output sections of statements and received on the input side.   Whereas the Z-pulse is used to communicate between State Sets within the same Box, the K-pulse is used to communicate between Boxes.

An example of a situation in which K-Pulses may be useful is yoking procedures in which the behavior of one subject determines the stimuli to which another subject is exposed. In the following code example, the "CONTROL" Box is running a procedure that implements a "classic" auto shaping procedure in which a key light is illuminated following an Inter-Trial Interval (ITI) of random duration with a mean of 20".   Pecking the illuminated key produces immediate 4" access to grain. In the absence of a key peck, the key light is extinguished after 8" and grain is delivered. The bird in the "YOKED" Box receives the same pattern of stimuli and reinforcers, but

that bird has no control over the events. The following pair of procedures, one for the control Box and the other for the yoked Box, illustrates the use of K-Pulses.

```
\ Auto-shaping Procedure for the "Control" Box

^Feeder     = 1
^HouseLight = 2
^KeyLight   = 3

S.S.1,
S1,
  #START: ON ^HouseLight ---> S2

S2,      \ 20" Mean ITI.  Tell Yoked Box When ITI is Over by
         \ Sending a K1-pulse
  1": WITHPI = 500 [ON ^KeyLight; K1] ---> S3

S3,      \ Tell Yoked Box That the Feeder is on by Sending a K2-pulse
  #R1 ! 8": OFF ^KeyLight; ON ^Feeder; K2 ---> S4

S4,
  4": OFF ^Feeder ---> S2

\ Auto-shaping Procedure for the "Yoked" Box

^Feeder     = 1
^HouseLight = 2
^KeyLight   = 3

S.S.1,
S1,
  #START: ON ^HouseLight ---> S2

S2,      \ K1-pulse Sent by Control Box When KeyLight Turned on
  #K1: ON ^KeyLight ---> S3

S3,      \ K2-pulse Sent by Control Box When RF Starts
  #K2: OFF ^KeyLight; ON ^Feeder ---> S4

S4,
  4": OFF ^Feeder ---> S2
```

K-Pulse Theory Of Operation And Technical Details

When a K-pulse is issued, it is not immediately available as an input to statements, and does not become available to other statements and Boxes until the beginning of the next interrupt (the next time Boxes are serviced).

For illustrative purposes assume that the system resolution is set to 10 ms and the control procedure is running in Box 1 and the yoked procedure is running in Box 2. When the control Box issues K1, the K1 is entered into a queue (a waiting list). Although Box 2 will be serviced immediately after Box 1 (within a millisecond), the K1 will not be presented to Box 2. Instead, when the next interrupt occurs, about 10 ms later, the K1 is removed from the queue and made available to both Boxes 1 and 2 and Box 2 will then react to the K1. Stated succinctly, K-Pulses are placed in a queue until the next processing sweep (interrupt) occurs.

If more than one Box issues the same K-pulse within the same processing sweep, only one K-pulse will be issued. For example, if Box 1 is counting the number of K1 pulses that occur and Boxes 2 and 3 simultaneously issue K1, Box 1 will detect only 1 K-pulse. Similarly, if the same K-pulse is issued several times within the same output statement, the net effect will be the same as if the K-pulse was issued only once. For example, 1":K1; K1 ---> SX is equivalent to 1": K1 ---> SX.

When a Box issues a K-pulse, it is available to all active Boxes on the next processing sweep. Also, a Box may react to one of its own K-Pulses. The following procedure could both issue and count the occurrence of K1.

```
S.S.1,
S1,
  1": K1 ---> SX


S.S.2,
S1,
  #K1: ADD A; SHOW 1,K1 Count,A ---> SX
```

K-Pulses should not be used in place of Z-Pulses; although superficially similar, these commands have different purposes. K-Pulses are used for communication between Boxes, whereas Z-Pulses are designed for communication between State Sets within Boxes. K-Pulses have the same priority level as normal MSN commands. Unlike Z-Pulses, K commands are treated as normal inputs in the sense that when K's are "stacked" with other commands, the topmost statement will be processed in the event of a tie. For example, in the following code, in the event of a simultaneous #K1 and #R1, transition would be to S2:

```
#K1: ---> S2
#R1: ---> S3
```

Processing Efficiency of the K-pulse

As noted above, do not treat K-Pulses as interchangeable with Z-Pulses. One reason for this concern is that issuing a K-pulse from within an MSN procedure generates a considerable amount of overhead because each Box must be processed to determine whether the occurrence of the K-pulse necessitates any transitions between states. When a K-Pulse is issued from the keyboard, it is not presented simultaneously to all Boxes. Instead, the operator specifies a list of one or more Boxes to receive the K-Pulse. Furthermore, MED-PC automatically spaces the delivery of keyboard K-Pulses across Boxes to help distribute the processing load. This is in contrast to the situation when K-Pulses are issued from an MSN procedure, in which case the K-Pulse is issued simultaneously to all Boxes; hence, issue K-Pulses from within MSN procedures sparingly. This is not to discourage their use, but rather to point out that they should not be used with abandon.

The Special Variable "BOX"

It is often desirable to have an MSN procedure that reacts to K-Pulses conditionally upon which Box issued the K-pulse. Consider the yoked auto-shaping paradigm presented above. In that example, the control Box issues K1 and K2 to indicate to the yoked Box the occurrence of critical events. As long as only one Box is running the control procedure and only one is running the yoked procedure at any given time, everything will work fine. Consider, though what would happen if Boxes 1 and 3 were simultaneously running the control procedure while Boxes 2 and 4 were running the yoked procedure. Whenever a stimulus change would occur in Box 1 or 3, stimulus changes would occur in both Boxes 2 and 4; the two yoked Boxes would each be yoked to two control Boxes. One way around this problem would be to write a separate control procedure for Box 1 that issues K1 and K2 and a different control procedure for Box 3 that issues K3 and K4. This solution would work, but would require writing and maintaining multiple copies of essentially the same procedure.

A more elegant solution to this problem is to adjust the number of the K-pulse on the basis of which Box is issuing or receiving the K-pulse. In the following example, the yoked auto shaping code has been modified so that Box 1 will issue K1 and K2, whereas Box 3 will issue K3 and K4 to communicate with Box 4. This is accomplished by incorporating a special variable named "BOX" into commands that receive and issue K-Pulses. BOX is always equal to the Box number of the Box in which the MSN program is running. In the control procedure in the following code example, the K-pulse in State 2 would be K1 when the procedure is running in Box 1 because BOX would equal 1. Similarly, when running in Box 3, the same statement would issue a K3. In the yoked procedure, K(BOX - 1) would respond to K1 (issued by Box 1) when running in Box 2 because BOX would equal 2. When running in Box 4, the yoked procedure would respond to K3 (Box 3's first K-pulse). By alternating between Control and Yoked Boxes any number could run the same procedure.

```
\ Modified Auto-shaping Code Demonstrating the use of the "BOX"
\ Variable
\
\ Auto-shaping Procedure for the "Control" Box

^Feeder     = 1
^HouseLight = 2
^KeyLight   = 3


S.S.1,
S1,
  #START: ON ^HouseLight ---> S2

S2,     \ 20" Mean ITI.  Tell Yoked Box When ITI is over by
        \ Sending a K-pulse Equal to this Box's BOX number
  1": WITHPI = 500 [ON ^KeyLight; K(BOX)] ---> S3

S3,     \ Tell Yoked Box that the Feeder is on by Sending a
        \ K-pulse Equal to this Box's BOX number plus 1
  #R1 ! 8": OFF ^KeyLight; ON ^Feeder; K(BOX+1) ---> S4

S4,
  4": OFF ^Feeder ---> S2

================================================

\ Auto-shaping Procedure for the "Yoked" Box

^Feeder     = 1
^HouseLight = 2
^KeyLight   = 3


S.S.1,
S1,
  #START: ON ^HouseLight ---> S2

S2,     \ K-pulse Sent by Control Box when KeyLight Turned On
        \ Look for the First K-pulse of the Preceding Box
  #K(BOX-1): ON ^KeyLight ---> S3

S3,     \ K-pulse Sent by Control Box when RF Starts
        \ Look for the Second K-pulse of the Preceding Box
  #K(BOX): OFF ^KeyLight; ON ^Feeder ---> S4

S4,
  4": OFF ^Feeder ---> S2
```

Mathematical Commands

ADD

ADD is generally used to increment a variable or array element by 1. It performs the same function as SET A = A + 1.

Syntax: ADD P1

Where: P1 = variable or array element

Comments: Stringing Variables with Comma separation permissible

Examples:

```
S1,
  1": ADD C ---> SX                  \ Every Second Add 1 to the Value
                                     \ of C
  #R2: ADD X, Y, Z, A(I) ---> SX  \ Every Response From Input 2 Add 1
                                     \ to the Value of Variables X, Y, Z
                                     \ and Array Element A(I)
```

SUB

SUB is used to decrement a variable or array element by 1. It performs the same function as SET A = A - 1.

Syntax: SUB P1

Where: P1 = variable or array element

Comments: Stringing Variables with Comma separation permissible

Examples:

```
S1,
  1": SUB C ---> SX                  \ Every Second Subtract 1 from the
                                     \ Value of C
  #R2: SUB X, Y, Z, A(I) ---> SX  \ Every Response From Input 2
                                     \ Subtract 1 from the Value of
                                     \ Variables X, Y, Z and Array
                                     \ Element A(I)
```

LIMIT - Increment or Decrement to a Bound

This function may be used to increment or decrement a variable. A limit is specified beyond which the variable will not be incremented or decrement. This function should be used when it is specifically desired to limit the value of a variable; it should not be used as an alternative to ADD, SUB or SET in cases where those functions will suffice. The processing overhead or performance penalty associated with LIMIT is somewhat higher than that associated with the alternative functions. This is not to say that LIMIT should not be used when necessary, but rather that it should not be used indiscriminately.

Syntax: LIMIT P1,P2,P3

Where: P1 =   Variable or array element to be incremented or decremented. It may not be a constant or a fixed number.

                P2 =   A variable, array element, constant or number specifying the numerical value by which P1 will be incremented or decremented each time LIMIT is executed.

                P3 =   The maximum or minimum value of P1. Once reached this value will be held no matter how many times LIMIT is executed.

```
S.S.1,  \ This code Fragment Adds 2 to X Every Second.
S1,     \ X will Achieve and Hold a Value of 10.
  1": LIMIT X,2,10 ---> SX


S.S.1,  \ This Code Fragment Adds 3 to X Every Second.
S1,     \ X will Achieve and Hold a Value of 9.
  1": LIMIT X,3,10 ---> SX


S.S.1,  \ This Code Fragment Subtracts 1 from X Every Second.
S1,     \ X will Achieve and Hold a Value of -5.
  1": LIMIT X,-1,-5 ---> SX
```

SET

SET is used to perform any of four basic mathematical operations involving two or more operands. Any mathematical function provided by Pascal (Delphi) can also be inserted within a MSN statement using In-Line Pascal (See Appendix C). Two forms of this command are possible as indicated by syntax A and syntax B.

Four "Operators" are permitted:

/	(Division)
*	(Multiplication)
+	(Addition)
-	(Subtraction)

Syntax A: SET P1 = P2 Operator P3

Syntax B: SET P1 = P2

Where: P1 = variable or array element

                 P2 and P3 = number, variable, or array element

                 Operator = /, *, +, or -

Comments: Stringing is permissible

                 P2 and/or P3 may be followed by " or ' to assign a time value to a variable or array element.

Assigning a new value to a constant is not permissible. It will not produce a translator error but will produce a Delphi compiler error during compilation, typically of the form "Undefined Label."

In the original MED-PC, complicated math expressions had to be broken into pieces. For example "SET A = 1 + 2 * 10 / 4 - 3" may have been written, "SET A = 2 * 10, A = A / 4, A = A - 2". Since Version 2.0 of MED-PC (DOS), complex expressions can be written directly; this example is now written as "SET A = 1 + 2 * 10 / 4 - 3".

Examples:

```
1':  SET A = 5 * A, C = B(K) ---> SX
#R3: SET A = 5 * A + B + C ---> SX  \ Note: Multiple Operations
```

BIN

BIN is an output command that can be used to generate frequency distribution as data is collected. The frequency distribution is output into a range of array elements specified in the call to BIN. The width of each "Bin" in the frequency distribution is user controllable. The first "Bin" always contains the total frequency, or total number of all categorized events, and the second "Bin" always contains the total number of events with values greater than that represented by the last "Bin." Like any array, the BIN data array must be dimensioned prior to State Set 1.

Syntax: BIN P1,P2,P3,P4,P5,P6

Where: P1 = Array which will hold the frequency distribution.

                P2 =   A variable or array element containing the number to be added to the frequency distribution.

                P3 =   The units of P2. If one is recording time, then P3 is how frequently P2 is incremented.

                P4 =   The width of each bin or cell of the distribution.

                P5 =   Array element, variable, constant, or number denoting the first counter or array element containing the BIN distribution. It is also the element into which the total frequency will be recorded.

                P6 =   Array element, variable, constant, or number denoting the last counter or array element into which the BIN distribution is recorded.

Example:

```
^Start = 0
^End   = 10

DIM C = 10


S.S.1,
S1,
  #R1: BIN C,A,0.1,5,^Start,^End; SET A = 0 ---> S1
  0.1": ADD A ---> SX
```

In this example, the frequency distribution is recorded into array C from element C(0) through element C(10). C(^Start) marks the first element of the distribution and will also contain the total frequency recorded, i.e., C(^Start+1) + C(^Start+2) + ... + C(^End). A is a variable containing the current value to be categorized. Values greater than the category assigned to C(^End) are placed in C(^Start+1), the second element in the BIN array. The 0.1 indicates that the data is being recorded with a resolution of 0.1" and the 5 indicates that each BIN is to be five seconds wide. Given the values above and a subject that responds 50 times with IRT's between 0.1 seconds and some indeterminate maximum, the following data array might result:

```
ELEMENT     BIN RANGE IN SECONDS  RESPONSES

C(0)        Total Responses        50
C(1)        Greater than 45        3

C(2)         0.0 - 5               3
C(3)         5.1 - 10              5
C(4)        10.1 - 15             7
C(5)        15.1 - 20             8
C(6)        20.1 - 25             9
C(7)        25.1 - 30             6
C(8)        30.1 - 35             4
C(9)        35.1 - 40             3
C(10)       40.1 - 45             2
```

The aforementioned example serves to "BIN" inter-response times. Sometimes it is desirable to "BIN" a response count by the elapsed time. The following example illustrates this with an FI20 schedule. Note: This does not require the BIN command.

```
\ Sample Program Showing how to Create Incremental "Bins."

\ C = Array Storing the Number of Responses in 5 Minute Bins
\ D = Experiment Duration with a Default Value of 60 Minutes
\ I = Index into the Array


DIM C = 200  \ Arbitrary value.  In this example only 13 elements
             \ (0 - 12) are used, one for total response count plus
             \ 12 five minute bins.


S.S.1,  \ Increment Response Count in C(0) Plus Bin C(I)
S1,
  0.01": SET D = 60, I = 1 ---> S2

S2,
  #START: SHOW 1,Bin,I, 2,Bin Count,C(I), 3,Tot Count,C(0) ---> S3
  1": SHOW 1,Bin,I, 2,Session,D ---> SX

S3,     \ Response Count and Display
  #R1: ADD C(0), C(I); SHOW 2,Bin Count,C(I), 3,Tot Count,C(0) ---> SX


S.S.2,  \ Increment to the Next Counter Every 5 Minutes.  Use a
        \ Variable and #T for a more Flexible Program.  See S.S.3.
S1,
  #START: SET C(I+1) = -987.987 ---> S2

S2,
  5': ADD I; SET C(I) = 0, C(I+1) = -987.987;
      SHOW 1,Bin,I, 2,Bin Count,C(I) ---> SX


S.S.3,  \ End the Session After D Amount of Time
S1,
  #START: SET D = D * 1' ---> S2

S2,
  D#T: ---> STOPABORTFLUSH
```

Statistical Commands

MSN contains a number of built in statistical commands that compute summary statistics on array elements. Either an entire array may be passed to the command or a range of array elements may be specified. In most instances, passing a range of array elements will be preferable if the amount of data cannot be predicted in advance.

Each command follows the same syntax:

Syntax: Command P1 = P2,P3,P4

Where: Command is the name of the statistical command.

P1 =   An array element or variable that will receive the result of the calculation.

P2 =   Array containing the source data. For GeometricMean and HarmonicMean, the array must not contain any zeros.

P3 =   Index of first array element to include in calculation. This will often be zero, but may be any value less than or equal to the maximum array element.

P4 =   Index of the last array element to include in the calculation. This will often be the last array element into which data have been recorded.

Statistical commands:

ARITHMETICMEAN: Computes the Arithmetic Mean of the array segment.

HARMONICMEAN: Computes the Harmonic Mean of the array segment. There may not be any zeros in the array segment.

GEOMETRICMEAN: Computes the Geometric Mean of the array segment. There may not be any zeros in the array segment.

MAXARRAY: Returns the Largest value in the array segment.

MINARRAY: Returns the Smallest value in the array segment.

MAXARRAYINDEX: Returns the Index (subscript) of the largest value in the array segment. The Index is relative to the entire array. In other words, if the first element of a segment has the largest value and that element is element 5 of the overall array, a 5 will be returned, rather than element 0.

MINARRAYINDEX: Returns the Index of the smallest value in the array. See MAXARRAYINDEX regarding indexing.

POPULATIONVARIANCE: Returns the Population Variances of the array segment.

SAMPLEVARIANCE: Returns the sample variance of the array segment.

SUMARRAY: Returns the Sum of the elements in the array segment.

SUMSQUAREARRAY: Returns the Sum of the squares of the elements in the array segment.

These commands should be used sparingly. It is a reasonable use of processor time to use these commands on an occasional basis on a reasonably sized array segment. There are no hard and fast rules, but trying to computing the variance of an array of 100,000 elements every 10 msec would likely result in an inability for the system to keep pace. On the other hand, computing this statistic every minute on 50 elements is probably reasonable.

As noted above, it is likely that a subset of an array will be passed, rather than an entire array. The reason for this is that arrays often contain several different types of data. In addition, many experiments collect a variable amount of data. For example, an experiment that ends after a fixed amount of time may include a variable number of reinforcers, in which case the number of latencies to respond to following each reinforcer would vary. In this case it would be necessary to track the last array element into which data were recorded and pass this to the statistics command.

Example:

```
^RF        = 1
^StartRatio = 2
^Pellet    = 1


DIM C = 200


S.S.1,  \ FR 5
S1,
  #START: ---> S2

S2,
  5#R1: ON ^Pellet; Z^RF;
       HARMONICMEAN D = C,0,I; ADD I; SHOW 1,Harmonic Mean,D ---> S3

S3,
  0.1": OFF ^Pellet; Z^StartRatio ---> S2


S.S.2,  \ Time Each Ratio
S1,
  #START: ---> S2

S2,
  #Z^RF: ---> S3
  0.1": SET C(I) = C(I) + 0.1 ---> SX

S3,
  #Z^StartRatio: ---> S2
```

This FR 5 program displays the Harmonic Mean of the duration (in seconds) of ratio completions. Notice that a variable, I, is used in the call to HARMONICMEAN to indicate the last element in the C array that should be included in the calculation. Variable I tracks the number of completed ratios and is incremented after the call to HarmonicMean.

Decision Functions

IF

IF permits the values of two numeric parameters, a numeric parameter and a variable, or two variables to be compared. Several syntaxes are permissible.

Six comparisons "Operators" are permitted:

=	(Equal To)
<	(Less Than)
<=	(Less Than or Equal To)
>	(Greater Than)
>=	(Greater Than or Equal To)
<>	(Not Equal To)

Syntax A: IF P1 Operator P2 [@Label1, @Label2]

@Label1: Output Section ---> Transition

@Label2: Output Section ---> Transition

Syntax B: IF P1 Operator P2 [@Label1]

@Label1: Output Section ---> Transition

Syntax C: IF P1 Operator P2 [Output Section] ---> Transition

Where: P1 & P2 =   Constant, number, variable, array element, or special identifier as described below.

Label1 & Label2 = Any text label; must be preceded by @.

Output Section = Any legal output command(s).

Transition =   Any legal Transition such as SX, STOPABORT, STOPKILL, or S1...S32 (given that S1...S32 is a valid State within the same State Set)

Comments: Unlimited nesting is permissible. @Label1 is the true condition and @Label2 is the false condition.   In syntax B and C a transition to SX with no output command occurs when the test condition is not met (i.e., when the test condition is false). The three syntax variations may be freely intermixed when nesting. Labels are arbitrary, and need not match, but must begin with @.

Examples and Discussion

Example A:

Because labels are arbitrary, spelling does not need to be consistent. If the comparison evaluates as TRUE, then the following statement (on the next line) is executed.   If the comparison is FALSE, then the statement two lines down is executed.

```
S.S.1,
S1,
  #R1: ADD A; OFF 1; ON 2 ---> S2

S2,
  2": OFF 2; IF A = 100 [@True, @False]
              @True: ---> STOPABORT
              @False: ON 1 ---> S1


S.S.1,
S1,
  #R1: ADD A; OFF 1; ON 2 ---> S2

S2,
  2": OFF 2; IF A = 100 [@True, @False]
              @True: ---> STOPABORT
              @F: ON 1 ---> S1
              \ @False and @F Are Not Required to Match.
```

The legal examples above illustrate a situation in which a #R1 increments a reinforcement counter, turns off a stimulus light, turns on a feeder and goes to S2. After 2", the feeder output is turned off and variable A (the reinforcement counter) is tested. If the value of A is now 100 then the statement associated with the label @True is executed and the procedure terminates. If the value is less than 100, the statement following the label @False is executed (the stimulus is turned ON and transition takes place back to S1).

Example B:

When a label is in the input section of a statement, a colon (:) must follow it, even when there is no output section to the statement.

Illegal Examples:

```
S.S.1,
S1,
  #R1: ADD A; OFF 1; ON 2 ---> S2

S2,
  2": OFF 2; IF A = 100 [@T, @F]
              @T ---> STOPABORT  \ Missing ":" After Label
              @F: ON 1 ---> S1
```

Example C:

Careful attention to proper syntax is important; for certain errors are not detected by the translator and will result in Pascal errors. An error which the translator does not detect but which will cause programs to malfunction is to have a transition arrow after labels. For example:

```
1": IF A [@True, @False] ---> S2
```

Example D:

Although not permitted by earlier versions of MED-PC, IF statements may be nested more than one deep. In the following example variables A, B and C are all tested with respect to Variable X. If all three are greater than or equal to X then transition is to State 2. If any variable fails the test then transition is to State 3.

```
S.S.1,
S1,
  1": IF A >= X [@1stTrue, @1stFalse]
         @1stTrue: IF B >= X [@True, @False]
                      @True: IF C >= X [@True, @False]
                                @True: ---> S2
                                @False: ---> S3
                      @False: ---> S3
         @1stFalse: ---> S3
```

The labels used with the IF command are arbitrary. For example, in the above example if A >= X is false then execution immediately drops down to the statement on the same tab, regardless of whether @1stFalse is properly spelled. It is, however, mandatory to use a label. Also, a colon must follow the label, even if there is no output section, as in "@False: ---> S3."

Example E:

A variation on the IF command is to specify only the true alternative. If the conditions tested by the IF are not met, then transition to SX automatically occurs without being stated. This is illustrated with Example E.

```
S.S.1,
S1,
  10": ADD A; IF A = 10 [@Go]
                 @Go: ON 1 ---> S2

S2,
  0.1": SET A = 0; OFF 1 ---> S1
```

In Example E, no transition will take place in State 1 until A is equal to 10. When A = 10 is true, transition to State 2 will occur.

Example F:

A final variation on the syntax permits labels for true and false conditions to be omitted. When this syntax is used, output commands are enclosed in square brackets following the logical comparison. If the logical comparison is TRUE then the commands enclosed in the brackets are executed and the transition indicated to the right of the arrows is executed. If the logical comparison is FALSE then transition to SX automatically occurs. This is illustrated by the following:

```
S.S.1,
S1,
  10": ADD A; IF A = 10 [ON 1] ---> S2

S2,
  0.1": SET A = 0; OFF 1 ---> S1
```

Example F functions equivalently to Example E. This format requires a pair of square brackets, even if they enclose nothing. For example:

```
S.S.1,
S1,
  10": ADD A; IF A = 10 [] ---> S2
```

Compound IF Statements

IF statements may be constructed such that several logical conditions must be met in order for the expression to evaluate as TRUE. This may be accomplished by placing each set of logical criteria in parentheses and connecting each set with AND, OR, NOT, AND NOT, or OR NOT. Parentheses serve to denote the order in which expressions are evaluated, in much the same way that parentheses control execution of algebraic expressions in SET statements. Logical expressions are always evaluated first within the deepest level of parentheses.

Examples:

In the following expression, output 1 is turned on only if A equals 1 and B equals 2.

```
#R1: IF (A = 1) AND (B = 2) [ON 1] ---> S2
    \ Note that each term "A = 1" and "B = 2"
    \ are enclosed in parentheses ().
```

In the following expression, output 1 will be turned on if either X + 3 equals 10, or if A equals 1 and B equals 2. Of course, if all three conditions are met, output 1 will also be turned on.

```
#R1: IF (X + 3 = 10) OR ((A = 1) AND (B = 2)) [ON 1] ---> S2
```

This example also demonstrates the use of mathematical expressions within IF statements. Whenever writing IF statements in which parenthetical expressions are used, be sure that the number of left and right parentheses are equal. Parentheses must be used whenever more than one logical condition is being tested.

The following are examples of illegal tests, followed by correct examples:

1)

```
  A = 1 OR B = 2      \ Illegal -- no parentheses
  (A = 1) OR (B = 2)  \ Legal
```

2)

```
  A = 1) OR (B = 2    \ Illegal - unequal number of parentheses
  (A = 1) OR (B = 2)  \ Legal
```

3)

```
  A = 10 OR 5         \ Illegal - each comparison must have 2 terms
  (A = 10) OR (A = 5) \ Legal
```

Special Identifiers

P1 and/or P2 may be a special identifier as listed below. One special identifier reflects the specified State Set's current state, and is represented by "S.S.#." Never alter the value of this variable. Other special identifiers may be used in expressions in the same way that any variable or array element may be used. For example, they may participate in logical tests with IF statements, be part of assignments with SET, and may serve as prefixes or suffixes to #K, #R, #Z, etc. MED-PC does not prevent one from changing the value of these identifiers with a SET command, but do so only with caution. These variables are also available for use in inline PASCAL expressions.

Examples and Discussion

Example A:

```
  1": IF S.S.3 = 5 [ON 1] ---> S2
```

The IF command makes a logical comparison on the basis of whether State Set 3 is currently in State 5. If the comparison is true output 1 is turned ON and a transition is made to State 2. NOTE: A complete list of the variables that may be queried can be found in the "Special Identifiers" section.

WITHPI

WITHPI is a probability gate that samples with replacement. WITHPI functions act a lot like an IF statement, but truth or falsity of decisions depend upon probabilistic decisions. As with IF, it may be used with three different syntaxes. Probabilities are always specified as chances out of ten thousand.

Syntax A: WITHPI = P1 [@Label1, @Label2]

                           @Label1: Output Section ---> Transition

                           @Label2: Output Section ---> Transition

Syntax B: WITHPI = P1 [@Label1]

                           @Label1: Output Section ---> Transition

Syntax C: WITHPI = P1 [Output Section] ---> Transition

Where: P1 =   Constant, number, variable, or array element.

                   Label1 & Label2 = Any text label containing only letters.

                   Output Section = Any legal output command(s).

                   Transition =   Any legal Transition such as SX, STOPABORT, STOPKILL, or S1...S32 (given that S1...S32 is a valid state within the same State Set).

Comments: Nesting is permissible using the first Syntax.

                   Stringing is not permitted.

                   Labels are arbitrary but must begin with @.

                   When a label is in the input section of a statement, it must be followed by ":" even when there is no output section to the statement.

Examples and Discussion

Example A:

```
S.S.1,
S1,
  #R1: WITHPI = 5000 [@Reinforcement, @NoReinforcement]
          @RF: ON 1 ---> S2
          @NoRF: ---> SX

S2,
  2": OFF 1 ---> S1
```

Because labels are arbitrary, spelling does not need to be consistent. If the probability gate evaluates as TRUE, then the following statement (on the next line) is executed. If the probability gate is FALSE, then the statement two lines down is executed.

In Example A, every response in State 1 has a pseudo-randomly determined probability of 5000/10000 (50%) of causing transition to the true alternative (@RF) in which case reinforcement is delivered and transition to State 2 will occur.

Changing the parameter from 5000 to 1000 would specify that response 1 would have a 10% (1000/10000) probability of resulting in transition to State 2.

Example B:

An error which TRANS does not detect but which will cause programs to malfunction is to have a transition arrow after labels. For example:

```
1": WITHPI = 1000 [@True, @False] ---> S2
```

Array Functions

Data that has been entered into arrays via a LIST declaration may be accessed via the commands LIST, RANDD, and RANDI. LIST successively draws values from an array, RANDI randomly selects with replacement from an array, while RANDD selects without replacement from an array.

LIST

List is first placed before State Set 1 to dimension an array and assign a value to each element in an array. It can then be used in the output section of a statement to select each value in sequence. When the last element in the list has been used, selection restarts at the beginning of the list.

Syntax A: Defining the array

                    LIST P1 = P2, P3, ..., Pn

Where: P1 = The name of the array to be declared (A...Z)

                    P2, P3, ..., Pn = Number

Comments: Each line must end on a comma <CR>.

Syntax B: Selecting elements from the array.

                    LIST P1 = P2(P3)

Where: P1 = Variable or array element

                    P2 = Array from which an item is to be drawn

                    P3 = Variable or array element used as subscript to array P2

Comments: Stringing is permissible

                    The value of P3 is automatically incremented by the LIST command. Assignments to P3 through other commands will affect the selection of subsequent items via the LIST command (i.e. it is possible to skip or retrace elements in the array via manipulation of the P3 subscript.) This must be done

with caution, however, as an illegal subscript value will be ignored by MED-PC, reverting the array back to subscript zero.

Array P2 may be declared with LIST or DIM statement.

Examples and Discussion

Running the following procedure would result in the following pattern of outputs: 1, 2, 3, 1, 2, 3, 1, 2, 3, …

Example A:

```
LIST A = 1, 2, 3


S.S.1,
S1,
  1": LIST B = A(K); ON B ---> S2

S2,
  1": OFF B ---> S1
```

In the first LIST statement above, an array named A is declared. The lowest element of an array is always referenced as Element 0. Element 0 contains the value 1, Element 1 contains the value 2 and Element 2 contains the value 3.

One second after this procedure is loaded, the statement "LIST B = A(K)" sets B equal to the value of element K of array A. Since all variables are automatically set to 0 at the beginning of program execution, the value of B is set equal to A(0) and ON B causes output 1 to be turned ON. One second later, output 1 is turned OFF in State 2. Following assignment of the value of A(K) to B, 1 is automatically added to the value of K so that the next time the list statement is executed, the array index (K) for array A is equal to 1 giving B a value of 2 to turn ON Output 2.

The LIST command continues to select successive array elements until the end of the list is reached. When the last element has been accessed, the array index (K) is reset back to 0.

Example B:

```
LIST A:  5, 10, 15, 20, 25, 30,  \ Note that each line
        30, 35, 40, 45, 50, 55,  \ must end in a comma
        60, 65, 70, 75, 80, 85   \ except for the last one.
```

RANDD

RANDD is similar to LIST except that it selects values from a list without replacement. A major difference, however, is that no subscript is specified with this command. Again, all elements in the array are drawn before any element is repeated. RANDD can only operate on arrays declared in a LIST statement.

Syntax: RANDD P1 = P2

Where: P1 = Variable or array element

                P2 = Array from which an item is to be drawn

Comments: Stringing of parameters is permissible.

                Array P2 must be declared with a LIST statement.

                The maximum number of elements that may be in an array manipulated by RANDD is 501 (elements 0...500).

                The order of the contents of P2 is not affected by RANDD. Selection actually takes place from an internal copy created and managed automatically by MED-PC.

Example:

Items are randomly selected from the LIST, however once selected an item is removed from the list until all items have been selected. This is selection without replacement. The following State Set might turn outputs on in the following order: 1, 2, 3, 2, 3, 1, 2, 1, 3, 3, 1, 2, ...

```
LIST A = 1, 2, 3


S.S.1,
S1,
  1": RANDD B = A; ON B ---> S2

S2,
  1": OFF B ---> S1
```

RANDI

RANDI is similar to RANDD in that it selects values from a list. Again, no array subscript is specified. RANDI randomly draws with replacement from an array, however, so elements can be repeated and may not occur an equal number of times (unless run over a long period of time). RANDI can only operate on arrays declared in a LIST statement.

Syntax: RANDI P1 = P2

Where: P1 = Variable or array element

                P2 = Array from which an item is to be drawn

Comments: Stringing of parameters is permissible.

Array P2 must be declared with a LIST statement.

The maximum number of elements that may be in an array manipulated by RANDI is 501 (elements 0...500)

The order of the contents of P2 is not affected by RANDI Selection actually takes place from an internal copy created and managed automatically by MED-PC.

Example:

The actual order of selection is totally random. A single item may be selected more than once even though some items have not been selected. This is selection with replacement. The following order of outputs might occur: 3, 1, 1, 2, 1, 3, 2, 2, 1, 3, ...

```
LIST A = 1, 2, 3


S.S.1,
S1,
  1": RANDI B = A; ON B ---> S2

S2,
  1": OFF B ---> S1
```

COPYARRAY

COPYARRAY is an output command that may be used to transfer the contents of one array to another. This command simplifies and speeds the transfer of data between arrays and can be used to move an entire array to a "buffer array" for saving while continuing to collect data. Copying the contents of one array to another is particularly useful for continuous running applications.

COPYARRAY takes three arguments: the source array from which data are to be copied, the target array to which data will be copied, and the number of elements of the source to copy to the target. Copying always starts with the first element of the source array (element 0) and data are always placed into the target array starting at element 0 of the target. The number of elements of the source array that are copied should never exceed the size of the target array. If an attempt is made to copy too many elements to the target array, a MED-PC runtime error message will be generated.

Syntax: COPYARRAY P1,P2,P3

Where: P1 = The source array from which data will be copied.

                P2 = The array into which data will be copied.

                P3 = The number of elements of P1 to copy to P2.

                P1 & P2 must be the name of an array.

                P3 =   Variable, constant, number, array element, or mathematical expression.

Example:

In the following example, the current cumulative response totals on inputs 1-3 are transferred from array A to array B every 5' and then printed. This technique ensures that all data are printed from the same moment (see description of PRINT command).

```
DIM A = 2
DIM B = 2


PRINTVARS = B


S.S.1,
S1,
  #R1: ADD A(0) ---> SX
  #R2: ADD A(1) ---> SX
  #R3: ADD A(2) ---> SX


S.S.2,
S1,
  5': COPYARRAY A,B,3; PRINT ---> SX  \ From A to B Copy 3 Elements
                                      \ (0, 1, 2) and then Print.
```

ZEROARRAY

This command sets all of the elements of an array to 0. The command takes the name of the array to zero as its only argument.

Syntax: ZEROARRAY P1

Where: P1 =   Must be the name of an array declared by LIST or DIM.

Example:

```
S.S.2,
S1,
  5': COPYARRAY A,B,3; ZEROARRAY A; PRINT ---> SX
```

The above uses the same S.S.2 from the COPYARRAY example; however, this time the counter in array A is also zeroed every 5'.

INITCONSTPROBARR

The INITCONSTPROBARR command is used to initialize the values of an array to the progression defined by the Hoffman-Fleschler constant probability distribution. This distribution is frequently used to determine the intervals comprising variable-interval schedules.

Syntax: INITCONSTPROBARR P1,P2

Where: P1 = An array defined by the List directive

                  P2 = The mean value of the array

Comments: INITCONSTPROBARR only works with arrays defined with the LIST command. The array must not be defined with a DIM command.

                  The particular values listed in the definition of the array are irrelevant; they will be over-written when this command executes.

                  If using the resulting array elements in conjunction with the #T operator to measure time, be sure to multiply each array element by 1" or 1' to scale the values to units of time. Although the resulting time values may not be even increments of the system resolution (e.g., a value of 0.751 would be represented on a 10 millisecond system as 75.1 after multiplication by 1"), this is not a concern because the system will automatically round the value up to the next multiple of the resolution (e.g., 75.1 would be automatically rounded up to 76, which would introduce only a very slight deviation from the theoretically-desired time value). If this is a concern, inline Pascal can be used to round the time value up or down according to standard rounding rules. This is illustrated in the second example below.

Basic Example:

```
LIST V = 1, 2, 3, 4, 5, 6, 7

S.S.1,
S1,
  1": INITCONSTPROBARR V,10 ---> S2
```

The code above initializes elements 0 through 6 of list V to values of 0.751, 2.425, 4.439, 6.966, 10.364, 15.596, and 29.459, respectively.

Example of Rounding the Value of an Element:

```
LIST V = 1, 2, 3, 4, 5, 6, 7

S.S.1,
S1,
  1": INITCONSTPROBARR V,10 ---> S2

S2,
  1": RANDD A = V;
      SET A = A * 1";
      ~A := Round(A);~ ---> SX  \ Eliminate the Fraction of a "Tick"
```

GETVAL

GETVAL is an output command that may be used to get the value of a variable or array element in another Box. This command is useful in situations in which it is necessary for one Box to monitor the status of another Box. This situation may arise when conducting experiments on yoked subject pairs. (See use of K-Pulses and the special variable named "BOX").

Syntax: GETVAL P1 = P2,P3

Where: P1 = The variable or array element in the present Box to be set.

                P2 = The Box number from which a datum is requested.

                P3 = The variable or array element whose value is being read.

Example:

```
S.S.1,
S1,
  1": GETVAL A = 2,B ---> SX  \ Set A equal to the value of
                              \ variable B in Box 2
```

Sometimes it is desirable to get a variables value from a prior day's run for the same Box. See Appendix C for an advanced programming technique to call Backproc.pas for this propose.

Data Handling Commands

SHOW

SHOW may be used to display data on the screen while a procedure is running. Each SHOW command takes three arguments. The first is the screen position (1-200), the second is the descriptive label, and the third is a number, variable or constant to be displayed on the screen. Each of up to 16 procedures may independently display the values of up to 200 variables along with descriptive labels on the screen. See the MED-PC User's Manual for full details on how these are displayed on the screen.

Syntax: SHOW P1,P2,P3

Where: P1 =   Whole number in range 1…200 expressed as an array element, variable, constant, or number.

                P2 =   A descriptive label that may contain upper and lower case letters, spaces, and digits. The label may be up to 255 characters long, but shorter descriptions are more practical.

                P3 =   Number, variable, constant, array element, or mathematical expression.

Comments: There may be up to 200 SHOW's per Box.

                The SHOW command is a secondary function and may be delayed while Boxes are processed; data saved, or sent to print manager.

Stringing is permissible.

Do not use the following characters in the descriptive label portion of a SHOW statement: { } \ ; " ' or a comma.

Example A:

The following example displays the label "Left Lever Presses" in the tenth position on the SHOW screen (last position in the second line of the SHOW area). Every second the value of X will increment by one and will be reflected on the screen.

```
S.S.1,
S1,
  1": ADD X; SHOW 10,Left Lever Presses,X ---> SX
```

Example B:

This example illustrates stringing of parameters in SHOW, the use of a constant value (5.01), the constant "^ThisValue" and a mathematical expression.

```
^ThisValue = 5


LIST A = 2, 5


S.S.1,
S1,
  1": SHOW 1,Value1,5.01, 2,Value2,^ThisValue, 3,Math,A(0)+A(1) ---> SX
```

When loaded to Box 1 this example creates the following display:

```
1) VALUE1   5.01     VALUE2   5     MATH   7
```

Updating the SHOW Display

SHOW commands are not displayed in real-time. When a Box issues a SHOW command, the data values are retained, but the data will not actually be displayed until the runtime system has time to update the screen. The values eventually shown on the screen will reflect the values of their respective variables at the moment that their respective SHOW commands were issued. In practice, the SHOW screen is usually updated within a small fraction of a second after any changes are made by active Boxes; most users would probably not even notice that updates are not in "real-time" except in the case of displaying running response totals for rapidly-responding subjects, in which case the response total shown on the screen will discontinuously count up. For example, a pigeon responding in a Box controlled by the following code would most likely produce SHOW output that appears on the screen as a discontinuously incrementing counter (perhaps incrementing as 3, 7, 8, 12, etc.):

Example A:

```
^Feeder = 5


S.S.1,
S1,
  20#R1: ADD Y; SHOW 2,RFS,Y; ON ^Feeder ---> S2

S2,
  2": OFF ^Feeder ---> S1


S.S.2,
S1,
  #R1: ADD X; SHOW 1,Responses,X ---> SX
```

Example B:

In the above sample, the displays are not present until the animal begins responding (SHOW 1) or responds 20 times (SHOW 2). Show Labels may be displayed even while variable values are zero to confirm that a program is running with the following changes to the above code.

```
^Feeder = 5


S.S.1,
S1,
  #START: SHOW 1,Responses,X, 2,RFS,Y ---> S2

S2,
  20#R1: ADD Y; SHOW 2,RFS,Y; ON ^Feeder ---> S3

S3,
  2": OFF ^Feeder ---> S1
```

CLEAR - Remove SHOW Counters

CLEAR works in conjunction with SHOW. As its name suggests, CLEAR blanks out SHOW command output. For example, one might SHOW successive IRT's or other events in a trial (up to 200) and then issue CLEAR 1,200 to erase the output of those SHOWs in preparation for the next trial. CLEAR may be used to remove any sub-range of counters and may take variables, numbers, calculations, or constants as arguments.

Syntax: CLEAR P1,P2

Where: P1 = number, variable, array element, or constant.

                P2 = number, variable, array element, or constant.

Comment: 1 <= P1 <= 200 i.e., P1 between 1 and 200 (inclusive)

                P1 <= P2 <= 200 i.e., P2 greater than or equal to P1 and less than or equal to 200.

It is not necessary to include a CLEAR command to initialize or clear the output left behind by one Box when the Box is reloaded. Each Box's SHOW area is automatically cleared when a Box is loaded.

The following examples are all legal:

```
#Z10:  CLEAR 40,50 ---> SX
60#R1: CLEAR A,B ---> SX
#R2:   CLEAR ^FIRST,^LAST ---> SX
```

PRINT

PRINT is an output command that may be used to generate printouts. By default, all variables and array elements are printed when this command is used. However, considerable control may be exerted over the appearance and contents of printouts through the PRINTVARS, PRINTFORMAT, and PRINTOPTION commands.

All printing is done through the Windows Print Manager. In the event that the printer is offline or out of paper or there is some other problem, Windows will present a dialog box indicating the nature of the problem. It is generally best to correct the problem and then select "RETRY." Data will not generally be lost under such circumstances.

Syntax: PRINT (must be in output section)

Comment: PRINT takes no arguments.

Example:

The following code illustrates a FI-10 interval schedule with two a second reinforcement. Each response is added to variable A. At the end of 60 minutes a complete annotated printout will occur of all variables even though only the A variable may have a value other than 0. No data is saved to disk.

```
S.S.1,
S1,
  10": ---> S2

S2,
  #R1: ON 1 ---> S3

S3,
  2": OFF 1; ADD B ---> S1


S.S.2,
S1,
  #R1: ADD A ---> SX


S.S.3,
S1,
  60': PRINT ---> S2

S2,
  1" ---> STOPKILL
```

Printing Issues

Different procedures in the same runtime program may use different formats.

Different MPC procedures may use different combinations of printer options without interfering with the options specified by other procedures, even when different procedures with different combinations of options are being run in multiple Boxes. For example, if Box 1 uses portrait and Box 2 uses landscape, each Box's data will print properly.

Printouts reflect data values at the moment of actual printing, not values at the time of printing requests.

When a request is made to print a Box's data, either from the menu system or from within a MedState Notation procedure, the request is not immediately fulfilled. A finite amount of time is required for the runtime system to generate the printout. Until the printout is generated, the data values within a printout may "float" during the period between the printing request being issued and the time at which the data are actually printed. For example, consider the following code:

```
DIM A = 2000
DIM Z = 2000


S.S.1,
S1,
  1": ADD A(0), Z(2000); PRINT ---> SX
```

In this example, the values of A(0) and Z(2000) are incremented every second and will always be equal. If a request to print is made after the Box has finished running, the values of data printed for A(0) and Z(2000) will be identical. However, if the data for this procedure are printed while the Box is still active, the values of both variables will continue to increment while the runtime system is constructing the printout. Different values are likely to be printed for the two variables, with the value of Z(2000) being larger than that of A(0) because variables and arrays are printed in alphabetical order. The basis for the discrepancy is that it takes time to generate a printout and A(0) will be placed on the printout before Z(2000).

There are, however, several different ways to ensure that data on a printout are "frozen" at the time of a printing request:

1) Issue a print command after a procedure has finished its work and it's data values will no longer change. The preceding code example could be changed to:

```
DIM A = 2000
DIM Z = 2000

S.S.1,
S1,
  1": ADD A(0), Z(2000) ---> SX

S.S.2,
S1,
  60': PRINT ---> STOPABORT  \ Values will stop changing when
```

```
                              \ this line executes
```

2) Transfer the to-be-printed data to special arrays or variables that won't be updated during the time between a printing request and the generation of the printout. The code example could become:

```
DIM A = 2000
DIM B = 1
DIM Z = 2000

PRINTVARS = B

S.S.1,
S1,
  #K1: SET B(0) = A(0), B(1) = Z(2000); PRINT ---> SX
                               \ "Freeze" the data and Print the values
                               \ by issuing K1 from the keyboard
  1": ADD A(0), Z(2000) ---> SX
```

3) Transfer an entire array with the COPYARRAY command.

```
DIM A = 2
DIM B = 2


PRINTVARS = B

S.S.1,
S1,
  #R1: ADD A(0) ---> SX
  #R2: ADD A(1) ---> SX
  #R3: ADD A(2) ---> SX

S.S.2,
S1,
  5': COPYARRAY A,B,3; PRINT ---> SX  \ From A to B Copy 3 Elements
                                      \ (0, 1, 2) and then Print.
```

4) Print data after a Box has stopped executing but before it's data is written to disk and before the Box is reloaded. Simply put, if a Box isn't running, its data values can't change.

In many instances it may not even matter whether data values float, particularly when the printout is being generated to get a quick look at the data, as opposed to generating archival printouts. Additionally, if small amounts of data are printed, the time interval between printing the first and last values will typically be quite small and often not detectable.

In addition to the asynchrony between printing the first and last data elements within a Box that may arise when printing while a Box is running, the amount of time it takes for a printout to appear on the printer varies according to a variety of factors. Typically, a printout is generated within seconds after a request. This may change though if a large number of printing requests have preceded the present request or if a great deal of disk-writing activity is underway. Specifically, printout generation does not occur while disks files are being written.

Queuing of Printouts and Availability of Data for Printing

When a request to print data is received by MED-PC, the request is placed in a queue or waiting line until MED-PC has time to process the request. Assuming that MED-PC is not occupied performing other tasks, the printout is generated in memory before actually being sent to the printer. During printout formation, the on screen memory indicators will rapidly run down until only about 5K remain. As the printout actually gets sent to the printer, memory locations are constantly released. As memory again becomes available, more printout is formed in memory.

A request to print data will be fulfilled whenever any of the following conditions are met:

1) The print request is issued while the Box is still running.

2) The Box is not running, but the Box was terminated by an ABORT (as opposed to KILL) and the Box has not been reloaded and the data has not yet been written to disk. The moment data is written to disk or abandoned; print requests will no longer be honored. However, if a print request has been issued, one may immediately write data to disk and/or reload the Box without adversely affecting the printout -- the request will be honored. Note, though, that in systems running under tight memory constraints, the release of memory normally associated with writing data to disk will be delayed until the data is finished being printed.

3) The Box is not running, but a request to print the Box's data is still in the print queue. This feature may be of little practical value.

Example:

```
60': PRINT ---> STOPKILL
     \ Data will print, even though transition is to STOPKILL
     \ because the print request was issued before the STOPKILL
```

Miscellaneous

The End Date and End Time indicators on printouts will be set to 0 when a print request is issued from within a procedure or when a print is issued from the keyboard and the Box is still running unless the "Print the time, not 00:00:00" data option is selected in the Hardware Configuration Utility.

FLUSH

Under typical scenarios, MSN programs are designed such that data is written to disk as the result of a transition to STOPABORT or STOPABORTFLUSH. Both of these commands also terminate execution of the program. However, it is sometimes desirable to write data to the disk and then continue execution of the program. This may be accomplished by placing the FLUSH command in the output section of an MSN statement.

Syntax: FLUSH

Comments: The data is written to disk without any user intervention, similar to the effects of STOPABORTFLUSH, except that program execution continues. Any formatting specified by DISKVARS or other formatting commands will influence the format of the resulting disk file. The name of the data file will correspond to the naming scheme specified during installation of the system, and multiple flushes will cause data to be appended to the same file -- multiple files will not be created, regardless of the number of flushes from within the same session. Note: these data files are distinct from the automatic backup file maintained by MED-PC.

Examples and Discussion:

```
S.S.1,
S1,
  #START: ---> S2

S2,
  #R1: ADD A ---> SX


S.S.2,
S1,
  #START: ---> S2

S2,
  10': FLUSH ---> SX
```

In this very simple example, issuing #START causes both State Sets to enter State 2. In State Set 1, every response increments variable A. In State Set 2, all data for the session is written to the disk every 10 minutes without any user intervention.

Miscellaneous Commands

DATE and TIME

These commands return information about the Date and Time. Having access to this may be useful in IF statements for setting experimental parameters on the basis of Time or Date information.

Syntax: DATE P1,P2,P3

TIME P1,P2,P3

Where: P1, P2, P3 = Variable or array element

Information returned by DATE into the parameters:

P1: Month

P2: Day

P3: Year

Information returned by TIME into the parameters:

P1: Hours (12 or 24 hour based on Windows setting)

P2: Minutes

P3: Seconds

Example:

In the following example, the calendar information is returned by DATE. If the first parameter (A) returns 10 then F is set equal to 5, establishing an FR 5. If the month is equal to anything other than 5, then the FR is set equal to 10.

```
S.S.1,
S1,
  #START: DATE A,B,C;
          IF A = 10 [@OctFR, @NotOctFR]
              @OctFR:    SET F = 5  ---> S2  \ FR 5
              @NotOctFR: SET F = 10 ---> S2  \ FR 10

S2,
  F#R1: ON 2 ---> S3

S3,
  2": OFF 2 ---> S1
```

BKGRND

BKGRND is an output command that may be used to run certain procedures in the background. BKGRND procedures must be written in the file Backproc.pas that is included with MED-PC. Up to ten (1…10) different BKGRND procedures may be declared at one time. Multiple boxes may simultaneously request the same BKGRND procedure without problems, because MED-PC will properly track which Boxes have requested the procedure. The same Box may have multiple simultaneous active requests for different BKGRND procedures, but note that a single Box may not request the same BKGRND procedure a second time until its previous request has been completed.

Syntax: BKGRND P1

Where: P1 = a number from 1 to 10 inclusive.

Example:

```
S.S.1,
S1,
  1": BKGRND 5 ---> S2
```

For more information on the BKGRND procedures please see Appendix C.

Listing of Special Identifiers

BOX

This identifier is synonymous and reflects the Box number of the Box in which the MSN procedure is executing.   BOX may be especially useful in conjunction with inter-Box communication (see section on inter-Box K-Pulses). Under no circumstances should this value be altered with a SET command.

SUBJECTNUMBER
EXPNUMBER
GROUPNUMBER

These values reflect the Subject, Experiment and Group numbers of the subject in the Box in which the experiment is executing. These values may be safely altered with a SET command but be aware that the Box's status line on the runtime screen will not be automatically updated to reflect changed values.

STARTMONTH
STARTDATE
STARTYEAR
STARTHOURS
STARTMINUTES
STARTSECONDS

These values reflect the Date and Time at the moment when the current Box was loaded with the "Open Session" menu selection. They are equal to the values displayed on the runtime screen on the Box's status line. STARTYEAR is a 2-digit number reflecting the last 2 digits of the year. For example, in 1991, STARTYEAR will equal 91[7]. Note: the use of the term start in this context bears no relationship to the issuance of a #START command from the keyboard. Procedures actually "start" the instant they are loaded. These values may be safely altered with a SET command but be aware that the Box's status line on the runtime screen will not be automatically updated to reflect changed values.

ENDMONTH
ENDDATE
ENDYEAR
ENDHOURS

---

[7]   Use a four-digit date if the Y2KCOMPLIANT command is used in the code. For a discussion on this command, see the next section of this Appendix "Commands that come before the first State Set."

ENDMINUTES

ENDSECONDS

These variables are set equal to the Date and Time when the Box's execution is terminated with KILL, ABORT or ABORTFLUSH. During procedure execution these variables are normally equal to 0. These values may be altered during procedure execution with a SET statement, but when procedure execution terminates, they will be automatically changed to the Date and Time of termination.

CURRENTMONTH

CURRENTDATE

CURRENTYEAR

CURRENTHOURS

CURRENTMINUTES

CURRENTSECONDS

These variables reflect the Date and Time at approximately the time they are accessed in an expression. It is important to recognize that CURRENTSECONDS is not a precise reflection of the present time. Under some circumstance, this variable (and all other CURRENT variables) may be a few seconds behind the actual value of the computer's clock because their values are updated only as MED-PC has spare time remaining after servicing Boxes. These values are in no way related to the internal timing of experimental events; experimental events are timed independently of the computer's clock. For a precise access to time, utilize BTIME. These values may be altered but it is pointless to do so because MED-PC will automatically correct them within a few seconds of their alteration.

SECSTODAY

This variable is a single variable that contains the cumulative number of seconds past midnight. SECSTODAY is subject to the same accuracy limitations as CURRENTSECS. SECSTODAY is useful for recording the approximate (accurate to within a few seconds) time of day when an event occurs. It is no more or less accurate than CURRENTHOURS, CURRENTMINUTES and CURRENTSECONDS, but does allow one to condense the information contained in those three variables into a single number. This allows one to record the time of occurrence of events in a minimum number of array locations. Subsequent data analysis software could be used to reconstruct hour, minute and second information.

DATETODAY

DATETODAY is a single number that condenses CURRENTYEAR, CURRENTMONTH and CURRENTDATE into a single number in the format YYMMDD. For example, June 2, 1990 would be expressed as 900602.

BTIME

This number is based upon MED-PC's internal timer interrupt system. BTIME is used internally to time experimental events.   BTIME is set to 0 when MED-PC is loaded and continues to increment throughout the session every time an interrupt occurs.   BTIME increments 1000/RESOLUTION times per second, where RESOLUTION is the timing resolution declared during installation. On a 50 ms system, BTIME increments 20 times per second. On a 20 ms system, BTIME increments 50 times per second.

BTIME may be useful for recording elapsed times, but there is no particular advantage to this technique over recording elapsed time by incrementing a MED-PC variable on a periodic basis with conventional MSN timing statements.

BTIME must never be altered.

Transitional Commands

Null Transition (SX)

Sometimes it is desirable to have a transition that does not reset the input conditions for the entire state. MedState Notation can do this with a command called SX, which is also known as the Null Transition. In code it would come after the transition arrow:

Syntax: `INPUT: OUTPUT ---> SX`

The following two examples will help explain the power of the Null Transition.

Example 1:

```
S2,  \ 1 minute ITI.  Count Responses during ITI.
  1': ---> S3
  #R1: ADD C ---> SX  \ A Response on Input 1 does not
                      \ reset the 1 minute timer
```

In Example 1 the program times 1 minute and counts all responses that happen on Input 1. Responses on Input 1 do not affect the 1 minute timer because of the transition to SX.

Example 2:

```
S2,  \ 1 minute ITI.
     \ Restart the ITI timer if animal presses the lever.
  1': ---> S3
  #R1: ADD C ---> S2  \ A Response on Input 1 does
                      \ reset the 1 minute timer
```

In Example 2 the program times 1 minute and counts all responses that happen on Input 1. But in this example a Response on Input 1 also resets the 1 minute timer because of the transition to S2. The animal is being punished for responding during the ITI.

STOPABORT and STOPKILL

These commands are actually the names of two special transitions. They are placed following a transition arrow and cause the procedure to immediately stop executing. Any outputs currently turned on get shut off immediately unless turned on by LOCKON. In addition, the Box's status lines on the monitor are cleared. The difference between STOPABORT and STOPKILL is that STOPABORT retains the values of all variables and array elements in memory for subsequent dumping into a file (Flush) and print can be executed before the dump. The data from procedures terminated by STOPKILL is not recoverable - the data is not placed in the dump queue. STOPABORT and STOPKILL automatically perform the same functions as their manual equivalents on the "close sessions" window. Please see the MED-PC User's Manual on how to save the data manually.

Syntax: Input: Output ---> STOPABORT

            Input: Output ---> STOPKILL

Example:

```
S.S.1,
S1,
  10#R1: ADD A; ON 1 ---> S2

S2,
  2": OFF 1; IF A = 50 [] ---> STOPABORT
```

STOPABORTFLUSH

STOPABORTFLUSH is identical to STOPABORT in that it is a transition that turns off all outputs, except those that are LOCKON'd, and stops procedure execution. Additionally, this command causes the data to be written to disk automatically. File structure is determined by the internal file format declared during installation and any "DISK" commands prior to State Set 1. File naming defaults to the scheme declared during installation unless a custom file name is assigned from the "Open Session" window.

Executing STOPABORTFLUSH will cause all data waiting to be written to the disk to be transferred to disk even if the data is awaiting transfer as the result of other Boxes executing STOPABORT.

Syntax: Input: Output ---> STOPABORTFLUSH

Example:

```
S.S.1,
S1,
  #R1: ADD A ---> SX
  #R2: ADD B ---> SX


S.S.2,
S1,
  60' ---> STOPABORTFLUSH
```

Commands that Come Before the First State Set

DIM - Dimensioning Arrays

DIM is used to define (dimension) the size of arrays. Like variables, array names are letters of the alphabet; however declaring a letter of the alphabet to be an array precludes its use as a simple variable (i.e., "A" can not be both an array and a simple variable). By default, all letters are defined as simple variables.

Syntax: DIM P1 = P2

Where: P1 =   A letter of the alphabet to be declared as an array.

                  P2 =   The maximum subscript of the array. Because arrays are always zero-based (zero is the lowest subscript), the number of elements is P2 + 1. The elements range from 0...P2.

Comments: The total number of variables and array elements per MSN program may not exceed 1,000,000.

Example:

```
DIM B = 10  \ Declare "B" as an array with elements 0...10


S.S.1,
S1,
  1: ADD B(5) ---> SX
```

Declaring an Array with the LIST Command

List is first placed before State Set 1 to dimension an array and assign a value to each element in an array. It can then be used in the output section of a statement to select each value in sequence. When the last element in the list has been used, selection restarts at the beginning of the list.

Syntax A: Defining the array

                LIST P1 = P2, P3, ..., Pn

Where: P1 = The name of the array to be declared (A...Z)

                P2, P3, ..., Pn = Number

Comments: Each line must end on a comma <CR>.

Syntax B: Selecting elements from the array.

                LIST P1 = P2(P3)

Where: P1 = Variable or array element

                P2 = Array from which an item is to be drawn

                P3 = Variable or array element used as subscript to array P2

Comments: Stringing is permissible

The value of P3 is automatically incremented by the LIST command. Assignments to P3 through other commands will affect the selection of subsequent items via the LIST command (i.e. it is possible to skip or retrace elements in the array via manipulation of the P3 subscript.) This must be done with caution, however, as an illegal subscript value will be ignored by MED-PC, reverting the array back to subscript zero.

Array P2 may be declared with LIST or DIM statement.

Examples and Discussion:

Running the following procedure would result in the following pattern of outputs: 1, 2, 3, 1, 2, 3, 1, 2, 3, ...

Example A:

```
LIST A = 1, 2, 3

S.S.1,
S1,
  1": LIST B = A(K); ON B ---> S2

S2,
  1": OFF B ---> S1
```

In the first LIST statement above, an array named A is declared. The lowest element of an array is always referenced as Element 0. Element 0 contains the value 1, Element 1 contains the value 2 and Element 2 contains the value 3.

One second after this procedure is loaded, the statement "LIST B = A(K)" sets B equal to the value of element K of array A. Since all variables are automatically set to 0 at the beginning of program execution, the value of B is set equal to A(0) and ON B causes output 1 to be turned ON. One second later, output 1 is turned OFF in State 2. Following assignment of the value of A(K) to B, 1 is automatically added to the value of K so that the next time the list statement is executed, the array index (K) for array A is equal to 1 giving B a value of 2 to turn ON output 2.

The LIST command continues to select successive array elements until the end of the list is reached. When the last element has been accessed, the array index (K) is reset back to 0.

Example B:

```
LIST A:  5, 10, 15, 20, 25, 30,  \ Note that each line
        30, 35, 40, 45, 50, 55,  \ must end in a comma
        60, 65, 70, 75, 80, 85   \ except for the last one.
```

Declaring Constants

The clarity of MSN programs may be enhanced through the use of constants. Constants are user-defined meaningful names that may be used in place of whole numbers in MSN programs. Constants are particularly useful for providing logical names for output numbers and Z pulses. Constant names must always begin with a carat '^.'

Syntax: ^CONSTANT = P1

Where: CONSTANT = the name being assigned

                P1 = the Input or Output number being assigned to a constant.

The following code illustrates several uses of constants.

```
\ Inputs
^LeftKey = 1

\ Outputs
^Feeder = 1
^Light  = 2


\ Z-Pulses
^RFBegin = 1
^RFEnd   = 2


\ Offsets into C Array
^RFS = 0  \ Counter 0 will count RFS

DIM C = 5


S.S.1,
S1,
  0.01": ON ^Light ---> S2

S2,
  10#R^LeftKey: ON ^Feeder; Z^RFBegin; ADD C(^RFS) ---> S3

S3,
  2": OFF ^Feeder; Z^RFEnd ---> S2
```

Comments:

1. It is acceptable to assign time values to constants

    (e.g., prior to the first State Set: ^FIVal = 30")

2. Do not attempt to change the value of a constant within a State Set

    (e.g., 1": SET ^FIVal = 60" ---> SX is illegal)

3. Constants must be declared as having an integer value.

    (e.g. ^Feeder = 1.1 is illegal)

Named Variables (VAR_ALIAS Command)

MED-PC allows the user to create names for variables that will appear in the Variable Changing dialog and Box-Loading Expert of the runtime system. This allows users to set parameter values using meaningful labels. For example, the VAR_ALIAS command may be used so that the user will be able to set the value of a variable named "FR Size," rather than an obscure array element, such as "C(10)." Note that variable aliases do not have any use within the body of MSN programs -- they are simply directives placed before the first State Set that establish meaningful aliases (essentially synonyms) for program variables.

Figure 13.3 is an example of the standard dialog used in the runtime system to view and manipulate variables and array elements. Figure 13.4 illustrates the use of the VAR_ALIAS command and was produced by the sample code below.

Figure 13.3 - Change Variables Dialog

The "Named Variables" Dialog corresponding to the sample code following the screen shot:

Figure 13.4 – Named Variables Dialog

Syntax: VAR_ALIAS A = B

Where: A = A descriptive label for the variable or array element
                B = The variable or array element

Examples:

```
\ Concurrent FR FI


VAR_ALIAS FR Size       = A(1)  \ Default = 10
VAR_ALIAS FI Size (Secs) = B    \ Default = 30"


DIM A = 10


S.S.1,  \ FR
S1,     \ Set default FR to 10
  0.01": SET A(1) = 10 ---> S2

S2,
  A(1)#R1: ON 1 ---> S3

S3,
  0.1": OFF 1 ---> S2


S.S.2,  \ FI
S1,     \ Set default FI to 30".
        \ Note, B will display as 3000 on a system
        \ with a 10ms Resolution
  0.01": SET B = 30" ---> S2

S2,
  B#T: ---> S3

S3,
  #R1: ON 1 ---> S4

S4,
  0.1": OFF 1 ---> S2
```

Comment: Variable aliases cannot be used within MSN programs instead of variable letters.
                For example, the following will not work:

```
VAR_ALIAS Response Total = A


S.S.1,
S1,
  #R1: ADD Response Total ---> SX
```

INITIALIZATIONCODE

In some instances it may be desirable to execute some code immediately, before a program begins to execute. This may be done through making calls to inline Pascal procedures residing in the User.pas.

INITIALIZATIONCODE executes immediately, prior to the first clock tick of an MSN program's execution. This command must be placed before the first StateSet. In the following example, StartActivityChamber executes before S1 of S.S.1.

Examples:

```
\ Start the Activity Monitor chamber running
\ before the Box finishes loading.
INITIALIZATIONCODE = ~StartActivityChamber(BOX);~


S.S.1,
S1,
  0.01": ---> S2
```

Comment: The code can be any valid function/procedure that resides in the User.pas. The procedure must not be named "Initialization" as this is a Pascal reserved word.

FINALIZATIONCODE

In some instances, it may be desirable to execute some code immediately after a program stops running. This may be done through making calls to inline Pascal procedures residing in the User.pas.

FINALIZATIONCODE executes immediately after the last clock tick of an MSN program's execution. This command must be placed before the first stateset. In the following example, StopActivityChamber executes after STOPABORTFLUSH.

```
\ Stop the Activity Monitor chamber after
\ the Box has finished running.
FINALIZATIONCODE = ~StopActivityChamber(BOX);~


S.S.1,
S1,
  10': ---> STOPABORTFLUSH
```

Comments: The code can be any valid function/procedure that resides in the User.pas. The procedure must not be named "Finalization" as this is a Pascal reserved word.

DEFINEMACRO

Some portions of code tend to be used repeatedly. For example, when operating a pellet dispenser, a tone and the dispenser may be turned on, and the houselight may be turned off. Redundantly writing out the code to do this in every state in which reinforcer delivery may occur tends to be tedious and error prone. One approach to avoiding this problem is to use Z-Pulses. Another approach is to use code macros. Code macros are code snippets that are defined and given a name prior to the first StateSet using the DEFINEMACRO command. Subsequently, the code snippet can be used in the rest of the program by referencing the name of the macro.

Code macros that are common to multiple programs can be saved in a single "Library" file. Doing so can simplify maintenance of code and enhance efficiency. Library files may be created using the editor built into the translator by saving a file containing only DEFINEMACRO commands. The file should be saved as a "Library" file by selecting "MSN Macro Library" from the "Save as type:" dropdown list of the Save As dialog box. Library files have a filename extension of ".LIB". Macros in the library file may be used in an MSN (.MPC) program by naming the library file in the MSN program using the Library command.

Declaring a macro

Syntax: DEFINEMACRO P1 = P2

Where: P1 = A name for the macro. The name may contain spaces, letters, digits and underscores

                P2 = Any valid MSN code enclosed in curly brackets {}.

Example:

```
DEFINEMACRO Outputs = {ON 1, 2, 3}
```

Invoking a macro

Syntax: %P1%

Where: P1 = The name of a macro. The name must be surround by "%" signs.

Example:

```
S.S.5,
S1,
  #Z2: %Outputs% ---> S2
```

Including code from a library file

Syntax: Library P1

Where: P1 = The name of a file containing DEFINEMACRO statements.

                  The file must be in the MPC subdirectory (where .MPC program files are
                  kept). The file name must not include a file path but it must include an
                  appropriate extension. Although the library file may have any filename
                  extension, the use of ".lib" is recommended.

Comments: Nesting of library files is permissible.

                  Library files may themselves include library statements to access code macros
                  defined in other libraries.

Correct examples:

```
Library Constants.lib
Library My Library.lib
```

Incorrect example:

```
Library C:\MEDPC IV\MPC\Constants.lib
```

Examples and Discussion:

Example A:

```
^HouseLight = 1
^Pellet     = 2
^Tone       = 3


DEFINEMACRO DeliverRF  = {OFF ^HouseLight; ON  ^Tone, ^Pellet}
DEFINEMACRO RFFinished = {ON  ^HouseLight; OFF ^Tone, ^Pellet}


S.S.1,
S1,
   10#R1: %DeliverRF% ---> S2

S2,
   1": %RFFinished% ---> S1
```

The code contained in a macro may span more than one line and the reference to a macro does
not need to be embedded in an MSN statement – the reference to the macro may be free
standing.

Example B:

```
DEFINEMACRO SS1 = {
                    S.S.1,
                    S1,
                        #START: ON 1 ---> S2
                    S2,
                        10" ---> S3
                  }


%SS1%
```

Macro definitions may themselves contain "nested" references to other macros. The following code expands properly.

Example C:

```
DEFINEMACRO Outputs = {1, 2, 4;}
DEFINEMACRO On      = {ON %Outputs%}
DEFINEMACRO SS1     = {
                       S.S.1,
                       S1,
                           1": %On% ---> SX
                      }


%SS1%
```

When writing programs containing code macros it is sometimes helpful to see the program with all of the code substitutions implemented – rather than viewing the first example as it is written, it may be helpful to view it as it will appear to the MED-PC translator. This can be particularly helpful if the translator reports that there are syntax errors. Macros may be "expanded" by selecting "Expand Macros" from the File menu while viewing a program that contains macros. If one were to do this for Example A, a window containing the following would appear:

```
^HouseLight = 1
^Pellet     = 2
^Tone       = 3


S.S.1,
S1,
  10#R1: OFF HouseLight; ON ^Tone, ^Pellet ---> S2

S2,
  1": ON HouseLight; OFF ^Tone, ^Pellet ---> S1
```

If desired, the now expanded version of a file containing macros may be saved as a separate program.

When there is code that is written that can be used in multiple programs it is often helpful to save the code as a macro in a library file. For example, a library file named Constants.lib might contain:

```
DEFINEMACRO PigeonConstants = {
                            ^HouseLight = 1
                            ^Feeder    = 2
                            ^KeyLight  = 3
                          }

DEFINEMACRO RatConstants = {
                          ^Tone      = 1
                          ^Pellets   = 2
                          ^HouseLight = 3
                        }
```

An MSN program named Pigeon.mpc could utilize the PigeonConstants in Constants.lib as follows:

```
Library Constants.lib

%PigeonConstants%


S.S.1,
S1,
   #START: ON ^HouseLight ---> S2
```

PRINTVARS

It is often desirable to print only a subset of the variables and arrays in a procedure. This is particularly true when many of the variables are used internally by the procedure and do not contain data. Additionally, when collecting hundreds or thousands of data points per session, it would be convenient to be able to print a few key indices to the printer after every session, and yet be able to save the detailed counters to disk file for later analysis.

The above objectives may be accomplished by using the PRINTVARS command. This command may be used to declare a list of variables that will be printed whenever a PRINT command is issued. The PRINTVARS command affects printing irrespective of whether the command to print was issued from within a state table or by a keyboard command. The PRINTVARS command in no way affects the variables that will be written to disk (but a parallel command, DISKVARS, is provided).

In the absence of a PRINTVARS directive, all variables and arrays (A-Z) are printed. To print selected variables, place a PRINTVARS directive before the first State Set of the procedure. The exact placement of PRINTVARS does not matter, provided that it is before the first State Set.

Syntax: PRINTVARS = P1, P2, ..., P26

Where: P1...P26 are variables or arrays A through Z

Comment: PRINTVARS must be placed before the first State Set.

In the following example, the PRINTVARS directive specifies that printouts should contain only variables/arrays A, J & K.

```
DIM J = 5


PRINTVARS = A, J, K
\ Printouts will contain Variables A & K and Array J.
\ This statement has no effect on what is written to disk.


S.S.1,
S1,
60': PRINT ---> STOPABORT  \ Print variables specified by PRINTVARS
                           \ from within state table after 1 hour
```

PRINTFORMAT

MED-PC automatically prints numbers such that 12 spaces are set aside for each number, with 8 digits reserved for the integer part of the number (to the left of the decimal), 1 space is used for the decimal and 3 spaces are provided for the decimal portion of the number. An example of a number printed in 12.3 format (the meaning of 12.3 will be detailed below) is, "12345678.123."

In many instances, it is useful to print data in other formats, particularly when trying to increase the amount of data printed per page. Placing a PRINTFORMAT statement before the first State Set of the procedure will control the printed format of numbers. PRINTFORMAT takes one argument consisting of a decimal number in which the integer (to the left of the decimal) indicates the total number of spaces to be occupied by the number and the decimal portion indicates the number of spaces to be set aside for the decimal portion of to-be-printed numbers.

Syntax: PRINTFORMAT = P1.P2

Where: P1 =   Number indicates the total number of spaces to be occupied by the number including the decimal point.

                P2 =   Number indicates the number of spaces to be set aside for the decimal portion of the number.

Examples:

```
PRINTFORMAT = 5.1  \ Print in five space, with 3 to left of decimal
                   \ 1 to right as in 123.1

PRINTFORMAT = 7.2  \ 1234.12

PRINTFORMAT = 6.0  \ 123456
```

The use of a PRINTFORMAT statement has no effect upon the internal representation of numbers. If multiple PRINTFORMAT statements are used in the same .MPC procedure, then only the last one is implemented.

If the digits to the left of the decimal point exceeds the total number of spaces set aside by the PRINTFORMAT statement, then the general formatting rules are temporarily set aside and the number is printed in as many spaces as are needed to represent the integer portion of the number. This may result in the printed line "spilling" onto the next line of the page. If the decimal portion of a number exceeds the space allocated, the number printed is rounded to the nearest value.

PRINTOPTIONS

PRINTOPTIONS provides control over the appearance of the headers that appear at the beginning of printouts. The headers include information such as the time that the experiment was loaded and the name of the program used to control the experiment. There are two options for the appearance of headers: FULLHEADERS versus CONDENSEDHEADERS. If PRINTOPTIONS is not explicitly specified, the default printout is to print a condensed header (CONDENSEDHEADERS), with no form feed (NOFORMFEEDS). When specified multiple options are separated by commas, and any option not specified will stay at its default value. Several samples are provided below. The FORMFEEDS option specifies that a page will be ejected from the printer after every PRINT command, whereas NOFORMFEEDS indicates that the data from one Box should be printed immediately after the last Box's data without ejecting a page.

Syntax: PRINTOPTIONS = P1, P2

Where: P1 = FULLHEADERS or CONDENSEDHEADERS

              P2 = FORMFEEDS or NOFORMFEEDS

Comments: Note that DOS versions of MED-PC also provided for an additional parameter: 80 vs. 132 columns. The translator will still accept these parameters in the service of downward compatibility, but these options have no effect on the appearance of printouts under the present version. These differences reflect fundamental differences in the approach taken to printing in Windows vs. DOS. Although there is no equivalent to the column width options (80 vs. 132), the present version of MED-PC can control the font size (and, hence, the amount of data on the page) via the PRINTORIENTATION, PRINTCOLUMNS and PRINTPOINTS commands that have been provided as enhanced alternatives to the older system.

              Commas separate multiple options

              CONDENSEDHEADERS and NOFORMFEEDS are the default settings and need not be specified.

              The order of options is irrelevant.

Examples and Discussion:
```
PRINTVARS    = A
PRINTFORMAT  = 9.2
PRINTOPTIONS = FULLHEADERS
```

```
    LIST A = 1234.67, 1234.67, 1234.67, 1234.67, 1234.67, 1234.67, 1234.67,
             1234.67, 1234.67, 1234.67, 1234.67, 1234.672, 1234.677


    S.S.1,
    S1,
      1": PRINT ---> STOPKILL
```

The code above will produce a printout similar to the following:

```
Start Date:  03/10/91
End Date:  03/10/91
Subject:  0
Experiment:  0
Group:  0
Box:  1
Start Time:  14:11:32
End Time:  14:11:33
Source Code:  PRINTSAM
A:
      0:   1234.67   1234.67   1234.67   1234.67   1234.67   1234.67   1234.67
      7:   1234.67   1234.67   1234.68
```

A second option is whether a form feed is issued after each Box is printed so that data for each Box begins at the top of the page. NOFORMFEEDS is the default setting, but FORMFEEDS causes the printouts to begin at the top of pages. Note that individual form feeds may also be issued from within the runtime menu system.

PRINTORIENTATION

This command is used to override system defaults with respect to whether a given printout occurs in Landscape (sideways) or Portrait (standard) orientation. The MED-PC default is Portrait.

Syntax: PRINTORIENTATION = P1

Where: P1 = Portrait or Landscape

PRINTCOLUMNS

The PRINTCOLUMNS command controls the number of columns in which the contents of arrays are printed. The use of this command will override any defaults set within the runtime menu system.

Syntax: PRINTCOLUMNS = P1

Where: P1 = The number of columns

Example, in which the C array will be printed in 3 columns:

```
    LIST C = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    PRINTCOLUMNS = 3

    S.S.1,
    S1,
      #K1: PRINT ---> SX
```

PRINTPOINTS

The PRINTPOINTS command controls the size of the font used to print data from the Box in which this command is issued. The use of this command will override any defaults set within the MED-PC menu system.

Syntax: PRINTPOINTS = P1

Where: P1 = The number of points (12 is the default)

Example, in which the data are printed in a small font:

```
LIST C = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10


PRINTPOINTS = 8


S.S.1,
S1,
  #K1: PRINT ---> SX
```

DISKVARS, DISKFORMAT, DISKOPTIONS, DISKCOLUMNS

DISKVARS, DISKFORMAT, DISKOPTIONS, and DISKCOLUMNS are analogous to the PRINT commands previously discussed, but are completely separate and independent. The disk commands determine which variables are saved to the hard disk and the format used during the save.

Syntax: DISKVARS = P1, P2, ..., P26

Where: P1...P26 are variables or arrays A through Z

Syntax: DISKFORMAT = P1.P2

Where: P1 =   Number indicates the total number of spaces to be occupied by the number including the decimal point.

                P2 =   Number indicates the number of spaces to be set aside for the decimal portion of the number.

Syntax: DISKOPTIONS = P1

Where: P1 = FULLHEADERS or CONDENSEDHEADERS

Syntax: DISKCOLUMNS = P1

Where: P1 = The number of columns

Example and Discussion:

Writing a data file with FLUSH takes place in the background, which is to say that FLUSH does not cause any interference with the speed and efficiency with which experimental events are processed; the data is written when the processor has free time. It is important to note that the amount of time that it takes to write a disk file is indeterminate; it is not advisable to make changes to the data structure(s) being written to disk until one may be reasonably certain that the data has actually been transferred to disk.

For a discussion of an analogous issue with the PRINT command, refer to the section of the manual covering that command. A way to cope with the indeterminacy is to collect data into one array and then periodically copy the contents of that array to a second array and then write the second array to disk.

```
\ Declare Array's A and B with elements 0...99
DIM A = 99
DIM B = 99


\ Declare that only Array B will be written to disk
DISKVARS = B


S.S.1,
S1,
   #START: ---> S2

S2,      \ Whenever a Response on Input 1 occurs, do the following:
         \ 1) Record the Elapsed Time since the last Response into A(I).
         \ 2) Clear X so that the Elapsed Timer is reset.
         \ 3) Update I so that the next Response is recorded into the
         \    next element of A.
         \ 4) If 100 Inter-Response Times have been recorded, its time
         \    to transfer data to disk by doing the following:
         \      A) Transfer the data to B so that the data is not
         \         altered by subsequent responses prior to MED-PC
         \         having an opportunity to transfer the data to disk.
         \      B) Set all elements of A to 0 so that new data may be
         \         logged into A.  Also set I to 0 so that recording
         \         resumes at the beginning of A.
         \      C) Issue the FLUSH command to request writing the
         \         elements of B to disk as time permits.
         \ 5) Transition is to S2 so that the 0.1" IRT timer is reset
         \    whenever a response occurs.
   #R1: SET A(I) = X, X = 0; ADD I;
        IF I = 100 [@Write, @NotYet]
          @Write: COPYARRAY A,B,100; ZEROARRAY A;
                  SET I = 0; FLUSH ---> S2
          @NotYet: ---> S2
   0.1": SET X = X + 0.1 ---> SX  \This line is the IRT timer.
         \ For some applications ADD X may be substituted
         \ in the Output Section for SET X = X + 0.1
```

Y2KCOMPLIANT

Y2K issues: A new MSN directive, "Y2KCOMPLIANT," has been created. The command gets placed before the first State Set and takes no arguments. The consequence of including this directive is that all years are 4 digits on printouts (disk and paper) and in all data files.

Syntax: Y2KCOMPLIANT

Example:

```
DISKOPTIONS  = FULLHEADERS
PRINTOPTIONS = FULLHEADERS


Y2KCOMPLIANT


S.S.1,
S1,
  1" ---> SX
```

Appendix B | Macro Commands

All Macro commands are listed in detail, with syntax, comments, examples and discussion. Each command is indexed for convenience.

LOAD

LOAD is the macro equivalent to the "File | Open Session" menu selection.

Syntax: LOAD BOX P1 SUBJ P2 EXPT P3 GROUP P4 PROGRAM P5

Where: P1 = The Box to be loaded. 1<= P1 <= 16

P2 = The Subject number

P3 = The Experiment number

P4 = The Group number

P5 =   The program name. P5 must be the name of a compiled MSN program

Example:

```
   LOAD BOX  1 SUBJ    3 EXPT    22 GROUP    15 PROGRAM BIGEXPT
```

Open a session in Box 1 for Subject 3, Experiment 22, Group 15. Run the program named BIGEXPT.

FILENAME

Custom filenames may be associated with an experimental session with the FILENAME macro command.

Syntax: FILENAME BOX P1 P2

Where: P1 is a Box number

P2 is a filename

The filename may be any legal Windows filename – up to 255 characters. An illegal filename will be detected when an attempt is made to write data to the illegally named file and a dialog box will be presented. The dialog will permit correction of the filename. If a file by the same name already exists, the data from the current session will be appended to the existing data.

Example:

```
   FILENAME BOX 3 R29AMPH.001
```

COMMENT

Comments may be associated with an experimental session with the COMMENT macro command. Comments appear at the end of printouts and data files.

Syntax: COMMENT BOX P1 P2

Where: P1 is a Box number

                P2 is the comment

Comments may be up to 256 characters in length, but fewer than 256 characters may appear on printouts unless a sufficiently small font is used.

Example:

```
COMMENT BOX 5 Morphine 10.0 mg/kg, 10 min pretreat.
```

MODIFY_IDENTIFIERS

MODIFY_IDENTIFIERS is the macro command for changing the Subject, Experiment, and Group identifiers associated with a session that is in progress. Session identifiers are used to generate filenames and appear on printouts and in data files.

Syntax: MODIFY_IDENTIFIERS BOX P1 SUBJECT P2 EXPERIMENT P3 GROUP P4

Where: P1 = A Box number

                P2 = The Subject identifier

                P3 = The Experiment identifier

                P4 = The Group identifier

Example

```
MODIFY_IDENTIFIERS BOX 1 SUBJECT 2 EXPERIMENT 5CSRT GROUP E304
```

Box 1 has been loaded. The command changes the Session Identifiers. The command line will be updated and printouts and data files subsequently generated will reflect the new values.

STOPABORTFLUSH

STOPABORTFLUSH is the macro equivalent to the MSN command "STOPABORTFLUSH" and the menu command "File | Close session" in conjunction with the "Stop, Save Data, Write Data to Disk" option. This command ends an experimental session and automatically writes the data to disk.

Syntax: STOPABORTFLUSH BOXES P1

Where: P1 = One or more Box numbers

Example:

```
STOPABORTFLUSH BOXES 2 5
```

Close the sessions in Boxes 2 and 5, then write the data to disk.

STOPABORT

STOPABORT is the macro equivalent to the MSN command "STOPABORT" and the menu command "File |Close Session" in conjunction with the "Stop, Save Data" option. This command ends an experimental session. The data is saved but is not automatically written to disk.

Syntax: STOPABORT BOXES P1

Where: P1 = One or more boxes to load.

Example:

```
STOPABORT BOXES 1 3 4 5
```

Close the sessions in Boxes 1, 3, 4 and 5 and save the data.

STOPKILL

STOPKILL is the macro equivalent to the MSN command "STOPKILL" and the menu command "File | Close Session" in conjunction with the "Stop, Abandon Data" option. This command ends an experimental session and abandons the data; the data cannot be subsequently saved to disk.

Syntax: STOPKILL BOXES P1

Where: P1 = One or more Box numbers.

Example:

```
STOPKILL BOXES 2
```

Close the session in Box 2.

SAVE_MANUAL

SAVE_MANUAL may be used in macros to write data to disk. This command is the macro equivalent to the menu command "File | Save Data Manually." This command will write to disk any data that was saved with either an MSN STOPABORT command or the menu equivalent. Before writing data to disk, a dialog box will be presented for each file. The dialog box provides the opportunity to change the MED-PC generated filename or to abandon the data. Note that this command is unnecessary when MSN programs terminate with the command STOPABORTFLUSH.

Syntax: SAVE_MANUAL

SAVE_FLUSH

SAVE_FLUSH is the macro equivalent to the menu command "File | Save Data (Flush)." This command will write to disk any data that were saved with either an MSN STOPABORT command or the menu equivalent. All data will be written to disk using MED-PC generated file names based on the file-naming scheme selected during installation of MED-PC. Note that this command is unnecessary when MSN programs terminate with the command STOPABORTFLUSH.

Syntax: FLUSH

PRINT

Data may be printed under macro control using the macro command PRINT. Data will be printed only for boxes that are currently running.

Syntax: PRINT BOXES P1

Where: P1 = One or more box numbers

Example:

```
PRINT BOXES 3 4
```

Print data for boxes 3 and 4.

BOX_PRINTER_SETTINGS

DEFAULT_PRINTER_SETTINGS

The appearance of printouts may be controlled by two macro commands. BOX_PRINTER_SETTINGS is equivalent to the menu selection "File | Print Format | Specific Boxes" and DEFAULT_PRINTER_SETTINGS is equivalent to "File | Print Format | Default Settings."

These commands differ only in that the former defines printing parameters for specific Boxes, while the latter defines default settings that will be in effect for all Boxes for the remainder of the session. The default settings are those that MED-PC uses when a Box is loaded. However, the default settings may be over-ridden for specific Boxes by subsequently executing BOX_PRINTER_SETTINGS from within a macro or by using the equivalent menu selection or equivalent MSN commands.

BOX_PRINTER_SETTINGS and DEFAULT_PRINTER_SETTINGS provide control over six aspects of printing:

1. PRINTWIDTH: the number of spaces occupied by each number on the printout. For example, a width of 12 would permit display of up to 11 digits plus a decimal point. The number of digits to the left and right of the decimal point are controlled by the PRINTDECIMALS parameter (below). The default PRINTWIDTH is 12.

2. PRINTDECIMALS: controls the number of digits displayed to the right of the decimal point. The default value of three specifies that three digits will be printed to the right of decimal points.

3. PRINTCOLUMNS: specifies the number of columns of data that will be printed on the printout. Default value is five.

4. PRINTPOINTS: specifies the point size of the characters. The default value is 12.

5. PORTRAIT or LANDSCAPE: determines whether the paper is oriented in Portrait (upright) or Landscape (sideways) mode. The default is PORTRAIT.

6. FORMFEEDS or NOFORMFEEDS: determines whether the output from each Box is printed on a fresh page. The default is NOFORMFEEDS, indicating that multiple data sets may be printed on the same page.

BOX_PRINTER_SETTINGS:

Syntax: BOX_PRINTER_SETTINGS PRINTWIDTH P1 PRINTDECIMALS P2 PRINTCOLUMNS P3 PRINTPOINTS P4 PORTRAIT NOFORMFEEDS BOXES P5

DEFAULT_PRINTER_SETTINGS:

Syntax: DEFAULT_PRINTER_SETTINGS PRINTWIDTH P1 PRINTDECIMALS P2 PRINTCOLUMNS P3 PRINTPOINTS P4 LANDSCAPE FORMFEEDS

Where: P1 =   The print width

                P2 =   The print decimals

                P3 =   The print columns

                P4 =   The print points

                P5 =   One or more Box numbers

Examples:

```
BOX_PRINTER_SETTINGS PRINTWIDTH 12 PRINTDECIMALS 3 PRINTCOLUMNS 5
PRINTPOINTS 12 PORTRAIT NOFORMFEEDS BOXES 1 2 3
```

Printouts for Boxes 1, 2 and 3 should contain 5 columns of numbers printed in 12-point characters. Each number should be printed in a field 12 characters wide, with 3 digits to the right of the decimal point. The paper will be in an upright orientation (Portrait) and a page will not be ejected following each Box.

Note: that BOX_PRINTER_SETTINGS will only effect currently loaded boxes.

```
DEFAULT_PRINTER_SETTINGS PRINTWIDTH 8 PRINTDECIMALS 2 PRINTCOLUMNS 10
PRINTPOINTS 10 LANDSCAPE FORMFEEDS
```

Default printouts should contain 10 columns of numbers printed in 10-point characters. Each number should be printed in a field 10 characters wide, with 2 digits to the right of the decimal point. The paper will be in sideways (Landscape) orientation and a page will be ejected after each Box is printed.

Note: that DEFAULT_PRINTER_SETTINGS will NOT affect currently loaded Boxes, but will alter the default settings for any subsequently loaded Boxes.

BOX_PRINTER_NAME

DEFAULT_PRINTER_NAME

The printer used to produce printouts may be selected by two macro commands. BOX_PRINTER_NAME is equivalent to the Printer dropdown box in the menu selection "File | Print Format | Specific Boxes," and DEFAULT_PRINTER_NAME corresponds to the Printer dropdown box in the menu selection "File | Print Format | Default Settings."

These commands differ only in that the former defines printing parameters for specific Boxes, while the latter defines default settings that will be in effect for all Boxes for the remainder of the session. The default settings are those that MED-PC uses when a Box is loaded. However, the default settings may be over-ridden for specific Boxes by subsequently executing BOX_PRINTER_NAME from within a macro or by using the equivalent menu or MSN commands.

BOX_PRINTER_NAME:

Syntax: BOX_PRINTER_NAME P1 BOXES P2

Where: P1 =   The name of an installed printer, surrounded by single quotes

                P2 = A list of box numbers

Example:

```
BOX_PRINTER_NAME 'HP LaserJet 6P' BOXES 1 2 3
```

Printouts for Boxes 1, 2 and 3 will be printed on the HP LaserJet 6P.

DEFAULT_PRINTER_NAME:

Syntax: DEFAULT_PRINTER_NAME P1

Where: P1 = The name of an installed printer, surrounded by single quotes

Example

```
DEFAULT_PRINTER_NAME 'Acrobat PDFWriter'
```

Send the printout to the Acrobat PDFWriter.

SET

SET is a macro command that may be used to change the value of variables, array elements, and named variables. This command is equivalent to the "Configure |Change Variables" menu command. SET may only be used to alter the values of variables in currently running boxes.

Syntax: SET P1 VALUE P2 MAINBOX P3 BOXES P4

Where: P1 = The variable or array element to alter

                   P2 = The value to which A should be set

                   P3 = The Box to be affected

                   P4 =   One or more additional boxes whose variables should be affected. The BOXES argument must not be omitted, even if the MAINBOX and BOXES arguments are identical.

Example 1:

```
SET X VALUE 10.1 MAINBOX 3 BOXES 3
```

Variable X is set to 10.1 in Box 3. Note that the BOXES argument must be specified even though MAINBOXES and BOXES are both set to 3.

Example 2:

```
SET D(29) VALUE 3 MAINBOX 1 BOXES 1,2,3
```

Array element D(29) is set to 3 in Boxes 1 through 3.

Example 3:

```
SET "FRSize" VALUE        10.000 MAINBOX 1 BOXES  1
```

Set named variable FRSize to 10 in Box 1. Note that the variable name must be in quotation marks.

DELAY

This menu option is available only while recording a macro. This option is used to insert a time delay into a macro so that the macro playback will pause for the specified time duration. The value must be specified in milliseconds. No delay will occur while the macro is being recorded.

Syntax: DELAY P1

Where: P1 = The number of milliseconds for which macro playback should be delayed.

Example:

```
Delay 1000
```

Pause the macro for 1 second.

Example of the Utility of Delays

This command can be useful when it is necessary to wait for a program to complete some action before the macro continues. For example, an MSN program might be written so that it immediately sets default values for variables. It would be convenient to be able to use a macro to load the program, allow the defaults to be set and then over-ride some of the values. Without a time delay between loading the program and over-riding the defaults, it would be possible to change the variables in the macro and to then have the MSN program change the values back to their defaults.

Consider the following MSN program:

```
\ FR Program

\ A = FR Size


S.S.1,
S1,
  0.01": SET A = 10 ---> S2

S2,
  A#R1: ON 1 ---> S3

S3,
  0.1": OFF 1 ---> S2
```

This program arranges a simple FR. Ten milliseconds after loading, "A" is set to 10. In S2, "A" responses on Input 1 turns on Output 1 (presumably connected to a pellet dispenser) and transitions to S3. S3 turns the Output off after 100 milliseconds and returns to S2 for another ratio run.

Now consider the following macro:

```
LOAD BOX  1 SUBJ 1 EXPT FR Demo GROUP 2 PROGRAM A2
```

```
SET A VALUE 20 MAINBOX 1 BOXES 1
```

If this macro is executed on a fast computer, it is possible that the Box will load and A will be set to 20 in less than 10 milliseconds. If that happens, then 10 milliseconds after the program loads, S1 of the program will set "A" back to 10.

The following macro avoids this problem by introducing a delay of 20 milliseconds after the box is loaded.

```
LOAD BOX  1 SUBJ 1 EXPT FR Demo GROUP 2 PROGRAM A2
DELAY 20
SET A VALUE 20 MAINBOX 1 BOXES 1
```

SENDING START, K-Pulses, and Responses to Boxes

A set of three macro commands provides equivalents to the menu commands for sending signals to boxes available in the dialog box accessible from "Configure | Signals."

START BOXES

Syntax: START BOXES

Optional Parameter: SYNCH

Example 1:

```
START BOXES 5 6 9
```

Send a START signal to Boxes 5, 6 and 9

Example 2:

```
START BOXES 1 3 SYNCH
```

Send a START signal to Boxes 1 and 3 and have them start on the same clock tick.

K

Syntax: K P1 BOXES P2

Where: P1 = A K signal

                P1 = One or more Box numbers

Optional Parameter: SYNCH

Example 1:

```
K 5 BOXES 1
```

Send K5 to Box 1.

Example 2:

```
K 10 BOXES 8 9 SYNCH
```

Send synchronized K10 to Boxes 8 and 9.

R

Syntax: R P1 BOXES P2

Where: P1 = A R signal

                P2 = One or more Box numbers

Optional Parameter: SYNCH

Example 1:

```
R 5 BOXES 1
```

Send R5 to Box 1.

Example 2:

```
R 10 BOXES 8 9 SYNCH
```

Send synchronized R10 to Boxes 8 and 9.

Synchronizing the Occurrence of Signals

By default, MED-PC staggers the occurrence of signals when more than one Box has been indicated in the "Boxes" panel. This feature promotes processing efficiency. However, it is sometimes necessary to simultaneously issue a signal to multiple Boxes. This need may arise, for example, when conducting yoked procedures. Signals may be issued simultaneously by checking the box labeled, "Synchronize Occurrence."

ON, OFF, LOCKON, LOCKOFF, and TIMED_OUTPUT

A set of five macro commands provide equivalents to the output controls available in the Dialog Box accessible from "Configure | Outputs."

* **ON** turns Outputs on.

        Syntax: ON P1 BOXES P2

        Where: P1 = An Output number

        P2 = One or more Box numbers

        Example: ON 3 BOXES 5 6 9 (Turn on Output 3 in Boxes 5, 6, and 9)

* **OFF** turns Outputs off.

        Syntax: OFF P1 BOXES P2

Where: P1 = An Output number

P2 = One or more Box numbers

Example: OFF 5 BOXES 1 (Turn off Output 5 in Box 1)

- **LOCKON** locks Outputs on.

    Syntax: LOCKON P1 BOXES P2

    Where: P1 = An Output number

    P2 = One or more Box numbers

    Example: LOCKON 3 BOXES 5 6 9 (Lock Output 3 on in Boxes 5, 6, and 9)

- **LOCKOFF** locks Outputs off.

    Syntax: LOCKOFF P1 BOXES P2

    Where: P1 = An Output number

    P2 = One or more Box numbers

    Example: LOCKOFF 3 BOXES 5 6 9 (Turn off Output 3 in Boxes 5, 6, and 9, without respect to whether the Output was turned on with LOCKON or ON)

- **TIMED_OUTPUT** turns an Output on for the specified time duration.

    Syntax: TIMED_OUTPUT P1 DURATION P2 BOXES P3

    Where: P1 = An Output number

    P2 = The number of seconds during which Output should be on

    P3 = One or more Box numbers

    Example: TIMED_OUTPUT 2 DURATION 2.50 BOXES 1 (Turn on Output 2 in Box 1 and turn it off 2.5 seconds later)

INPUTBOX, NUMERICINPUTBOX, and TEXTINPUTBOX

INPUTBOX is one of the most powerful macro commands. This command allows macros to ask the operator questions and then the results of those questions may be used as parameters for any other commands (not just the SET command used in the following example). For example, a macro could pause to ask the operator to enter the weight of a rat and then that weight could be automatically used later in the macro to set the value of a variable via the SET macro command.

Syntax: INPUTBOX P1 P2 P3 P4

Where: P1 =   A quoted string containing the title for the input box (displayed at the top of the dialog box)

P2 =   The text of the input prompt that appears in the middle of the dialog box

P3 =   A default value for the user's response -- the operator may modify this value or click OK to accept the default

P4 = A macro variable in which to store the response

Comments: There is no limit to the number of INPUTBOX commands that could be included in a macro, so it would be feasible to construct an extended interaction with the operator that could be used to set up a series of experimental sessions. Macro variables do not need to be declared -- use any desired word. However, it would be a good idea to prefix variable names with a symbol, such as "%", to make it easier to read and debug the macro. The value stored in the macro variable may be used in any subsequent line of the program, and it may be used more than once -- either in subsequent INPUTBOX commands or in other macro commands (such as SET or LOAD). The results stored in macro variables may be used in ANY macro command -- not just SET.

No validation is performed on the user's response; the user is allowed to enter anything or even leave the field blank. If the response will be used to set a variable value, it would be preferable to use NUMERICINPUTBOX (see below).

Consider the following macro:

```
LOAD BOX 1 SUBJ Rat 15 EXPT FR GROUP One PROGRAM Concurrent FR FI
INPUTBOX "Parameters for Rat 15" "Enter weight (grams)" "250" %Weight
SET A VALUE %WEIGHT MAINBOX 1 BOXES
```

The first line loads the Box (specifying that the Subject is Rat 15, among other parameters), and then the second line displays a Dialog Box that will wait until the operator enters the rat's weight.

Figure 13.5 – Dialog Box
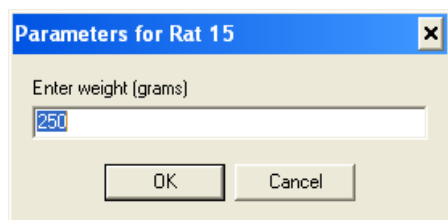
A key thing to notice about the INPUTBOX command above is that the last parameter is %Weight. %Weight is a variable that could have been named anything and is not displayed on the screen. However, the variable is used to store the value that was entered into the input field by the operator. It is this variable (%Weight) that is then used in the SET command as the value to be assigned to the MSN variable "A."

NUMERICINPUTBOX

The syntax of this command is identical to INPUTBOX, but the user is required to enter a number (that may include a decimal point). If the user leaves the input field blank or the field contains non-numeric information (such as letters), a message will be displayed and the user will be required to enter a different response. This command is preferred over INPUTBOX if the response will be used with a SET command to avoid attempting to set a program variable to something other than a number (which would cause an error).

TEXTINPUTBOX

The syntax of this command is identical to INPUTBOX, but the user is required to enter a response that contains something other than spaces. If the user leaves the input field blank or the field contains only spaces, a message will be displayed and the user will be required to enter a different response.

Input Box Editor

Entering the parameters for the INPUTBOX command (or the numeric or text variants) is greatly simplified by using the macro editor ("Macros | Editor"), which allows the user to construct an INPUTBOX command using a dynamic mockup of the input box, as illustrated in Figure 13.6:

Figure 13.6 – Input Box
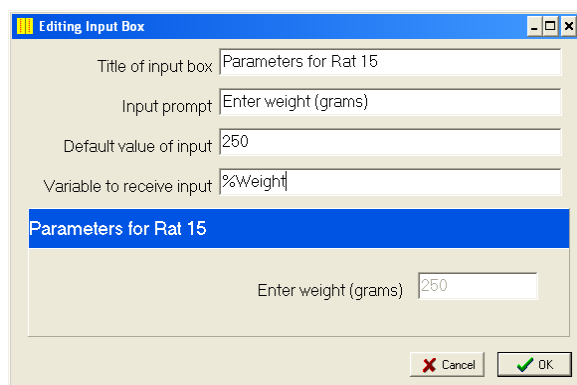
SHOWMESSAGE

This command is used to display messages during macro playback. When a SHOWMESSAGE command is encountered, the macro pauses and displays a dialog box with the text contained in the macro. The macro resumes execution after the OK button is clicked. This command is useful for prompting the user before some further macro action occurs.

Syntax: SHOWMESSAGE "P1"

Where: "P1" = A string of text enclosed in "" (e.g.. "Put Rat 5 in the chamber")

Comments: SHOWMESSAGE can only be added to macros via the macro editor -- there is no way to place a SHOWMESSAGE in a macro while recording the macro.
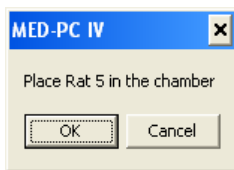
Example:

The following macro loads Rat 5 into Box 2 and then displays a Dialog Box. After the user clicks the OK button, the macro resumes and issues a start command.

```
LOAD BOX  2 SUBJ 5 EXPT Demo GROUP One PROGRAM Concurrent FR FI
SHOWMESSAGE "Place Rat 5 in the chamber"
START BOXES  2
```

Figure 13.7 is the show message produced by the macro:

Figure 13.7 – Show Message Produced by the Macro

PLAYMACRO

PLAYMACRO is an extremely useful macro command that provides for nested playback of macros.

Syntax: PLAYMACRO P1

Where: P1 = The file name of a macro, including the full path name.

Example:

```
PLAYMACRO C:\Mymacros\FR.mac
```

Play a macro named "C:\Mymacros\FR.mac."

A typical use of PLAYMACRO would be to write a separate macro for each rat in a squad of rats that are simultaneously run. A macro called "SQUAD1.MAC" could then be written that consists of a series of PLAYMACRO commands to execute the macros for each rat. For example, SQUAD1.MAC could contain the following commands to run the macros for rats 1-4:

```
PLAYMACRO C:\MEDPC IV\MACRO\RAT1
PLAYMACRO C:\MEDPC IV\MACRO\RAT2
PLAYMACRO C:\MEDPC IV\MACRO\RAT3
PLAYMACRO C:\MEDPC IV\MACRO\RAT4
```

MED-PC automatically records a completely qualified filename when a PLAYMACRO statement is recorded in a macro as the result of playing a macro while recording a macro. It is extremely important to remember to specify completely qualified filenames when creating or modifying macros with a text editor. MED-PC does assume that macros specified in a PLAYMACRO statement are located in the same directory as the macro containing the PLAYMACRO command.

There is no limit to the depth of nesting of macros. This means that it is acceptable for SQUAD1.MAC to execute RAT1.MAC.   RAT1.MAC could then execute a macro named LOADBOX1.MAC which could execute yet another macro. However, macro calls must not be recursive or circular. For example, SQUAD1.MAC must not attempt to play itself. A subtler situation occurs when two or more macros create a circular chain of macro calls. For example, if MACRO1.MAC contains a call to MACRO2.MAC, MACRO2.MAC must not attempt to play MACRO1.MAC. Recursive or circular macro calls are detected automatically by MED-PC and result in termination of macro execution. In addition, an error message is entered in the log.

EXIT_WHEN_DONE

The macro command EXIT_WHEN_DONE is equivalent to the menu command "File | Exit When All Boxes are Finished." The purpose of this feature is to cause MED-PC to shutdown automatically as soon as no Boxes are active.

Syntax: EXIT_WHEN_DONE P1

Where: P1 = ON or OFF

ON enables the feature and sets the checkmark on the corresponding menu item. OFF disables the feature and removes the corresponding checkmark from the menu.

LOG_OPTIONS

The display options for the session log may be set via the macro command LOG_OPTIONS. This command does not affect what is recorded in the log, but rather what is presently displayed in the log. In other words, the log will be nearly blank if all options are set to blank. However, the events are not lost -- all events for the entire session would be displayed if the options were later enabled.

Syntax: LOG_OPTIONS KEYBOARD P1 MACROS P2 ERRORS P3 BOXES P4

Where: P1 = ON or OFF to determine whether keyboard and mouse events are
                     displayed.

     P2 = ON or OFF to determine whether macro commands are displayed.

               P3 = ON or OFF to determine whether error messages are displayed.

               P4 = ON or OFF to determine whether events generated by MSN statements,
                     such as session termination commands (STOPABORT, STOPKILL and
                     STOPABORTFLUSH), are displayed.

Example:

```
LOG_OPTIONS KEYBOARD ON MACROS OFF ERRORS ON BOXES ON
```

The log will display all entries except for commands executed by macros.

RESET_ERRORS

RESET_ERRORS is the macro equivalent to the menu command "View | Reset Error Indicator." The error indicator appearing at the bottom of the screen may be reset under macro control by issuing the command, RESET_ERRORS. This command takes no arguments.

The error indicator consists of the words, "ERRORS! CHECK LOG!". The error indicator appears whenever an error is detected within MED-PC or within an MSN program. The error indicator will reappear after being reset in case of any subsequent errors. Errors may be viewed using the Session Log.

Syntax: RESET_ERRORS

Appendix C | Pascal

MED-PC allows for the inclusion of user-written Pascal routines directly in programs written in MSN. This is useful when performing complex online calculations that might otherwise be impractical to perform using standard MSN statements. There are two mechanisms for including Pascal routines in MSN programs.

The first mechanism is referred to as Inline Pascal and consists of placing Pascal source code directly in an MSN program. Inline Pascal code is executed as soon as it is encountered and is suitable for most tasks.

The second mechanism is known as a Background Procedure and consists of placing a reference to a Pascal procedure (subroutine) in an MSN program. Unlike Inline Pascal code, Background procedures are not executed immediately. Instead, they are executed whenever MED-PC has unused processing time available. Background procedures are suitable for very time-consuming, complex calculations or for tasks such as writing to the disk or printer that may take an indeterminate amount of time.

Inline Pascal Procedures

An interesting feature of MedState Notation is that it allows programmers fluent in Pascal to use "inline" Pascal. In other words, it is possible to include straight Pascal commands in the output section of a MedState Notation statement. This facility allows one to directly insert Pascal commands in a MedState Notation procedure. To place Pascal statements in an output section involves simply enclosing the Pascal statements with the ~ character. The following example would compute and set MSN variable A to the value of PI:

```
S.S.1,
S1,
  0.01": ~A := PI;~ ---> S2

S2,
  1": SHOW 1,PI,A ---> SX
```

For code too lengthy to place directly in the output statement, or code that will be used by several procedures, writing functions which can be called from MedState Notation procedures might be more useful. The appropriate place to put such functions would be inside the file USER.PAS. Several examples of user procedures are provided in USER.PAS. USER procedures are executed immediately upon being encountered in an output statement and hence compete with normal MSN statements for processing time.

Under earlier versions of MED-PC, there were important restrictions on inline procedures that reflected the fact that MED-PC was based on 16-bit technology and that certain Pascal functions were not re-entrant. In practical terms, this meant that some functions could not be called from inline Pascal without risking a system crash. This is not a concern under MED-PC, due to the 32-bit architecture of this version. Memory allocation routines and other functions that were not

re-entrant under earlier versions of Windows may now be safely employed within Inline Pascal procedures.

Writing INLINE Pascal Procedures

INLINE Pascal procedures consists of three parts:

1. A call to the procedure from within the MSN procedure.

2. The Pascal procedure, placed within the USER.PAS file.

3. The header for the procedure, placed between the INTERFACE and IMPLEMENTATION keywords appearing at the top of the USER.PAS file.

Below is a simple example of an INLINE Pascal procedure. The procedure simply adds A and B and places the result in C. The procedure call is like any Pascal procedure call and must be terminated by a semicolon. The entire procedure call must be enclosed by tildes (~). An extremely important detail is that the first parameter of the procedure call should always be MG. The reasons for this requirement are subtle and beyond the scope of the present discussion, but be sure to observe this requirement. Failure to do so could result in unpredictable program behavior and even system crashes. Note that 'A', 'B' and 'C' may be passed directly; one need not be concerned about how MED-PC differentiates between an 'A' belonging to Box 1 versus an 'A' belonging to Box 2.

1. Sample MSN procedure:

```
S.S.1,
S1,
  1": ADD A, B;
     ~ADDAB(MG, A, B, C);~ --->S2  \ Call ADDAB, Given MG and
                                    \ Variables A&B, Return with Value C

S2,
  1": SHOW 1,A=,A, 2,B=,B, 3,C=,C ---> S1
```

2. ADDAB would be placed anywhere in USER.PAS after the IMPLEMENTATION keyword. Note that MPCGlobal (of type MPCGlobalPtr) is the first parameter. Furthermore, all code of the procedure must be nested within the scope of MPCGlobal (i.e., the 'With MPCGlobal^ Do…End;' construct:

```
Procedure ADDAB(MPCGlobal: MPCGlobalPtr;
                Num1, Num2: Extended;
                Var Result: Extended);

Begin
  With MPCGlobal^ Do
  Begin
    Result := Num1 + Num2;
  End;  {With}
End;  {ADDAB}
```

3. A copy of the procedure declaration must be placed between the INTERFACE and IMPLEMENTATION keywords, as follows:

```
INTERFACE

Procedure ADDAB(MPCGlobal: MPCGlobalPtr;
                Num1, Num2: Extended;
                Var Result: Extended);

IMPLEMENTATION
```

Accessing Arrays By Passing Their Starting Address As Untyped VAR Parameters

INLINE procedures designed to manipulate arrays are somewhat more complicated to handle, particularly because arrays cannot be passed to the procedure in the same fashion as a simple variable. A convenient way to manipulate an array, however, is to pass it as an untyped variable parameter and declare an array to reside at the starting address of the variable. To utilize the ConstantVI procedure listed below, one would write MED-PC code similar to the following:

```
^Feeder = 1


LIST A = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,17,18,19,20


S.S.1,
S1,
  1": ~ConstantVI(MG, A, Round((Sizeof(A)/Sizeof(A[0]))), 30);~ ---> S2

S2,
  #START: RANDD B = A; SET B = B * 1" ---> S3

S3,
  B#T: ---> S4

S4,
  #R1: ON ^Feeder ---> S5

S5,
  0.1": OFF ^Feeder; RANDD B = A; SET B = B * 1" ---> S3
```

In the above example note that dummy data is placed in array A with the LIST command. This is done to declare an array to the desired dimension. DIM could not be used because RANND requires the array to be declared with the LIST. As soon as the USER call is executed the dummy values are replaced with appropriate values for the desired VI.

The format of this demo USER call is ConstantVI(MG, P1, P2, P3) where:

MG =   The literal string 'MG' is required as the first parameter.

P1 =   Name of array.

P2 =   Number of elements in array - in the example above, this number was calculated automatically.

P3 =   VI value in seconds - don't put " after P3, though.

Note also that in State 2, B is multiplied by 1" because ConstantVI returns a number, not a time value.

The key to manipulating array A is that X within ConstantVI will be set to the starting address of array A. Array Tn is then declared to start at the absolute starting address of variable X. The reason why Tn is declared with 1,000,500 elements is to ensure that ConstantVI will work with a MED-PC array of any dimension; this eliminates the need to alter ConstantVI for different MSN procedures. It is vital to not manipulate elements of Tn greater than the upper dimension of the MSN procedure's array A; elements of Tn beyond the upper bound of array A are memory locations assigned to other variables and arrays.

The Procedure ConstantVI is provided below in its entirety. It may also be found along with other procedures in the USER.PAS file supplied with the MED-PC software.

```
Procedure ConstantVI(MPCGlobal: MPCGlobalPtr;
                     Var X;
                     N: Integer;
                     VI: Extended);

Var
  I: Integer;
  Term1, Term2: Extended;
  Tn: Array [1..1000500] of Extended Absolute X;

Begin
  With MPCGlobal^ Do
  Begin
    Term1 := 1 + Ln(N);
    For I := 1 to N Do
    Begin
      If I < N Then
        Term2 := Ln(N-I)
      Else
        Term2 := 0;
      Tn[I] := (VI) * (Term1+((N-I)*Term2) - ((N-I+1)*Ln((N-I+1))));
    End;  {For}
  End;  {With}
End;  {ConstantVI}
```

Warning: Array Accesses Must be Within the Declared Range of the Array!

Under DOS releases of MED-PC, it was acceptable with user-written code to read from beyond the ends of an array without adverse consequences. Windows does not tolerate this and will issue GPFs (General Protection Faults). For this reason, it is essential to stay within array bounds, even when reading array elements. For example, assuming that the following declaration was made in an MSN program, DIM X = 10, one must not issue a statement such as, Y := BigArr[Vars['X'].StartOffset+11];

Accessing Arrays by Using DataRec

A very effective, universal method for accessing both variables and arrays is to pass the Box number of the calling Box as well as character variables corresponding to the names of the MED-PC variables and arrays to be manipulated. The ADDAB procedure call could be rewritten as, ADDAB(MG, BOX, A, B, C); where BOX is the MSN special identifier that reports the Box in which the procedure is running. The Pascal procedure would be rewritten as follows:

```
Procedure ADDAB(MPCGlobal: MPCGlobalPtr;
                Box: Byte;
                Var1, Var2, Var3: Char);

Var
  Data: ^DataRec;  { DataRec is compatible with }
                   { all MED-PC data structures }

Begin
  With MPCGlobal^ Do
  Begin
    Data := BoxPointer[Box];

    { The following With statement simplifies references }
    { to the data for the present Box                    }
    With Data^, Data^.Description, Data^.Header Do
    Begin
      { BigArr[Vars[Var3].StartOffset] accesses the      }
      { data associated with Var3.  In the example above }
      { above, Variable C will be referenced.  The 2nd   }
      { and 3rd terms reference Variables A and B.        }

      BigArr[Vars[Var3].StartOffset] := BigArr[Vars[Var1].StartOffset]
                                      + BigArr[Vars[Var2].StartOffset];
    End;  {With}
  End;  {With}
End;  {ADDAB}
```

In the preceding example, Data refers to the block of memory starting at the address held in BoxPointer. Elements of the BoxPointer array are automatically set to the starting address of the data structure for each active Box. In brief, Data maps onto the data associated with Box's data. The variables and arrays associated with the Box's data are stored in a contiguous block that may be thought of as one big array known as Data^.BigArr. The location of a given variable within Data^.BigArr is stored within Data^.Description.Vars in the form of a StartOffset from the beginning of Data^.BigArr. To manipulate a variable such as C, one manipulates the corresponding element of Data^.BigArr. Variable C, for example, is accessed as BigArr[Vars['C'].StartOffset].

Array elements may be accessed in an analogous fashion by adding the array element number to the StartOffset of the array. For example, to add array elements C[0] and C[1] and place the result in D[10], one could use the following code:

```
BigArr[Vars['D'].StartOffset+10] :=
    BigArr[Vars['C'].StartOffset+0] + BigArr[Vars['C'].StartOffset+1];
```

All MSN special identifiers, except S.S., may be referenced directly in USER procedures by their usual name, provided that the references occur within the scope of "With Data^, Data^.Description, Data^.Header Do", as illustrated above. For example, to change a Subject Number to the value of array element E(19), one could use the following code:

```
Procedure ChangeSNum(MPCGlobal: MPCGlobalPtr;
                     Box: Byte);

Var
  Data: ^DataRec;
  Temp: String;
  ReturnCode: Integer;

Begin
  With MPCGlobal^ Do
  Begin
    Data := BoxPointer[Box];
    With Data^, Data^.Description, Data^.Header Do
    Begin
      Str(BigArr[Vars['E'].StartOffset+19]:8:0, Temp);
      While Temp[1] = '' Do
      Begin
        Delete(Temp, 1, 1);
      End;  {While}
      StrPCopy(SubjectSt, Temp);
    End;  {With}
  End;  {With}
End;  {ChangeSNum}
```

Background Procedures

As mentioned at the beginning of this Appendix, there are a variety of Pascal functions that must not be called from within Pascal procedures or called from directly within an MSN procedure due to their lack of reentrancy. However, the MSN command BKGRND allows the use of these functions within MSN procedures.

The source code for background procedures must be placed in the file BACKPROC.PAS. This is a departure from the DOS version of MED-PC, in which background procedures were contained in USER.PAS. If porting from DOS, place only the body of the background procedures in the file BACKPROC.PAS. Do not copy the procedure headers.

In contrast to normal inline user Pascal code, procedures called via BKGRND do not execute as soon as they are called from within an MSN procedure. INLINE procedures execute completely when called, but BKGRND procedures execute in the background, where background is defined as time periods during which experimental events are not being serviced (as the result of the occurrence of timer interrupts). BKGRND procedures do not require "real-time" to execute, for they are executed entirely in the background. This means that irrespective of how complex or time-consuming the procedure is, it will not interfere with the processing of experimental events.

BKGRND is an output command that merely requests execution of a given user-written Pascal procedure. Unlike inline procedures, the exact time at which a BKGRND procedure is executed

is not predictable, but there are methods that may be employed to determine whether a BKGRND procedure has, in fact, executed; a technique for doing so is illustrated below.

BKGRND procedures are not called from MSN procedures via enclosure in tildes (~). Instead, one issues the BKGRND output command accompanied by a number from 1 to 10. The numeric parameter indicates which of up to 10 different user-written BKGRND procedures should be called. As soon as the BKGRND command is executed, a request is made to MED-PC's internal routines to process the BKGRND request as soon as some processing time is available. Under normal circumstances the BKGRND procedure will be executed within a few tens or hundreds of milliseconds, depending upon the processor on which MED-PC is executing, the number of active Boxes, etc. If MED-PC is operating near capacity, in the sense that the average cycle time (the "A" indicator on line one of the MED-PC runtime screen) approaches the resolution value declared during installation, then it may take a second or two to begin execution of the procedure.

Heavy disk activity could also slow the response time to BKGRND procedures. Due to the indeterminacy of the delay until a BKGRND request is processed, it is important to issue requests for BKGRND procedures only at times when it is possible to wait long enough for the request to be honored and the procedure executed. Convenient times might include the beginning and end of sessions, during long inter-trial intervals, time-outs between multiple schedule components, etc.

The example below demonstrates how to call and write a BKGRND procedure. It shows how to write a data file at the end of a session that contains information that will be read at the beginning of the next session. This example could serve as the basis for developing MSN code for "continuous" experimental sessions in which the events of one experimental session are used to define the contingencies for a subsequent session. In S.S.1, S1, BKGRND procedure 2 is requested and transition to S2 occurs. Note that transition from S2 will not occur until D equals 1. Examination of the Pascal code for BKGRND procedure 2 (below) shows that D is set to 1 after the data file is read. Thus, D indicates whether the BKGRND procedure has finished execution. Of course, any variable could be used for this purpose, but D was chosen for the present example. **In the present example, the data file will have the same name as the Subject running in the Box that called the BKGRND procedure plus a '.txt' extension and will be saved in the default data folder.**

(Example: MED-PC IV\DATA\animal 4500.txt).

The data file consists of the elements of array B. MED-PC automatically passes the Box number to a BKGRND procedure when it is called, but note that no other parameters are passed to the procedure. As a result, all parameters used by the procedure must be contained within MSN arrays and variables. Using "D" as a flag to indicate completion of the BKGRND procedure and reading data directly to and from "B" illustrates this aspect of utilizing BKGRND procedures.

Another mandatory aspect of employing BKGRND procedures is that the last line of the procedure MUST be "BackRequest[Box][1] := False;", where the "1" is the number of the BKGRND procedure; notice that both BKGRND procedures shown below include this line, except

that the second subscript varies as a function of the number of the BKGRND procedure. This statement is important, for it informs MED-PC that the procedure has finished executing for the requested box.

The MSN code below also illustrates writing a data file at the end of the session. Importantly, notice that transition to STOPABORT does not occur until "D" has been set to 1 (after being cleared before the request for BKGRND (1); this ensures that the procedure is executed before STOPABORT.

MSN Illustration using BKGRND Procedure calls:

```
DIM B = 2


\ DEMONSTRATE READING IN A FILE -- AS IN "CONTINUOUS" SESSIONS
S.S.1,
S1,     \ Request User-written Background Proc #2 -- read in
        \ data and session parameters presumably
        \ determined by last session's performance.
  0.1": BKGRND 2 ---> S2

S2,     \ Wait until the Background Proc sets D -- so that we
        \ know that the file has been read.
  0.1": IF D = 1 [] ---> S3

S3,     \ Show 'em for demo's sake
  0.1": SHOW 1,B(0)=,B(0), 2,B(1)=,B(1), 3,B(2)=,B(2) ---> S4

S4,     \ Simulate setting the values based on some pseudo behavior
  #K1: ADD B(1); SHOW 2,B(1)=,B(1) ---> SX
  #K2: ADD B(2); SHOW 3,B(2)=,B(2) ---> SX
  #K3: ---> S5  \ Let K3 simulate some basis for ending the session
  1": ADD B(0); SHOW 1,B(0)=,B(0) ---> SX

S5,
  0.1": SET D = 0; BKGRND 1 ---> S6

S6,     \ Don't go immediately into STOPABORT or STOPKILL
        \ Wait until background Proc says done via D = 1
  0.1": IF D = 1 [] ---> STOPKILL
```

The following is the code for BKGRND procedure 1. This procedure will generate an error if it cannot write to the data file.

```
{$F+} {BE SURE NOT TO REMOVE THE FAR DIRECTIVE TO THE LEFT!!}
{THIS PROC DEMONSTRATES WRITING TO A DATA FILE IN THE BACKGROUND}
Procedure BackProc1(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var
  Data: ^DataRec;
  St:   String;
  OutF: Text;
  I:    Word;

Begin
  With MPCGlobal^ Do
  Begin
    Data := BoxPointer[Box];
    With Data^, Data^.Description, Data^.Header Do
    Begin
      St := ExtractFilePath(ParamStr(0)) + 'Data\' +
                                 Trim(LowerCase(SubjectSt)) + '.txt';

      AssignFile(OutF, St);
      Rewrite(OutF);
      {
        Vars['B'].Size EXEMPLIFIES HOW TO DETERMINE THE NUMBER OF
        ELEMENTS ASSOCIATED WITH ANY ARRAY OR VARIABLE.  THE FIRST
        ELEMENT OF ARRAYS OR VARIABLES IS REFERENCED AS 0 AND THE SIZE
        FIELD GIVES THE HIGHEST SUBSCRIPT.  THE FOLLOWING LOOP WRITES
        ALL ELEMENTS OF ARRAY B TO THE OUTPUT FILE.
      }
      For I := 0 To Vars['B'].Size Do
        Writeln(OutF, BigArr[Vars['B'].StartOffset+I]:9:2);
      Close(OutF);

      {
        THE FOLLOWING LINE SET VARIABLE D SO THAT THE CALLING BOX CAN
        KNOW THAT THE JOB IS DONE.  ANY VARIABLE OR ARRAY ELEMENT COULD
        BE SUBSTITUTED.
      }
      BigArr[Vars['D'].StartOffset] := 1;  { SYNTAX FOR ADDRESSING A }
                                           {    SIMPLE VARIABLE      }
    End;  {With}

    I := IORESULT;  { REMOVES ANY ERROR CONDITIONS. }
                    { IMPORTANT IF DOING FILE I/O   }
    BackRequest[Box][1] := False;  { THIS MUST ALWAYS BE INCLUDED }
                                   { TO LET MED-PC KNOW THAT THE  }
                                   { JOB IS DONE.                 }
  End;  {With}
End;  {BackProc1}
{$F-}
```

The following is the code for BKGRND procedure 2. This procedure will not generate an error message if it cannot read from the data file.

```
{$F+}{$I-} {THE $I- DIRECTIVE PREVENTS THE PROGRAM FROM CRASHING }
          {IF THE FILE DOESN'T EXIST                           }
{THIS PROC DEMONSTRATES READING FROM A DATA FILE IN THE BACKGROUND}
Procedure BackProc2(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Var
  Data: ^DataRec;
  St:   String;
  InF:  Text;
  I:    Word;

Begin
  With MPCGlobal^ Do
  Begin
    Data := BoxPointer[Box];
    With Data^, Data^.Description, Data^.Header Do
    Begin
      St := ExtractFilePath(ParamStr(0)) + 'Data\' +
                                  Trim(LowerCase(SubjectSt)) + '.txt';

      AssignFile(InF, St);
      Reset(InF);
      If (IORESULT = 0) Then
      Begin
        {
          Vars['B'].Size EXEMPLIFIES HOW TO DETERMINE THE NUMBER OF
          ELEMENTS ASSOCIATED WITH ANY ARRAY OR VARIABLE.  THE FIRST
          ELEMENT OF ARRAYS OR VARIABLES IS REFERENCED AS 0 AND THE
          SIZE FIELD GIVES THE HIGHEST SUBSCRIPT.  THE FOLLOWING LOOP
          READS ALL ELEMENTS OF ARRAY C IN FROM THE INPUT FILE.
        }
        For I := 0 To Vars['B'].Size Do
          Readln(InF, BigArr[Vars['B'].StartOffset+I]);
        Close(InF);
      End;  {If}

      {
        THE FOLLOWING LINE SET VARIABLE D SO THAT THE CALLING BOX CAN
        KNOW THAT THE JOB IS DONE.  ANY VARIABLE OR ARRAY ELEMENT COULD
        BE SUBSTITUTED.
      }
      BigArr[Vars['D'].StartOffset] := 1;  { SYNTAX FOR ADDRESSING A }
                                           {    SIMPLE VARIABLE      }
    End;  {With}

    I := IORESULT;  { REMOVES ANY ERROR CONDITIONS. }
                    { IMPORTANT IF DOING FILE I/O    }
    BackRequest[Box][2] := False;  { THIS MUST ALWAYS BE INCLUDED }
                                   { TO LET MED-PC KNOW THAT THE  }
                                   { JOB IS DONE.                 }
  End;  {With}
End; {BackProc2}
{$F-}{$I+}
```

Compiling Background Procedures

BACKPROC.PAS is a dynamically linked library (DLL), rather than a Pascal unit. For this reason, it is necessary to explicitly (manually) recompile BACKPROC.PAS whenever changes are made. Additionally, the BACKPROC.PAS needs to be recompiled whenever MED-PC is updated (including minor version updates). This is only necessary if custom BACKPROC routines are being used; it is not necessary if BACKPROC.PAS is not being used. BACKPROC.PAS may be recompiled by opening Trans IV and compiling one of the programs written with the Build option.

Guidelines for writing and calling BKGRND procedures:

BKGRND procedures are called from the output section of MSN statements.

Syntax: BKGRND X

Where: 1 <= X <= 10

BKGRND procedures are not declared in the INTERFACE section of BACKPROC.PAS, but are placed in BACKPROC.PAS anywhere after IMPLEMENTATION.

BKGRND procedures must always be declared as follows:

Syntax: Procedure BackProc1(MPCGlobal: MPCGlobalPtr; Box: Byte); Export;

Where: BackProc1 corresponds to BKGRND 1, BackProc2 corresponds to BKGRND 2, etc.

The procedure header of a BKGRND procedure must be preceded by a far compiler directive {$F+}. Follow the procedure with {$F-}.

The last line of the procedure should contain:

        BackRequest[Box][1] := False;

where the numeral corresponds to the number of the BKGRND procedure. Note: if a BKGRND procedure appears to execute repeatedly, even though it is called only once from the MSN procedure, verify that the present statement is included and that it's second subscript is properly defined.

Do not remove or alter the initialization section of USER.PAS; this is the last block of code in USER.PAS prior to "END."

Multiple Boxes may simultaneously request the same BKGRND procedure without problems, for MED-PC will properly track which boxes have requested the procedure.

The same Box may have multiple simultaneous active requests for different BKGRND procedures, but note that a single Box may not request a BKGRND procedure a second time until its previous request has been completed.

Several BKGRND procedures have been included in BACKPROC.PAS, but users may feel free to replace them with their own code, subject to the limitations described.

Importing Inline & Background Pascal Code from Earlier MED-PC Versions

Recommended Steps

1. Read all documentation on Inline Pascal and Background Procedures. After doing so, the following instructions should make sense. The following references should be read:

    Appendix C:

    Inline Pascal Procedures

    Background Procedures

2. Do not try to edit an existing user.pas file for use in MED-PC. Instead, copy all inline routines to the USER.PAS supplied with MED-PC.

3. Add the 'MPCGlobal: MPCGlobalPtr' declaration to all procedure headers and add 'MG' to all calls to these procedures.

4. Verify that all array references are within the legal bounds of the array declarations.

5. Copy the CONTENTS of any user-written BACKGROUND procedures to the MED-PC supplied file 'BACKPROC.PAS.' Do not copy the procedure header declarations from earlier BACKPROC.PAS files. The present step is unnecessary if the user is not porting any background procedures.

    Unlike earlier versions of MED-PC, BACKPROC.PAS and USER.PAS are now automatically recompiled whenever any .MPC programs are translated. Explicitly recompiling USER.PAS and BACKPROC.PAS is no longer necessary.

6. Note that the fundamental data type for MED-PC variables and arrays is now the 10-Byte Extended, as opposed to the Real type used in earlier versions. The Extended type is a floating-point implementation, so functions used with earlier Reals generally work with Extended.

MED-PC's Internal Data Structures

This section describes the internal structure of the data structures associated with Boxes as they execute. It is not necessary for most MSN programmers to read or understand this section, but this information is provided for the benefit of sophisticated Pascal programmers. Every effort will be made to maintain compatibility with the following definitions across future releases of MED-PC, but no guarantees are offered.

Each compiled .MPC procedure has a unique nested record definition. Whenever a given .MPC procedure is loaded with the runtime load command, an element of the appropriate record is created with Pascal's NEW() procedure. Below is an example of the definition created by TRANS IV for a .MPC procedure named TEST1.MPC.

```
Type
  MPCProgramType = Record
    Description: DescriptionRec;
    Header: HeadRec;
    A,B,C,D,E,F,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z: Extended;
    G: Packed Array [0..10] of Extended;
  End;
  MPCProgramArrType = Packed Array [1..16] of ^MPCProgramType;

Var
  MPCProgramRec: MPCProgramArrType;
```

The following record definitions are found within MEDTYPES.DCU and are used to define data structures referenced in the data definitions of individual MSN procedures:

```
RecordRecPtr = ^RecordRec;
RecordRec = Record
  St: String20;   {String20 = String[20];}
  Next: Pointer;
End;

VarRec = Record
  VarKind, HasList: Byte;
  Size: Integer;
  StartOffset: Integer
End;

DescriptionRec = Record
  RecordSize: Longint;
  Vars: Packed Array ['A'..'Z'] of VarRec;
  PPrintoutDescription, DPrintoutDescription: TUserPrinterSettings;
  ThePrinter: Pointer;
  PPrintOptions, PPrintWidth, PPrintDecimals,
  PA_H, PI_P, PQ_X, PY_Z: Byte;
  DPrintOptions, DPrintWidth, DPrintDecimals,
  DA_H, DI_P, DQ_X, DY_Z: Byte;
  TimeDurs, Times, Ins: Integer;
  RecordH, RecordT: Packed Array [1..2] of RecordRecPtr;
  AddRecord: Byte;
  Y2KCompliant: Boolean;
  VarAliasList: TStringList;
  FlushPermission: Boolean;
End;
```

```
HeadRec = Record
  Source: String80;  {String80 = String[80];}
  StartMonth, StartDate, StartYear, EndMonth, EndDate, EndYear,
  SubjectNumber, ExpNumber, GroupNumber, BoxNumber,
  StartHours, StartMinutes, StartSeconds, EndHours,
  EndMinutes, EndSeconds, Y2KStartYear: Real;
  SubjectSt, ExpSt, GroupSt: ShortString;
  TheLibHandle: THandle;
  FileName: ShortString;
  Comments: ShortString;
End;
```

The procedure has one array, "G," declared by DIM G = 10. Each .MPC procedure has an array of pointers to data structures. TEST1Rec is an array of pointers to a type of data structure appropriate to TEST1.MPC. The type of the array is MPCProgramArrType, which consists of an array of 16 pointers to MPCProgramType. Each element of MPCProgramArrType is of MPCProgramType and contains all of the data associated with a box running TEST1.MPC.

If TEST1.MPC were loaded into Box 3, Element 3 of MPCProgramRec would be assigned memory via NEW. Within the MPCProgramType are two nested records, Description and Header, which contain definitions of all simple MED-PC variables (within "A".."Z") and all MED-PC arrays (within "A".."Z"). The order in which these records, variables, and arrays are declared is always in a constant order:

1. Description

2. Header

3. Simple variables, in alphabetical order

4. Arrays, in alphabetical order

The two records, Description and Header, are defined within MEDTYPES.DCU. Description contains miscellaneous information about the .MPC procedure and the Box running the procedure. Its fields have the following functions:

RecordSize is set to the number of bytes occupied by a given Box's data structure after the Box's data structure has been assigned memory by NEW. It is used primarily to free the memory occupied by a Box's data after the Box is unloaded. In the example, it is the size of an element of TEST1ArrType.

VARS is an array that describes the way in which each MED-PC letter is utilized. It is especially useful to understand the use of this array when writing INLINE or BKGRND Pascal procedures. The elements of VARS are indexed by characters 'A'..'Z.' For example, information about Box 3's use of 'C' is contained in TEST1Rec[3]^.Description.VARS['C'] (assuming Box 3 is loaded with TEST1.MPC). Within VARS, the following fields are defined:

VarKind: 0 indicates that the letter is a simple variable. 1 denotes an array.

HasList: 1 indicates that the array was declared via the LIST command.

Size: This contains the total number of data elements occupied by the data structure. For a simple variable, Size always equals 1. For arrays, Size always equals the largest subscript plus 1. Array G, dimensioned internally as 0..10 as the result of the MED-PC statement DIM G = 10, has a Size of 11.

StartOffset: Because the order in which Description, Header, the simple variables and the arrays is declared is always constant, the data associated with the simple variables and arrays is always located in a fixed location in memory, relative to the starting address of the overall data structure. This allows the relative location of each variable and array to be recorded in StartOffset, where a StartOffset of 0 indicates that the variable or array starts immediately after Header. A StartOffset of 1 indicates that the variable or array starts in the second element of the data area (6 bytes after the start). Note that this value is expressed in terms of data elements, not bytes.

In TEST1Rec, "A" has a StartOffset of 0, B has a StartOffset of 1, etc. Because "G" begins immediately after the simple variables, its StartOffset is 25. Of course, if another array started after "G", its StartOffset would not be 26, but rather 36. The use of StartOffset is considerably simpler than it sounds and is useful in USER code primarily in conjunction with the DataRec structure defined in a later section.

PPrintoutDescription and **DPrintoutDescription**: These variables control the appearance of printouts and disk files. These variables contain data related to the choices made in the printer configuration dialog and may also be set by MSN program options. The structure of these records follows:

```
Type
  PuserPrinterSettings = ^TUserPrinterSettings;
  TuserPrinterSettings = Record
    FWidth: Integer;
    Decimals: Integer;
    Columns: Integer;
    Points: Integer;
    Orientation: Integer;
    CondensedHeaders: Integer;
    FormFeeds: Integer;
    FontName,
    PrinterName: String;
  End;
```

PPrintOptions: This variable encodes the printer options selected by entering the PRINTOPTIONS command prior to the first state set. Each bit reflects the presence or absence of a particular printer option.

PPrintWidth: This is the total field width, including decimal point and decimal digits, in which data will be printed on the printer. This variable is affected by the PRINTFORMAT command.

PPrintDecimals: The number of digits to be printed to the right of the decimal point on printouts. This variable is affected by the PRINTFORMAT command.

PA_H: Each bit of this variable encodes whether a given MED-PC variable or array in the range 'A' through 'H' will be printed. The value of this variable is 255 by default, but is altered by the PRINTVARS command. The value of 'A' will be printed when the low-order bit is set to 1; similarly, the value of "H" is printed when the high-order bit is set to 1. For example, a value of 131 (binary 10000011) would print 'A', 'B' and 'H.'

PI_P, **PQ_X**, and **PY_Z**: These variables are analogous to PA_H.

DPrintOptions, **DA_H**, **DI_P**, **DQ_X**, and **DY_Z**: These variables are analogous to PPrintOptions PA_H through PY_Z except that they control the internal structure of disk files.

TimeDurs, **Times**, **Ins**, **RecordH**, **RecordT**, and **AddRecord**: These variables are presently not used and may be dropped in subsequent releases.

Y2KCOMPLIANT is set to true if the Y2KCOMPLIANT directive is present before the first State Set of the MSN program. By default, this variable is set to false.

VarAliasList: This is a TStringList that contains the names of all variable aliases and the variables or array elements to which they refer.

FlushPermission: Used internally to indicate whether the data may be written to disk automatically, versus requiring user intervention.

The following fields comprise the Description record.

Source: This string contains the name of the .MPC procedure running in the current Box.

StartMonth, **StartDate**, **StartYear**, **EndMonth**, **EndDate**, **EndYear**, **SubjectNumber**, **ExpNumber**, **GroupNumber**, **BoxNumber**, **StartHours**, **StartMinutes**, **StartSeconds**, **EndHours**, **EndMinutes**, and **EndSeconds**: These variables comprise the special identifiers described in detail in Appendix A.

DataRec - Universal Access to a Box's Data

The Definition of DataRec

```
DataRec = Record
  Description: DescriptionRec;
  Header: HeadRec;
  BigArr: Array [1..1000500] of Extended;
End;
```

Explanation of DataRec

DataRec is a record definition that is not directly involved in defining the data structure for .MPC procedures, but is compatible with all possible data definitions for MSN procedures. The fields of the nested Description and Header records provide access to all information about printing and disk options, variable definitions, etc., as described above. BigArr is an array that starts at the same address as the data for a given Box. A Box's variables and array elements may be located and accessed within BigArr by using the information contained within VARS.

The primary purpose of DataRec is to provide a universal mechanism for accessing the data associated with a Box. The key to this approach is that the record is completely self-documenting in the sense that the fields of VARS provides complete information about whether a letter is a variable or an array (via the VarKind field) and the number of elements associated with the letter. To access data in USER procedures by way of DataRec, one must pass the Box's number to the Pascal procedure contained in the USER.PAS, for example:

```
Procedure Demo(MPCGlobal: MPCGlobalPtr; Box: Byte);
```

Next, a variable that points to a structure of type DataRec should be declared:

```
Var
  Data: ^DataRec;

Begin
  With MPCGlobal^ Do
  Begin
```

Within the body of the procedure, Data should be set to point to the data associated with the Box so that the Box's data may be accessed. The address of the Box's data is stored by MED-PC within BoxPointer.

```
        Data := BoxPointer[Box];
```

Accessing the data within Data^ is simplified by using Pascal's WITH keyword to partially de-reference Data^. The following WITH block allows all fields within the Description and Header records to be accessed without the prefix "DATA^." Additionally, BigArr may be referenced without the prefix "DATA^."

```
        With Data^, Data^.Description, Data^.Header Do
        Begin
          {User Pascal code goes here}
        End;  {With}
    End;  {With}
End;  {Demo}
```

An example, using DataRec was provided in the Inline Pascal Procedures Section.

Appendix D | Contact Information

Please contact MED Associates, Inc. for information regarding any of our products.

Visit our website at www.med-associates.com for contact information.

For technical questions, email support@med-associates.com.

INDEX