med associates inc

instrumentation and software for research

MED-PC®

SOF-736/SOF-737

PROGRAMMER'S MANUAL

DOC-301
Rev. 1.3

MED Associates, Inc.
P.O. Box 319
St. Albans, Vermont 05478

Phone: 802.527.2343
Fax: 802.527.5095
www.med-associates.com

Notes

Table of Contents

Notes

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 work through 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 the Tutorials at the end of each chapter to brush up on MedState Notation.

The experienced user may use Appendix A to review the syntax for various commands.

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 - S.S.#,

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 Notation, each State Set is represented by the **S.S.#,** heading, where # is a unique 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, named constants, or variables. The periods following each "S" and the comma following the number are required.

States - S#,

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. Note that there is no period in the State Label, but a comma does follow the number.

Statements - General Description

States are comprised of commands called Statements. 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 transition arrow "--->", and the transition is to the right of the arrow. The transition arrow must be formed by three dashes followed by the greater than symbol "--->".

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 ---> S2
```

Which may be read: "IF a response on Input 1 is received, THEN add one to the variable 'A,' and GOTO State 2."

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 executed code. 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 Windows path separator, **<u>not</u>** 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)
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 Named Constants and in numbering State Sets and States.

Named Constants

Named Constants are a convenient means of substituting words for frequently used integers. Named 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 "^Hopper" follows

```
^Hopper = 3


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

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

Named Constants must be declared as having an integer value. For example, **^Hopper = 3.1** is illegal, as Named Constants can only be represented as integer (whole) numbers. It is also illegal to attempt to assign a value to a Named Constant during program execution. For example

```
S.S.1,
S1,
  Input: SET ^Hopper = 6 ---> S2
```

is illegal as it attempts to assign the value of 6 to the Named Constant ^Hopper after the program has already started.

Up to 2000 Named Constants may be declared in a single procedure and Named Constants are restricted to holding values within the range of -2,147,483,647 (-2^31) to 9,223,372,036,854,775,807 (2^63).

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

Variables

MedState Notation programs automatically have 26 variables available. The variables are named according to the alphabet. By default, the letters "A" through "Z" are automatically defined as "Simple" variables that may hold a single value.

Simple variables do not need to be explicitly declared - you may simply use the variable without first declaring it. For example, the following program increments the value of variable A by 1 every time a response on input 1 is received:

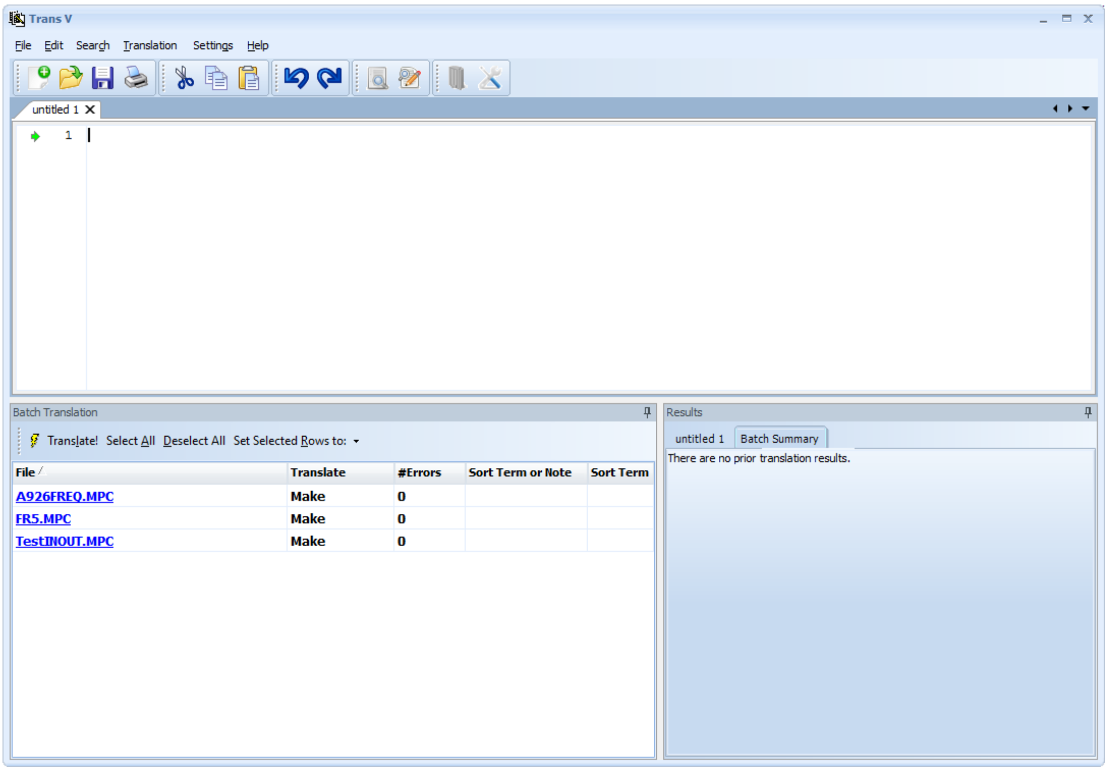
```
S.S.1,
S1,
  #R1: ADD A ---> S1
```

Tutorial 1: Writing the first program

Now that the basic terminology and concepts for writing a simple program have been covered, the next step is to replace the vague notions of "INPUT," OUTPUT," and "NEXT" from the previous sections with concrete examples of code. This exercise will cover how to arrange State Sets, States, and Statements, as well as teach the proper use of Statements, Named Constants, Inputs, and Outputs. For this example, assume that the operant chamber is equipped with a House Light connected to output 7, although any output device can be used.

The first step is to open TRANS by using the shortcut that was placed on the desktop when MED-PC was installed. Once TRANS opens, select **File**| **New File**[1].

Figure 2.1 - Open Trans


Trans V
File Edit Search Translation Settings Help
untitled 1
Batch Translation
Translate! Select All Deselect All Set Selected Rows to:
Results
untitled 1 Batch Summary

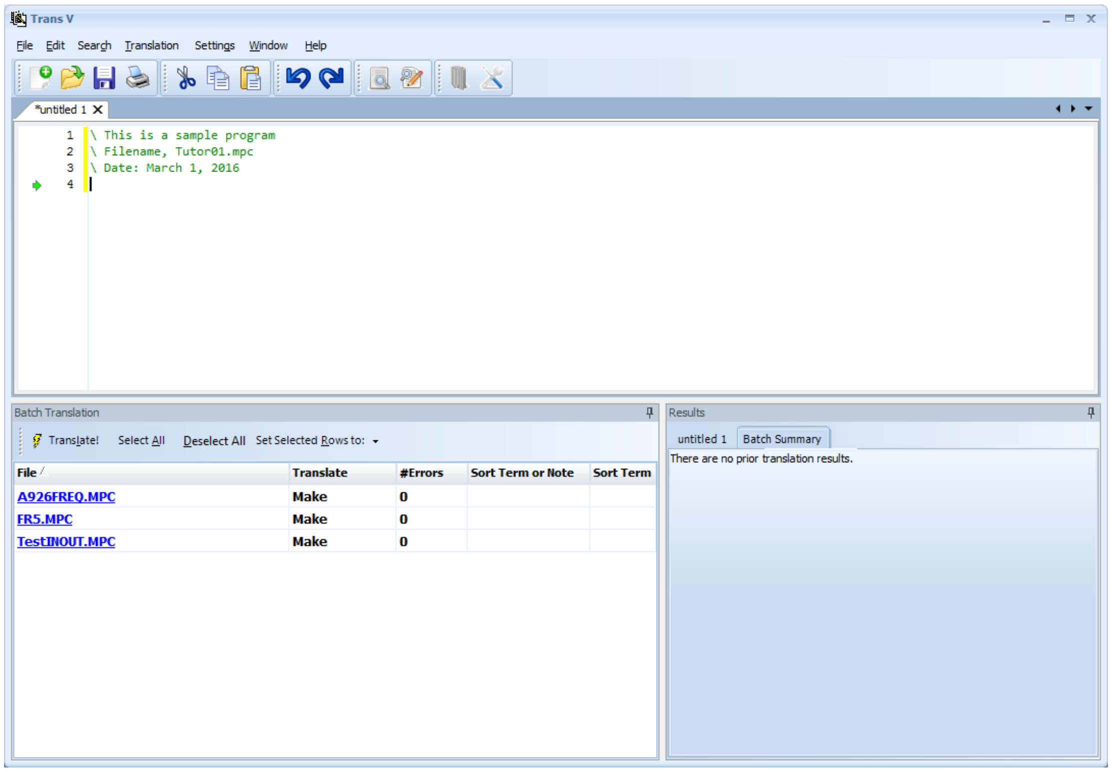
File	Translate	#Errors	Sort Term or Note	Sort Term
A926FREQ.MPC	Make	0		
FR5.MPC	Make	0		
TestINOUT.MPC	Make	0		

There are no prior translation results.

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, the text must be saved as unformatted ASCII or DOS text and it must be saved with the extension *.MPC (where * is the filename.).

Next, enter the comments. The comment section is a very important part of the program. A description of the protocol that the program runs, variables that can be changed, data that the program collects, and often a history of changes that have been made to the program and why those changes were made can all be kept in the comment section.

Figure 2.2 - Enter Comments


Trans V
File Edit Search Translation Settings Window Help
*untitled 1
Batch Translation
Translate! Select All Deselect All Set Selected Rows to:
Results
untitled 1 Batch Summary

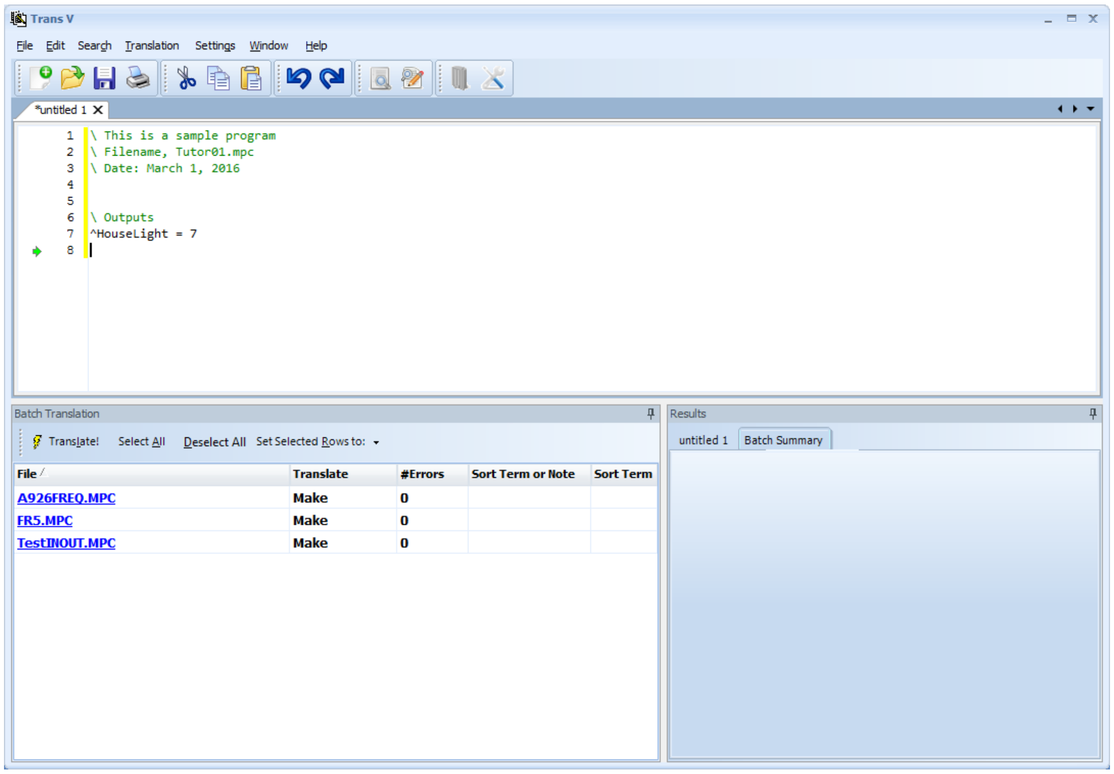
```
\ This is a sample program
\ Filename, Tutor01.mpc
\ Date: March 1, 2016
```

File	Translate	#Errors	Sort Term or Note	Sort Term
A926FREQ.MPC	Make	0		
FR5.MPC	Make	0		
TestINOUT.MPC	Make	0		

There are no prior translation results.

Next, define the Named Constants. This is a simple program with only one input, time, and one output, the House Light. Therefore, define one Named Constant for the House Light and call it "^HouseLight." In most hardware configurations the House Light is connected to output 7.

Figure 2.3 - Define Named Constants

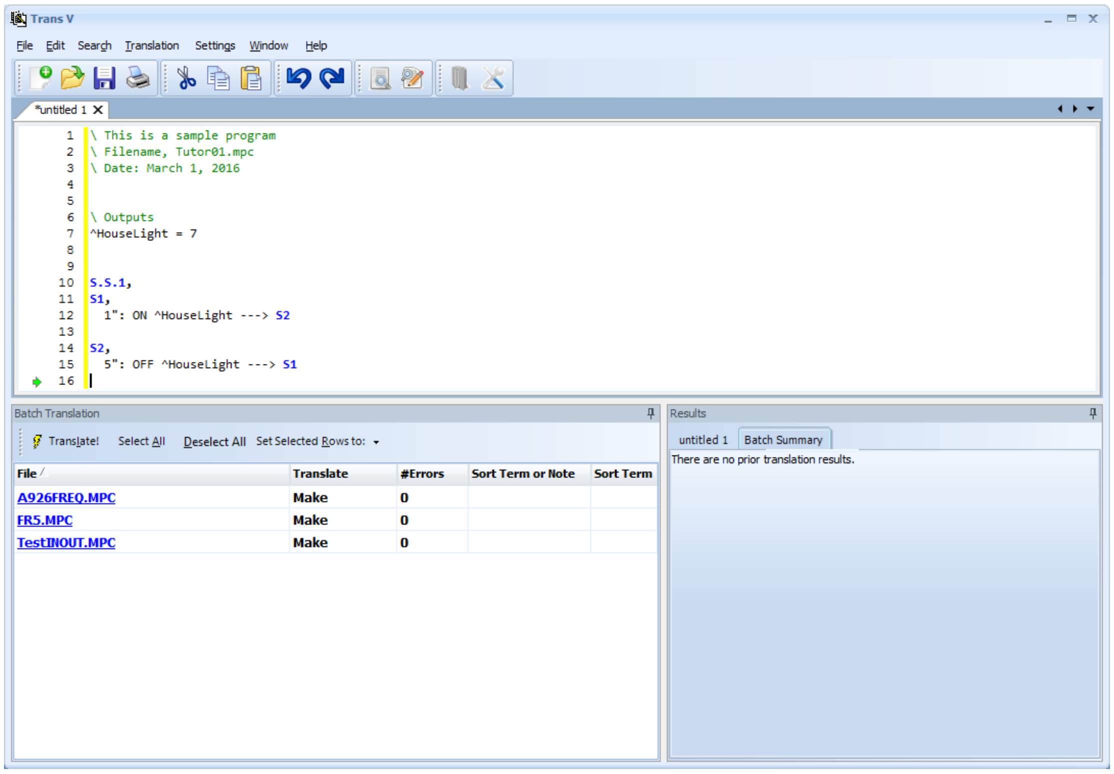

Trans V
File Edit Search Translation Settings Window Help
*untitled 1
Batch Translation
Translate! Select All Deselect All Set Selected Rows to:
Results
untitled 1 Batch Summary

```
\ This is a sample program
\ Filename, Tutor01.mpc
\ Date: March 1, 2016


\ Outputs
^HouseLight = 7
```

File	Translate	#Errors	Sort Term or Note	Sort Term
A926FREQ.MPC	Make	0		
FR5.MPC	Make	0		
TestINOUT.MPC	Make	0		

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


Trans V
File Edit Search Translation Settings Window Help
*untitled 1

```
\ This is a sample program
\ Filename, Tutor01.mpc
\ Date: March 1, 2016


\ Outputs
^HouseLight = 7


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

S2,
  5": OFF ^HouseLight ---> S1

```

Batch Translation
Translate! Select All Deselect All Set Selected Rows to:

File	Translate	#Errors	Sort Term or Note	Sort Term
A926FREQ.MPC	Make	0		
FR5.MPC	Make	0		
TestINOUT.MPC	Make	0		

Results
untitled 1
Batch Summary

There are no prior translation results.

Tutorial 1: Let's Review

This section is the program "Header." It is created by using comments.

```
\ This is a sample program
\ Filename, Tutor01.mpc
\ Date: March 1, 2022
```

This section declares the Named Constants used to define outputs appearing in this program.

```
\ Outputs
^HouseLight = 7
```

This program uses one State Set (**S.S.1,**) and two States (**S1,** and **S2,**).

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

S2,
  5": OFF ^HouseLight ---> S1
```

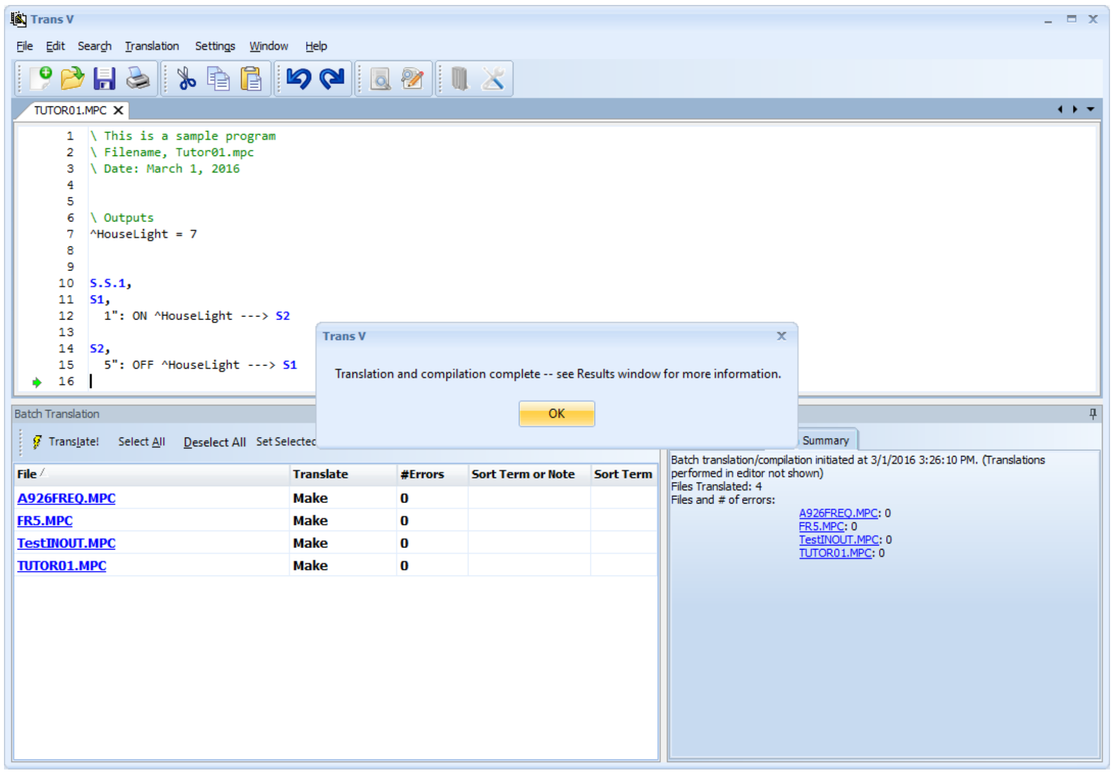
A useful exercise is to learn how to read through the programs in a narrative fashion. This program consists of a single State Set with two States. This program could be read by saying:

> In State 1, wait one (1) second, turn on the House Light, then goto State 2.
>
> In State 2, wait five (5) seconds, turn off the House Light, and go back to State 1.

It is a good practice to get into the habit of narrating through the programs in this manner.

Save this program as Tutor01.mpc in the default directory[2]. Then click **Translation | Batch Translate!** Trans will automatically compile any programs that are out of date and display a message box showing that the compilation was completed successfully. See Figure 2.5.

Figure 2.5 - Translation Completed Successfully


Trans V
File Edit Search Translation Settings Window Help
TUTOR01.MPC
Batch Translation
Translate! Select All Deselect All Set Selected
Summary
OK

```
\ This is a sample program
\ Filename, Tutor01.mpc
\ Date: March 1, 2016


\ Outputs
^HouseLight = 7


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

S2,
  5": OFF ^HouseLight ---> S1
```

Trans V

Translation and compilation complete -- see Results window for more information.

File	Translate	#Errors	Sort Term or Note	Sort Term
A926FREQ.MPC	Make	0		
FR5.MPC	Make	0		
TestINOUT.MPC	Make	0		
TUTOR01.MPC	Make	0		

```
Batch translation/compilation initiated at 3/1/2016 3:26:10 PM. (Translations
performed in editor not shown)
Files Translated: 4
Files and # of errors:
                    A926FREQ.MPC: 0
                    FR5.MPC: 0
                    TestINOUT.MPC: 0
                    TUTOR01.MPC: 0
```

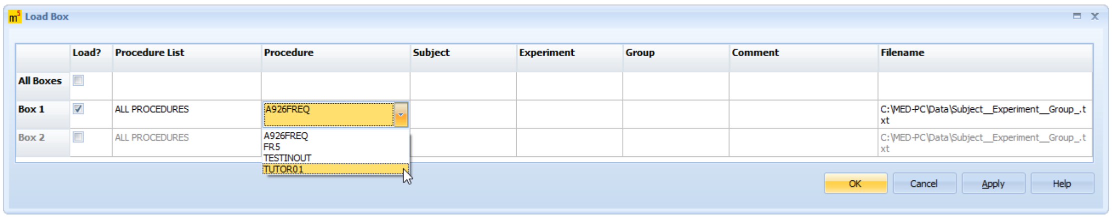
If there are any errors debug the program to resolve the issues.

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 for **Translation | Batch Translate!**.

Once the program has been successfully translated and complied, open MED-PC.

Once MED-PC is opened, select **Sessions** | **Load Box...** click the checkbox next to Box 1 and then select Tutor01.mpc from the Procedure pull down menu. Click **OK** to open.

Figure 2.6 - Load Box



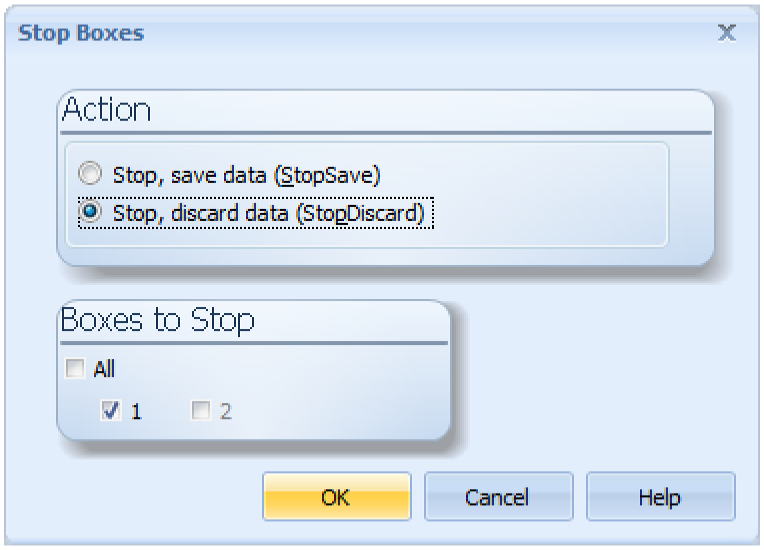
Load Box

	Load?	Procedure List	Procedure	Subject	Experiment	Group	Comment	Filename
All Boxes	☐							
Box 1	✓	ALL PROCEDURES	A926FREQ					C:\MED-PC\Data\Subject__Experiment__Group_.txt
Box 2	☐	ALL PROCEDURES						C:\MED-PC\Data\Subject__Experiment__Group_.txt

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 **Sessions | Stop Box...** The screen shown below will appear.

Figure 2.7 - Close Experimental Sessions Menu



Stop Boxes

Action

( ) Stop, save data (StopSave)
(●) Stop, discard data (StopDiscard)

Boxes to Stop

☐ All
✓ 1 ☐ 2

Select the checkbox next to **Box 1** and the **Stop, discard data (StopDiscard)** radio button. Note that the light turning on and off continues when in this window. This is because the STOPDISCARD 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 as an input, what may be of more interest to researchers is the input from a test subject 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 subject 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.

Syntax: `P1#RP2: OUTPUT ---> NEXT`

Where: P1 = The number of responses that must occur in order to progress to the output section of the statement

P2 = The logical input number of the device that is collecting inputs

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."

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 P1 ---> NEXT`

Where: P1 = 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.

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 subject 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 subject is being punished for responding during the ITI.

Multiple Input Statements

The two examples above also serve to illustrate that you can have more than one input statement in a State. When MED-PC processes a State it goes from the top to the bottom. It checks the first input and sees if its criteria has been met. If it has, then it does whatever commands are in the output section and arrows to whatever is next. If the first input criteria has not been met, then it will check the next input statement to see if its criteria has been met. It will do this until it finds an input criteria that has been met or it runs out of inputs. Whichever comes first.

Multiple Commands in Output Section

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

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

You can string together multiple commands in this fashion using a semicolon between each command.

SHOW

In the program at the end of Chapter 2 the only way to know that the program was running was by watching the light blink on and off or by looking at the status in the Box Status Panel. The SHOW command can display information to the screen that indicates to the user that a MED-PC program is running. The Show Command Output Panel of the screen may be used to display data for each active Box in any of the 200 available show positions (numbered 1 - 200).

Syntax: `INPUT: SHOW P1,Label,P2 ---> NEXT`

Where: P1 = Position in the Show Command Output window where the information is to be displayed. Must be a whole number in the range 1...200

Label = A user defined name for that position

P2 = Number, variable, Named Constant, array element, or mathematical expression to be displayed in position P1

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 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.

SHOWEX

SHOWEX is an extended version of SHOW that adds a fourth parameter providing the ability to control the number of digits displayed after the decimal point.

Syntax: `INPUT: SHOWEX P1,Label,P2,P3 ---> NEXT`

Where: P1 = Position in the Show Command Output window where the information is to be displayed. Must be a whole number in the range 1...200

Label = A user defined name for that position

P2 = Number, variable, Named Constant, array element, or mathematical expression to be displayed in position P1

P3 = Number of digits of precision to display to the right of the decimal point. MED-PC will display up to 8 digits after the decimal point. This value may be either a number, variable, Named Constant, or array element.

Example 1:

```
1": SHOWEX 1,Two,3.14159265,2 ---> SX
```

When loaded into MED-PC this example will display the following:

Box 1:	Two	3.14

This example shows what happens when only two digits of precision is requested.

Example 2:

```
   1": SHOWEX 1,Four,3.14159265,4 ---> SX
```

When loaded into MED-PC this example will display the following:

Box 1:	Four	3.1416

This example shows what happens when four digits of precision is requested. Notice that the final digit 5 was rounded up to 6.

Example 3:

```
   1": SHOWEX 1,Zero,3.14159265,0 ---> SX
```

When loaded into MED-PC this example will display the following:

Box 1:	Zero	3

This example shows what happens when no digits are requested after the decimal point.

Example 4:

```
   1": SHOWEX 1,Eight,3.14159265,8 ---> SX
```

When loaded into MED-PC this example will display the following:

Box 1:	Eight	3.14159265

This example shows that MED-PC can display up to eight digits of precision after the decimal point.

All of the above examples could have been done with one SHOWEX statement by stringing the parameters together:

```
   1": SHOWEX 1,Two,3.14159265,2,  2,Four,3.14159265,4,
              3,Zero,3.14159265,0, 4,Eight,3.14159265,8 ---> SX
```

When loaded into MED-PC this example will display the following:

Box 1:	Two	3.14	Four	3.1416	Zero	3	Eight	3.14159265

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 TRANS and type in the following (changes from Tutor01 noted in bold):

```
\ This is a sample program
\ Filename, Tutor02.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^HouseLight = 7


\ Variables
\  L = Left Lever Response Counter


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

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: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^HouseLight = 7


\ Variables
\  L = Left Lever Response Counter


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

S2,
  #R^LeftLever: ADD L; SHOW 1,Left Responses,L ---> 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.

As soon as the program is loaded the House Light will turn on immediately. Press the Left Lever and notice that the Left Response counter increments on the screen.

Do not be alarmed if the Left Response counter is not going up sequentially (..., 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 began running immediately when they were loaded into the Box. However, the experiment may require that 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 always desirable. This chapter will deal with this problem, as well as how to stop the program without having to issue the **Sessions | Stop Box...** command.

#START

The #START command gives the experimenter 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 ^HouseLight ---> 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."

Exactly how to issue a START command is explained in Tutorial 3.

STOPDISCARD

Note: Users upgrading from MED-PC IV might remember the STOPKILL command. This command has been deprecated. STOPKILL has been replaced with the command STOPDISCARD. Programs that contain the old command will still continue to compile and run, however, it is strongly recommended that the new STOPDISCARD command is used instead.

The STOPDISCARD command is a special transition that causes the program that is running to immediately stop executing. Any outputs currently turned on, but not "LOCKED ON" (see Appendix A), 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. When the STOPDISCARD command is executed any and all data is wiped from memory and cannot be recovered.

Syntax: `INPUT: OUTPUT ---> STOPDISCARD`

Real code may look like this:

```
S12,
  2': ---> STOPDISCARD
```

Where after 2 minutes, the program will stop, the Box's status will be cleared, and all the data in memory will be wiped clean.

STOPSAVE

Note: Users upgrading from MED-PC IV might remember the STOPABORT and STOPABORTFLUSH commands. These commands have been deprecated. STOPABORT and STOPABORTFLUSH have both been replaced with the command STOPSAVE. STOPABORT has no direct equivalent and is now interpreted as STOPSAVE. Programs that contain the old commands will still continue to compile and run, however, it is strongly recommended that the new STOPSAVE command is used instead.

Like STOPDISCARD, STOPSAVE is a special transition that turns off all outputs that are not locked on and stops procedure execution. The big difference is that STOPSAVE will save the data to the hard drive before wiping the memory clean.

Syntax: `INPUT: OUTPUT ---> STOPSAVE`

Real code may look like this:

```
S2,
  60': ---> STOPSAVE
```

Where after 60 minutes, the program will stop, the Box's status will be cleared, the data will be saved to the hard disk, and then the data in memory will be wiped clean.

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 and have the program stop on its own after a period of one minute.

Open TRANS and type in the following:

```
\ This is a sample program
\ Filename, Tutor03.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^HouseLight = 7


\ Variables
\  L = Left Lever Response Counter


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

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

To prevent the program from beginning as soon as it is loaded, the START command was added under State Set 1, State 1.

Finally add a new State Set so the program will stop on its own after one minute by adding the following code at the bottom of the new program:

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

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

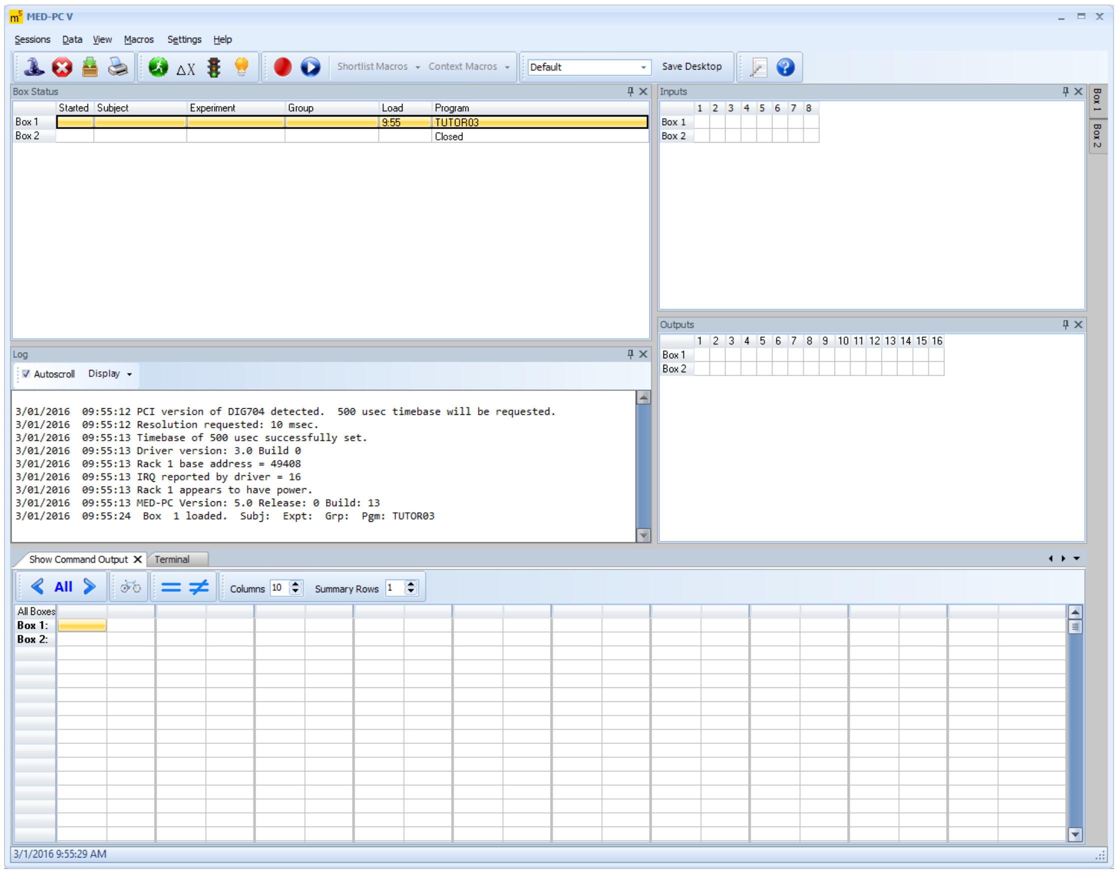
The final product should look like this:

```
\ This is a sample program
\ Filename, Tutor03.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^HouseLight = 7


\ Variables
\  L = Left Lever Response Counter


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

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


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

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

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 clock's current time). This shows that the Box was indeed loaded, and is now awaiting a start command. See Figure 4.1.

Figure 4.1 - Box Loaded and Waiting for Start Command


MED-PC V
Sessions Data View Macros Settings Help
Shortlist Macros Context Macros Default Save Desktop
Inputs
Outputs
Show Command Output Terminal
All Columns 10 Summary Rows 1
All Boxes
Box 1:
Box 2:

Box Status

	Started	Subject	Experiment	Group	Load	Program
Box 1					9:55	TUTOR03
Box 2						Closed

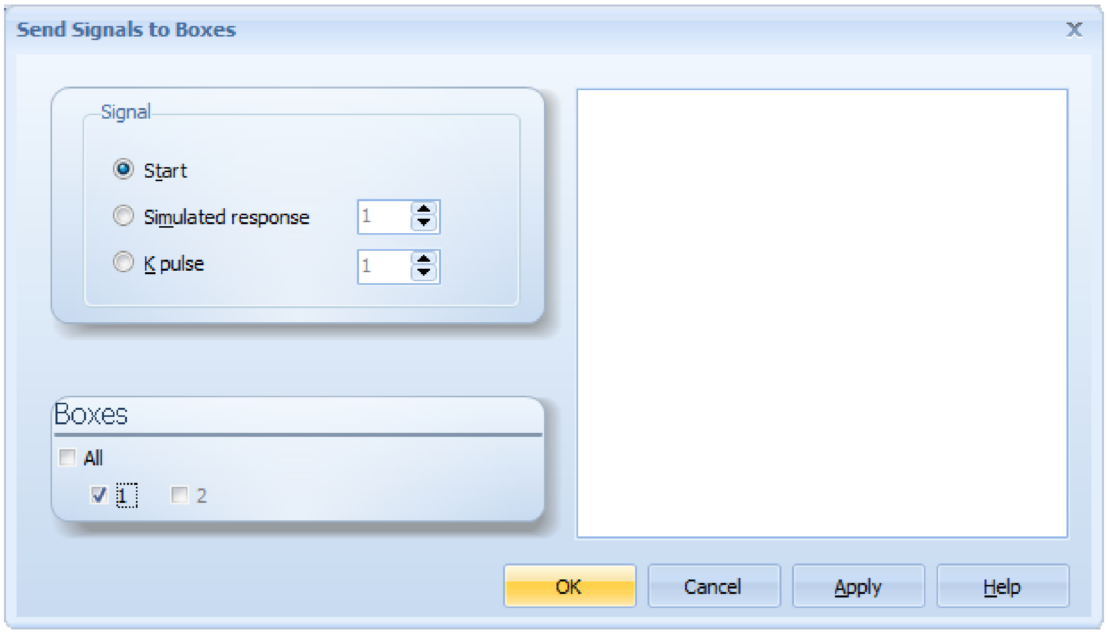
Log

```
3/01/2016  09:55:12 PCI version of DIG704 detected.  500 usec timebase will be requested.
3/01/2016  09:55:12 Resolution requested: 10 msec.
3/01/2016  09:55:13 Timebase of 500 usec successfully set.
3/01/2016  09:55:13 Driver version: 3.0 Build 0
3/01/2016  09:55:13 Rack 1 base address = 49408
3/01/2016  09:55:13 IRQ reported by driver = 16
3/01/2016  09:55:13 Rack 1 appears to have power.
3/01/2016  09:55:13 MED-PC Version: 5.0 Release: 0 Build: 13
3/01/2016  09:55:24  Box  1 loaded.  Subj:  Expt:  Grp:  Pgm: TUTOR03
```

3/1/2016 9:55:29 AM

Issue a START command by selecting **Sessions | Signals (Start, #K, #R)...** The screen shown below will appear.

Figure 4.2 - Send Signals to Boxes



Select the **START** radio button and the checkbox next to **BOX 1**, as shown in Figure 4.2, and then click the **OK** button. The program is now running.

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

The data has been saved to the hard disk in the selected MED-PC Data folder. MED-PC can now safely be closed.

This concludes the third Tutorial.

CHAPTER 5 | CREATING A FIXED RATIO (FR) SCHEDULE PROTOCOL

Z-Pulses (#Z)

MedState Notation utilizes Z-Pulses to communicate between State Sets. Programs composed of multiple State Sets are more readable and easier to maintain than single State Set programs, 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 pellet hopper 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 subject has or has not met the conditions defined).

Like State Sets and States, Z-Pulses must be numbered, with 32 being the highest allowable Z-Pulse number[3]. Unlike the other commands that have been learned so far and 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.

Output Syntax: `INPUT: ZP1 ---> NEXT`

Input Syntax: `#ZP1: OUTPUT ---> NEXT`

Where: P1 = An integer between 1 and 32 and is the same in both examples. Also note that when used as an *output*, the syntax is **ZP1** but when used as an *input*, the syntax is **#ZP1**.

In real code, it may look more like this:

```
S.S.1,
S1,
  #START: ON ^HouseLight; Z1 ---> S2  \ Issue Z1 to launch S.S.2


S.S.2,
S1,
  #Z1: ---> S2  \ Z1 received from S.S.1, S1
```

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 of 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 from 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 TRANS and type in the following:

```
\ This is an FR-5
\ Filename, Tutor04.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  L = Left Lever Response Counter
\  R = Reinforcer Counter


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

Add the code for the Fixed Ratio (remember, this is an FR-5), reinforcer counter, and a means of issuing reinforcers in another State Set. The Z-Pulse (Z1) allows for the reinforcer timer to be in another State Set that will be programmed later.

```
S2,     \ Reinforcer Counter and Display
        \ Z-Pulse (Z1) acting as an Output
  5#R^LeftLever: ADD R; SHOW 2,Reinforcers,R; Z1 ---> SX
```

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

```
S.S.2,  \ Response Counter and Display
S1,     \ This will put label the "Left Responses"
        \ and its value "L" on the screen after
        \ the START command is issued
  #START: SHOW 1,Left Responses,L ---> S2

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

Now it is time to insert the code for the reinforcer timer using the Z-Pulse generated in State Set 1, State 2. 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, State 2):

```
S.S.3,  \ Reinforcer Timer
S1,     \ Z-Pulse (Z1) acting as an Input
  #Z1: ON ^Reinforcer ---> S2

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

The final bit of code is for the session timer. It is identical to the session timer used in the previous Tutorial except it is now in S.S.4.

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

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

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

```
\ This is an FR-5
\ Filename, Tutor04.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  L = Left Lever Response Counter
\  R = Reinforcer Counter


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

S2,     \ Reinforcer Counter and Display
        \ Z-Pulse (Z1) acting as an Output
  5#R^LeftLever: ADD R; SHOW 2,Reinforcers,R; Z1 ---> SX


S.S.2,  \ Response Counter and Display
S1,     \ This will put label the "Left Responses"
        \ and its value "L" on the screen after
        \ the START command is issued
  #START: SHOW 1,Left Responses,L ---> S2

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

```
S.S.3,  \ Reinforcer Timer
S1,     \ Z-Pulse (Z1) acting as an Input
  #Z1: ON ^Reinforcer ---> S2

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


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

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

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 clock's current time).

Now issue the START command, then press the Left Lever repeatedly to get responses and reinforcer counts on the screen. When one minute has passed and the program has shut down the data will be saved to the hard disk.

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 subtraction (-). Although always an output command, two forms of this command are possible as indicated by Syntax A and Syntax B.

Syntax A: `INPUT: SET P1 = P2 ---> NEXT`

Syntax B: `INPUT: SET P1 = P2 Operator P3 ---> 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 within 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 Named Constant, however, is not permissible.

Real code may look like this:

```
1': SET A = A * 5, B = C(I) ---> SX  \ 2 SET commands strung together
#R3: SET A = (A + B) * 5 + C ---> SX
#START: SET A = A * 1" ---> S2
```

MedState Notation also allows for complex expressions (e.g., 1 + [(2 * 10) / 4] - 3) to be written directly:

```
SET A = 1 + ((2 * 10) / 4) - 3)
```

[4] Any mathematical function provided by Pascal can also be inserted within a MedState Notation statement using In-Line Pascal (see Appendix C).

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: `P1#T: OUTPUT ---> NEXT`

Where: P1 = Variable or array element

In the following example the variable "F" is set to a value of 1 second:

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

S2,
  F#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:

```
S2,
  F#T: ---> SX
  1":  ---> SX  \ Error #24.  Multiplie time statements in one state
```

Tutorial 5: Creating a Fixed Interval (FI) Schedule

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

```
\ This is an FI
\ Filename, Tutor05.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  F = Fixed Interval Value
\  L = Left Lever Response Counter
\  R = Reinforcer Counter


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

S4,     \ Reinforcer Counter and Display      \ Changed S2 to S4
  #R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> S3
                                              \ Was Show 2
                                              \ Delete 5 before #R
                                              \ Transition is to S3
S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2     \ Was SHOW 1

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


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


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

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

Next add the "SET" code below. In this program the variable "F" will represent the Fixed Interval time value. The default value will be 10 seconds, but later in this Tutorial the Fixed Interval value will be changed from 10 seconds to 15 seconds while the program is running *without changing any of the code*.

Insert the "SET" code below into State Set 1:

```
S1,
  1": SET F = 10 ---> S2

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

Finally add the time command #T with variable F as State 3 of State Set 1:

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

The final product should look like this:

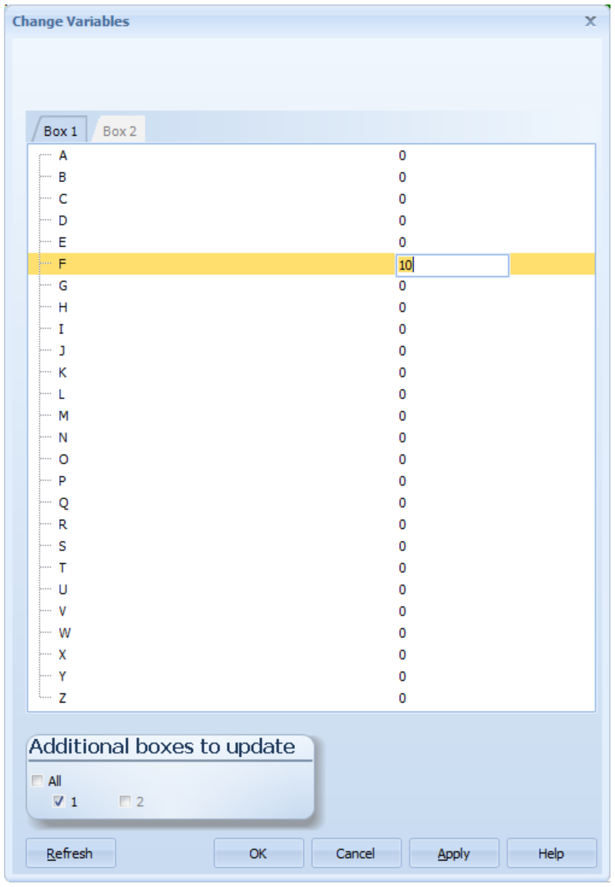
```
\ This is an FI
\ Filename, Tutor05.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  F = Fixed Interval Value
\  L = Left Lever Response Counter
\  R = Reinforcer Counter


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

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

S3,
  F#T: ---> S4

S4,      \ Reinforcer Counter and Display
  #R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> S3


S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
```

```
S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


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

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

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 reinforcer counts on the screen. After one minute has passed the program will end saving the data. DO NOT EXIT MED-PC. Instead, open the session again and follow the steps below to change the program to an FI-15.

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

Figure 6.1 - Displaying Variables from Box 1



A	0
B	0
C	0
D	0
E	0
F	10
G	0
H	0
I	0
J	0
K	0
L	0
M	0
N	0
O	0
P	0
Q	0
R	0
S	0
T	0
U	0
V	0
W	0
X	0
Y	0
Z	0

All [✓ 1] 2

The variable data for the displayed box can be changed from this screen. All Boxes can be changed by clicking "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 Fixed Interval) on one Box (Box 1) needs to be changed. The code written for this program uses the variable F as the Fixed Interval value, so select the row with the variable F.

To change this program to an FI-15, replace 10 with 15 and click **OK**. 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 any changes that were made are forgotten.

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." See Appendix A | Internal Representation of Time for more information on 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 until 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 an established criterion has been met. There are many variations to the IF command and this chapter will deal with three of them: Single, Nested, and Compound. 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 SECTION ---> NEXT1
          @Label2: OUTPUT SECTION ---> NEXT2
```

Where:		
	P1	= Named Constant, number, variable, array element, or special identifier
	P2	= Named Constant, number, variable, array element, or special identifier
	Operator	= 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
	OUTPUT SECTION	= Any legal output command(s)
	NEXT1 & NEXT2	= Any legal Transition such as SX, STOPDISCARD, STOPSAVE, or S1...S32 (given that S1...S32 is a valid State within the same State Set)

If the comparison evaluates as TRUE, then the immediately following statement (i.e. @Label1) is executed. If the comparison is false, then the second following statement (i.e. @Label2) is executed.

5 Please note, there are also three different 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 possible for the screen data to disagree with the saved data (the saved data would be correct, but the screen may not have had time to update prior to the end of the session) or the program may stop in the middle of an event (like issuing a reinforcer). 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,
  0.01": SET S = S + 0.01;
         IF S/60 >= M [@TrueEnd, @FalseContinue]
            @End: Z32 ---> S2
            @Cont: ---> SX

S2,
  2": ---> STOPSAVE
```

This code adds 0.01 to the variable S every 10 milliseconds, and then converts the new value S to minutes (S/60). If the Max Session Time (represented by the variable M) is set to sixty minutes, then this IF statement will stop the program at one hour or any fraction above it. If the session has been running for less than one hour, the program will continue running (coded for by @Cont: ---> SX). When an hour has passed, the program transitions to S2 where it waits two seconds before shutting down (allowing the screen to be updated, the reinforcer to be issued, 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).

Notice that the labels in the IF statement do not match exactly (@TrueEnd and @End, @FalseContinue and @Cont]. This is because labels are arbitrary and need not match. Because labels are arbitrary, spelling does not need to be consistent, however, they must begin with @. 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.

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 statement, 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:

Syntax:
```
INPUT: IF (P1 Operator P2) AND (P3 Operator P4) [@True, @False]
          @True:  OUTPUT ---> NEXT1
          @False: OUTPUT ---> NEXT2
```

Real Code may look like this:

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

S2,
  2": ---> STOPSAVE
```

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-5
\ Filename, Tutor06A.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  F = Fixed Ratio Value
\  L = Left Lever Response Counter
\  M = Max Session Time (Minutes)
\  R = Reinforcer Counter
\  S = Session Timer


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

S2,                                                    \ Changed S1 to S2
  #START: ON ^HouseLight ---> S3

S3,     \ Reinforcer Counter and Display               \ Changed S2 to S3
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> SX
                                                       \ Was SHOW 2
                                                       \ ADD F Before #R
```

```
S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2            \ Was SHOW 1

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


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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

The position of the displays for the response and the reinforcer counters have been shifted over one (SHOW 1 became SHOW 2) so that another counter can be added before them.

Also the definitions of several variables have been added at the beginning of the program. 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 F (the Fixed Ratio value) and M (Duration of the program) 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 Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF S/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,
  2": ---> STOPSAVE
```

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 F 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. A nested IF statement is an IF statement in the TRUE and/or FALSE output sections of another IF statement. 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 N will be the maximum number of reinforcers the subject will be allowed.

The program already used a Z1 pulse to signal the Reinforcer Timer. The new Z32 pulse terminates certain program functions after the Session Timer runs out or the subject has enough reinforcers. Without the Z32 pulse, the subject could get another reinforcer and the response counter could still be counting after the procedure is "terminated" because of the two 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: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  F = Fixed Ratio Value
\  L = Left Lever Response Counter
\  M = Max Session Time (Minutes)
\  N = Max Number of Reinforcers
\  R = Reinforcer Counter
\  S = Session Timer


S.S.1,  \ Main Control Logic for FR
S1,
  1": SET F = 5, M = 1, N = 10 ---> S2

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

S3,     \ Reinforcer Counter and Display
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> SX
  #Z32: ---> S1
```

```
S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Responses,A ---> S2

S2,
  #R^LeftLever: ADD A; SHOW 2,Responses,A ---> SX
  #Z32: ---> S1


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


S.S.4,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes, S/60;
         IF S/60 >= M [@True, @False]          \ Check Session Time
            @True: Z32 ---> S3
            @False: IF R >= N [@True, @False]  \ Check Max Reinforcers
                       @True: Z32 ---> S3
                       @False: ---> SX
                    \ Z32 Pulse added for signaling Session End.  It
                    \ will send both S.S.1, S3 and S.S.2, S2 back to S1

S3,
  2": ---> STOPSAVE
```

Save this program as Tutor06B.mpc then translate/compile it. Open MED-PC and load/run it. This program will now stop in one of two ways, when the subject has received the Max Reinforcers (defined by N) or the program Max Time has been reached (defined by M), whichever happens first.

Tutor06C.mpc

The primary purpose of this program is to see how a "Compound" IF statement works. A compound IF statement uses AND, OR, NOT, AND NOT, or OR NOT to create a more complex logical comparison. State 2 of State Set 4 contains the compound IF statement and simplifies the information in Tutor06B.mpc's into five lines (from seven).

```
\ NOTE: CHANGES IN BOLD ARE CHANGES FROM Tutor06B.MPC
\
\ This is an FR-5
\ Filename, Tutor06C.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7
```

```
\ Variables
\  F = Fixed Ratio Value
\  L = Left Lever Response Counter
\  M = Max Session Time (Minutes)
\  N = Max Number of Reinforcers
\  R = Reinforcer Counter
\  S = Session Timer


S.S.1,  \ Main Control Logic for FR
S1,
  1": SET F = 5, M = 1, N = 10 ---> S2

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

S3,      \ Reinforcer Counter and Display
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> SX
  #Z32: ---> S1


S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
  #Z32: ---> S1


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


S.S.4,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF (S/60 >= M) OR (R >= N) [@True, @False]
            @True: Z32 ---> S3
            @False: ---> SX

S3,
  2": ---> STOPSAVE
```

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 with 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 array variable defined.

DIM Command

The size of the array must be declared before the first State Set. As with all MedState Notation 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 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 total number of elements is P2 + 1. The element's indices range from 0...P2.

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 0, so an array with 25 elements to fill with data is written as:

```
DIM C = 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, however, 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

If defining an array too small results in an error, then the solution is to declare the array size larger than the amount of data that is expected to be collected. But, if for example the array is declared to have 500 data elements (DIM C = 499) and the subject only responds 25 times, then when the array is saved to the data file it would contain 475 empty data elements. To solve this problem, arrays should be sealed so that only the data of interest up to where the array is sealed is saved to the data file. All data after the array seal is excluded from the data file.

The array seal is accomplished with the MedState Notation command -987.987 in conjunction with the "SET" command (e.g., SET C(250) = -987.987). The real code would look like this:

```
S2,
  #R^LeftLever: IF I > 10000 [@ArrayFull, @Continue]
                  @Full: ---> S1
                  @Cont: SET C(I) = T, T = 0; ADD I;
                         IF I > 10000 [@ArrayFull, @SealArray]
                            @Full: ---> S1
                            @Seal: SET C(I) = -987.987 ---> SX
```

In this example (which will be seen again in the Tutorial), a test is done to see if the end of the array has been reached. If the array is not full, then the response is added to array C, the index into the array is incremented, and then another check for the end of the array is done. If the array is full, then the program transitions to S1 stopping 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 array is not full, then the array seal 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 Tutor06C.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: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7
```

```
\ Variables
\  C() = IRT Data Array
\  F   = Fixed Ratio Value
\  I   = Index into IRT Data Array C
\  L   = Left Lever Response Counter
\  M   = Max Session Time (Minutes)
\  N   = Max Number of Reinforcers
\  R   = Reinforcer Counter
\  S   = Session Timer
\  T   = IRT Timer


DIM C = 49


S.S.1,  \ Main Control Logic for FR
S1,
  1": SET F = 5, M = 1, N = 10 ---> S2

S1,
  #START: ON ^HouseLight ---> S3

S3,     \ Reinforcer Counter and Display
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> SX
  #Z32: ---> S1


S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
  #Z32: ---> S1


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX
  #Z32: ---> S1


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

S2,
  #R^LeftLever: SET C(I) = T, T = 0; ADD I ---> SX
  #Z32: ---> S1
```

```
S.S.6,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF (S/60 >= M) OR (R >= N) [@True, @False]
            @True: Z32 ---> S3
            @False: ---> SX

S3,
  2": ---> STOPSAVE
```

After opening MED-PC, load and start the program. When testing it, do not press the lever more than 50 times (remember, DIM C = 49). When the Box times out after one minute, go to the **MED-PC\Data** folder and open the just-created data file. You will see something like the following 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. Each value in the C Array is relative to the previous value. It is the amount of time that has elapsed since the previous response was recorded. Note that if you made less than 50 responses on the left lever, you will see 0's for all of the unused IRT array elements.

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: March 1, 2022


\ Inputs
^LeftLever = 1
```

```
\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  C() = IRT Data Array
\  F   = Fixed Ratio Value
\  I   = Index into IRT Array C
\  L   = Left Lever Response Counter
\  M   = Max Session Time (Minutes)
\  N   = Max Number of Reinforcers
\  R   = Reinforcer Counter
\  S   = Session Timer
\  T   = IRT Timer


DIM C = 999


S.S.1,  \ Main Control Logic for FR
S1,
  1": SET F = 5, M = 1, N = 10 ---> S2

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

S3,     \ Reinforcer Counter and Display
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> SX
  #Z32: ---> S1


S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
  #Z32: ---> S1


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX
  #Z32: ---> S1
```

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

S2,
  #R^LeftLever: IF I > 999 [@ArrayFull, @Continue]
                   @Full: ---> S1
                   @Cont: SET C(I) = T, T = 0; ADD I;
                          IF I > 999 [@ArrayFull, @SealArray]
                             @Full: ---> S1
                             @Seal: SET C(I) = -987.987 ---> SX
  #Z32: ---> S1


S.S.6,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF (S/60 >= M) OR (R >= N) [@True, @False]
            @True: Z32 ---> S3
            @False: ---> SX

S3,
  2": ---> STOPSAVE
```

Although the array is a much larger number than necessary (1000 elements), the array will seal itself no matter how the program stops. When the program is done running, look at the recently created data file in the **MED-PC\Data** folder. 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 a session where 43 responses were made:

```
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
```

In the sample data above the program was allowed to time out as opposed to getting the maximum numbers of reinforcers. Unlike the data file from the Tutor07A.mpc program, this data file does not contain all of the excessive zeros. This change was achieved by making use of -987.987 to seal the IRT array.

CHAPTER 9 | ARRAY COMMANDS AS OUTPUTS

An introduction to the LIST, RANDD, AND RANDI 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 programmer to create 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 create an array, the syntax of LIST is:

Syntax: `LIST P1 = P2, P3, ..., Pn`

Where:	P1	= The name of the array to be declared (A…Z)
	P2, P3, ..., Pn	= Numerical elements of the P1 array

Comments: A list declaration may be on more than one line, but each line must end on a comma. The last line should not end on a comma.

Example 1:

```
LIST Z = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

This example shows a LIST that has been declared all on one line.

Example 2:

```
LIST Z =  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
         11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
         21, 22, 23, 24, 25, 26, 27, 28, 29, 30
```

This example shows a LIST that has been declared on multiple lines. Note that the first two lines end with a comma and that the last line has no final comma.

LIST (as an output)

The second use of the LIST command is found in the output section of statements.

Syntax: `INPUT: LIST P1 = P2(P3) ---> NEXT`

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: The value of P3 is automatically incremented by the LIST command.

Array P2 may be declared with a LIST or DIM statement.

Stringing is permissible.

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 in sequential order, 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 Z = 1, 2, 3


S.S.1,
S1,
  1": LIST Y = Z(I); SHOW 1,Y Value,Y ---> SX
```

RANDD

RANDD is similar to LIST (as an output) and is used to automatically select data from an array created with the LIST command. The difference between the LIST and RANDD command is that while LIST pulls its values from the array sequentially, RANDD pulls them randomly without replacement.

Think of a bucket full of numbers. When a number is drawn from the bucket, that number is now considered used and is set aside making it no longer available. That number cannot be drawn from the bucket again until all numbers have been drawn, at which point the bucket is refilled with all of the available numbers.

RANDD draws values from the array in a Dependent manner (hence the final "D" in RANDD). Each value must be drawn from the array before an element can be repeated.

The following example shows how the program above could use the RANDD command:

```
LIST Z = 1, 2, 3


S.S.1,
S1,
  1": RANDD Y = Z; SHOW 1,Y Value,Y ---> 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 RANDD. Also, unlike the LIST program that would present the data 1, 2, 3, 1, 2, 3, …, this program might cause the numbers 1, 3, 2, 2, 1, 3, 2, 3, 1 to be successively displayed on the screen (all numbers in the list must first be selected before the list is refilled and a number can be reselected a second time).

RANDI

RANDI is closely related to RANDD with the difference being in that RANDI randomly selects from an array, but with replacement.

Using the bucket analogy again, when a number is drawn from the bucket, that number is immediately put back into the bucket and becomes available to be drawn again.

RANDI draws values from the array in an Independent manner (hence the final "I" in RANDI). That means that any previous access does not affect the next access. Each element has an equal chance of being selected with each RANDI call.

Substituting RANDI in the example code on the previous page might cause the numbers 2, 2, 1, 3, 2, 1, 3, 3, 1 to be successively displayed on the screen. If the program was allowed to run for an hour or more one should expect all three numbers to be drawn from the list about the same number of times.

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: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


\ Variables
\  C() = IRT Data Array
\  F   = Ratio Value Drawn from List Z
\  I   = Index into IRT Data Array C
\  L   = Left Lever Response Counter
\  M   = Max Session Time (Minutes)
\  N   = Max Number of Reinforcers
\  R   = Reinforcer Counter
\  S   = Session Timer
\  T   = IRT Timer
\  Z() = Variable Ratio Array


DIM C = 999

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


S.S.1,  \ Main Control Logic for VR
S1,
  1": SET M = 1, N = 10 ---> S2                  \ Removed F = 5

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

S3,
  1": RANDD F = Z; SHOW 4,VR =,F ---> S4
  #Z32: ---> S1

S4,     \ Reinforcer Counter and Display         \ Changed S3 to S4
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> S3
  #Z32: ---> S1                                  \ Transition is to S3
```

```
S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,     \ Don't count Responses during VR selection
  1":   ---> S3
  #Z32: ---> S1

S3,                                            \ Changed S2 to S3
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
  #Z32: ---> S1
  #Z1:  ---> S2


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX
  #Z32: ---> S1


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

S2,
  #R^LeftLever: IF I > 999 [@ArrayFull, @Continue]
                   @Full: ---> S1
                   @Cont: SET C(I) = T, T = 0; ADD I;
                          IF I > 999 [@ArrayFull, @SealArray]
                             @Full: ---> S1
                             @Seal: SET C(I) = -987.987 ---> SX
  #Z32: ---> S1


S.S.6,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF (S/60 >= M) OR (R >= N) [@True, @False]
            @True: Z32 ---> S3
            @False: ---> SX

S3,
  2": ---> STOPSAVE
```

As the program is run, notice that the value of VR is shown on the runtime screen. As responses are generated and reinforcers are issued, the Variable Ratio (VR) will equal 1, 5, 10, 15, 19 (not necessarily 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.

One other code change in S.S.2 should be called to attention. An additional State was added so that the program doesn't count left lever responses during the time that the program is selecting the VR. When the START command is issued, the program waits one second in S.S.1, S3 and then selects the first VR. S.S.2 also waits that same one second before it starts looking for responses on the left lever. When the VR has been met in S.S.1, S4, a Z1 pulse is issued and then the program goes back to S3 to wait one second before selecting the next VR. S.S.2, S3 looks for that Z1 pulse and when it receives it, it goes back to S2 and again waits the same one second before looking for responses on the left lever. This change means that only the responses that are counted towards the VR in S.S.1, S4 are added to the Left Lever Response Counter.

A similar change could have been done in S.S.5, but it was decided to let the IRT Data Array C count all responses. Even the ones that are not counted towards the VR.

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 "Max Reinforcers," rather than an obscure variable, such as "N." Note that variable aliases do not have any use within the body of MedState Notation programs and are simply directives placed before the first State Set that establish meaningful aliases (essentially synonyms) for program variables.

Syntax: `VAR_ALIAS P1 = P2`

Where: P1 = A descriptive label for the variable or array element

P2 = 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 (1=Yes  0=No) = 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 select whether or not to record SoftCR data before issuing the START command. The default value is set to 1=Yes to record SoftCR Data in S.S.1, S1.

Tutorial 9: Using the VAR_ALIAS Command

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

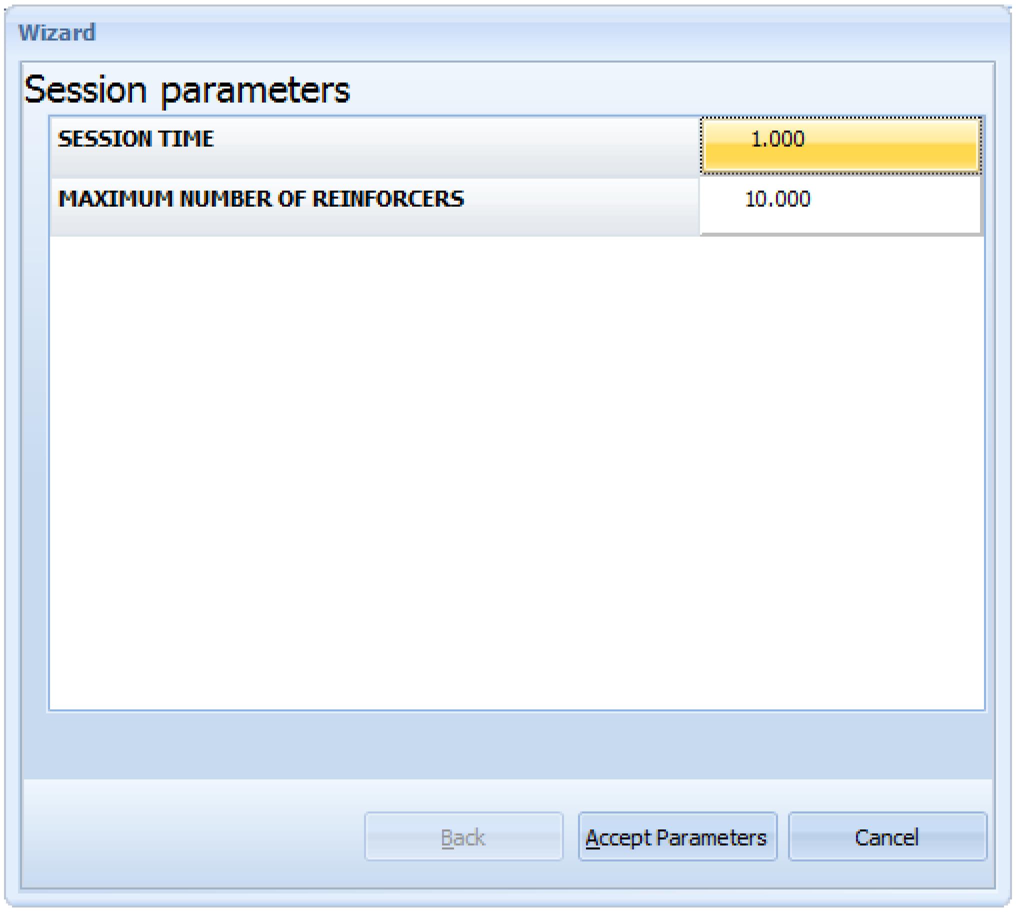
```
\ This is a VR-10
\ Filename, Tutor09.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


VAR_ALIAS Session Time                   = M  \ Default = 1 minute
VAR_ALIAS Maximum Number of Reinforcers = N  \ Default = 10


\ Variables
\  C() = IRT Data Array
\  F   = Ratio Value Drawn from List Z
\  I   = Index into IRT Data Array C
\  L   = Left Lever Response Counter
\  M   = Max Session Time (Minutes)
\  N   = Max Number of Reinforcers
\  R   = Reinforcer Counter
\  S   = Session Timer
\  T   = IRT Timer
\  Z() = Variable Ratio Array


DIM C = 999

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


S.S.1,  \ Main Control Logic for VR
S1,
  0.01": SET M = 1, N = 10 ---> S2                \ Changed 1" to 0.01"

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

S3,
  1": RANDD F = Z; SHOW 4,VR =,F ---> S4
  #Z32: ---> S1

S4,     \ Reinforcer Counter and Display
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> S3
  #Z32: ---> S1
```

```
S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,     \ Don't count Responses during VR selection
  1":   ---> S3
  #Z32: ---> S1

S3,
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
  #Z32: ---> S1
  #Z1:  ---> S2


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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


S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX
  #Z32: ---> S1


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

S2,
  #R^LeftLever: IF I > 999 [@ArrayFull, @Continue]
                 @Full: ---> S1
                 @Cont: SET C(I) = T, T = 0; ADD I;
                        IF I > 999 [@ArrayFull, @SealArray]
                           @Full: ---> S1
                           @Seal: SET C(I) = -987.987 ---> SX
  #Z32: ---> S1


S.S.6,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF (S/60 >= M) OR (R >= N) [@True, @False]
            @True: Z32 ---> S3
            @False: ---> SX

S3,
  2": ---> STOPSAVE
```

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 **Sessions** | **Wizard for Loading Boxes**.

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

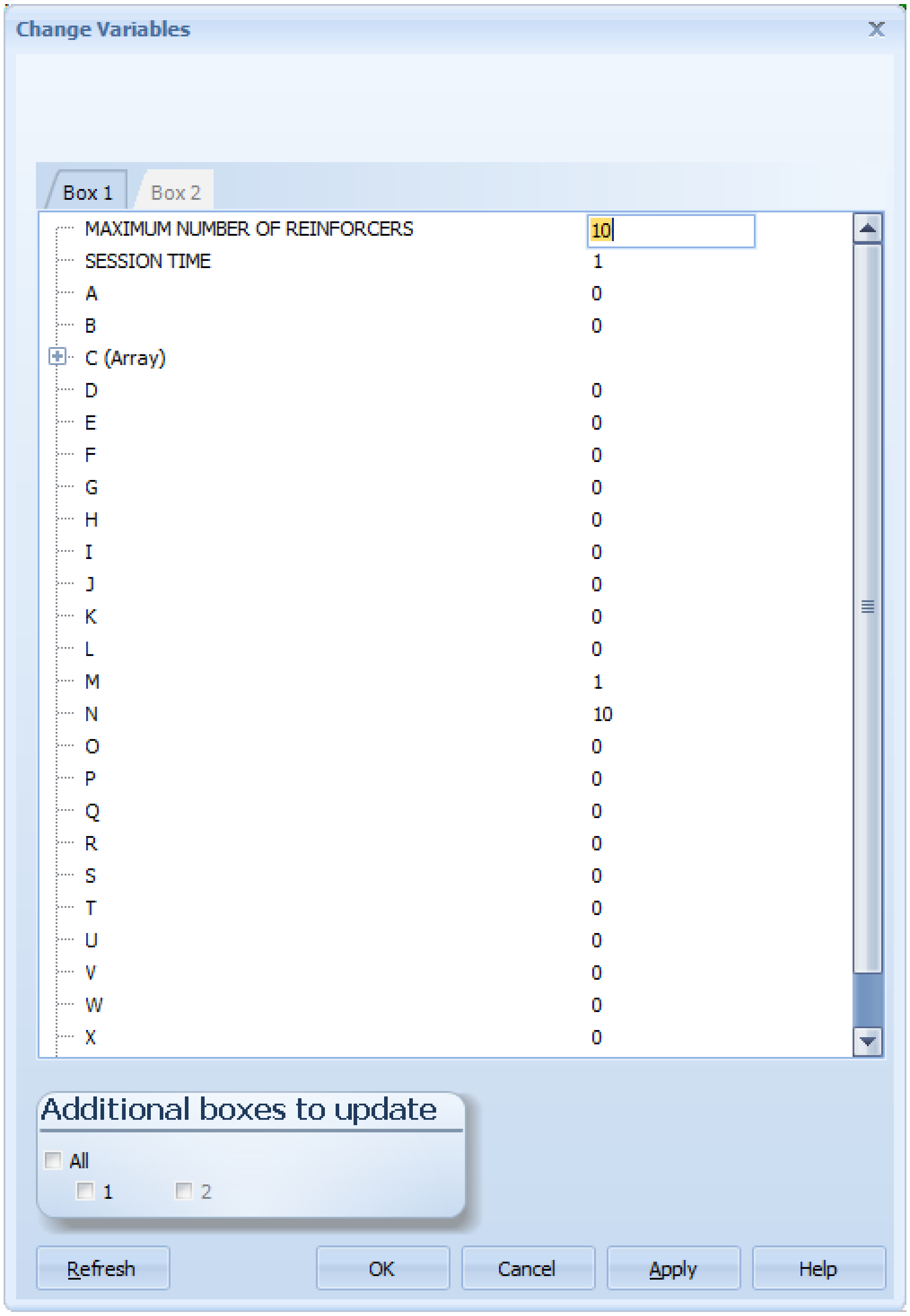
Figure 10.1 - Session Parameters Screen



SESSION TIME	1.000
MAXIMUM NUMBER OF REINFORCERS	10.000

The Named Variables can also be changed once MED-PC is started by selecting **Sessions | Change Variables..**. from the menu. Select the appropriate Box and either change the variables M and N directly or change the Named Variables that are listed at the top of the screen. Click **Apply** and **OK** to accept the changes.

Figure 10.2 - Displaying Variables and Named Variables for Box 1


Change Variables

Box 1 | Box 2

MAXIMUM NUMBER OF REINFORCERS	10
SESSION TIME	1
A	0
B	0
C (Array)	
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0
M	1
N	10
O	0
P	0
Q	0
R	0
S	0
T	0
U	0
V	0
W	0
X	0

Additional boxes to update

☐ All

☐ 1 ☐ 2

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

Up to this point the programs have demonstrated how to assign variables, create and fill arrays, record data, and how to display that data. However, none of those programs used all of the available variables and those unused variables ended up with zeros next to them in the data files. This can lead to a cluttered data file with a lot of useless numbers. 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 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. The default is Portrait.

Syntax: `PRINTORIENTATION = P1`

Where: P1 = LANDSCAPE or PORTRAIT

Setting the Number of Columns on Printouts/Data Files (PRINTCOLUMNS/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 menuing system. 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. The default is five columns.

Syntax: `PRINTCOLUMNS = P1`

Where: P1 = 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 font size used to print data. The use of this command will override any defaults set within the MED-PC menuing system. The default is a 12-point font size.

Syntax: `PRINTPOINTS = P1`

Where: P1 = The number of points

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 total number of spaces to be occupied by the number including the decimal point.

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

PRINTFORMAT Examples

```
PRINTFORMAT = 5.1  \ Print in five spaces, with 3 to left of decimal
                   \ point and 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.

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

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 the MedState Notation program or from the menuing system. 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 = The variables and/or arrays A through Z to be printed

Real Code may look like this:

```
PRINTVARS = C, L, M, N, R, S, Z
```

PRINTOPTIONS (Condensed vs. Full Headers) (Formfeeds vs. No Formfeeds)

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 or CONDENSEDHEADERS.

Full Header Example:

```
Start Date: 03/01/16
End Date: 03/01/16
Subject: 0
Experiment: 0
Group: 0
Box: 1
Start Time: 14:07:54
End Time: 14:08:20
MSN: TUTOR10
```

Condensed Header Example:

```
BOX:  1 SUBJECT:       0 EXPERIMENT:       0 GROUP:       0 MSN:  TUTOR10
START: 03/01/16  14:07:54  END: 03/01/16  14:08:20
```

Options for when to print the data include FORMFEEDS or NOFORMFEEDS. The FORMFEEDS option specifies that the data will be ejected from the printer after every PRINT command; whereas NOFORMFEEDS indicates that the data will be sent to the printer and will remain queued in the printer until a final form feed is sent to the printer.

The defaults are CONDENSEDHEADER and NOFORMFEED.

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 ---> STOPSAVE
```

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: March 1, 2022


\ Inputs
^LeftLever = 1

\ Outputs
^Reinforcer = 3  \ In this code, this is a Pellet Dispenser
^HouseLight = 7


VAR_ALIAS Session Time                  = M  \ Default = 1 minute
VAR_ALIAS Maximum Number of Reinforcers = N  \ Default = 10
```

```
\ Variables
\  C() = IRT Data Array
\  F   = Ratio Value Drawn from List Z
\  I   = Index into IRT Data Array C
\  L   = Left Lever Response Counter
\  M   = Max Session Time (Minutes)
\  N   = Max Number of Reinforcers
\  R   = Reinforcer Counter
\  S   = Session Timer
\  T   = IRT Timer
\  Z() = Variable Ratio Array


DIM C = 999

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


PRINTORIENTATION = LANDSCAPE
PRINTCOLUMNS     = 4
PRINTOPTIONS     = FULLHEADERS, FORMFEEDS
PRINTVARS        = C, L, R


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

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

S3,
  1": RANDD F = Z; SHOW 4,VR =,F ---> S4
  #Z32: ---> S1

S4,     \ Reinforcer Counter and Display
  F#R^LeftLever: ADD R; SHOW 3,Reinforcers,R; Z1 ---> S3
  #Z32: ---> S1


S.S.2,  \ Response Counter and Display
S1,
  #START: SHOW 2,Left Responses,L ---> S2

S2,     \ Don't count Responses during VR selection
  1":   ---> S3
  #Z32: ---> S1

S3,
  #R^LeftLever: ADD L; SHOW 2,Left Responses,L ---> SX
  #Z32: ---> S1
  #Z1:  ---> S2


S.S.3,  \ Reinforcer Timer
S1,
  #Z1: ON ^Reinforcer ---> S2

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

```
S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX
  #Z32: ---> S1


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

S2,
  #R^LeftLever: IF I > 999 [@ArrayFull, @Continue]
                  @Full: ---> S1
                  @Cont: SET C(I) = T, T = 0; ADD I;
                         IF I > 999 [@ArrayFull, @SealArray]
                            @Full: ---> S1
                            @Seal: SET C(I) = -987.987 ---> SX
  #Z32: ---> S1


S.S.6,  \ Session Timer & Max Reinforcer Limiter
S1,
  #START: SHOW 1,Session Minutes,S/60 ---> S2

S2,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60;
         IF (S/60 >= M) OR (R >= N) [@True, @False]
            @True: Z32 ---> S3
            @False: ---> SX

S3,
  2": PRINT ---> STOPSAVE
```

Translate and compile the program, open it in MED-PC and run. After one minute or ten reinforcers 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 concludes Tutorial 10.

CHAPTER 12 | MACROS

What are Macros and Why Should I Use Them?

The macro feature is one of the key aspects of the MED-PC software. Macros are text files that automate routine 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 or mouse can also be issued from a macro file.

Creating Macros

The easiest way to create a macro is to turn on the macro recorder, use the menuing 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 record button (it looks like a red circle) on the tool bar or by selecting the **Macros | Record Macro** menu item. During recording, the text "Recording Macro" appears at the bottom of the main window of MED-PC and the red circle changes to a red square on the tool bar.

The macro recorder may be turned off by either clicking the record button on the tool bar a second time or by selecting the **Macros | Stop Recording** menu item. Selecting this option presents a file dialog that allows the user to specify the file name and directory for the macro.

Using the DELAY Command/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 P1`

Where: P1 = 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, a MedState Notation 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, pause for a brief time to allow the defaults to be set, and then override some of the values. Without a time delay between loading the program and overriding the defaults, it would be possible to change the variables in the macro, and to then have the program change the values back to their defaults. Consider the following program:

```
\ FR


\ A = FR Size


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

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

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

This program arranges a simple FR. 100 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 FR
SET A VALUE 20 MAINBOX 1 BOXES 1
```

When this macro is run, it will load the program FR into Box 1 and then variable "A" will be set to the value of 20 almost immediately. The FR program will also start running as soon as it is loaded, but S.S.1, S1 will not execute until 0.1" (100 milliseconds) after the program is loaded which will cause the variable "A" to be set back to the value of 10.

The following macro avoids this problem by introducing a delay of 1000 milliseconds (1s) after the Box is loaded before trying to set the value of A.

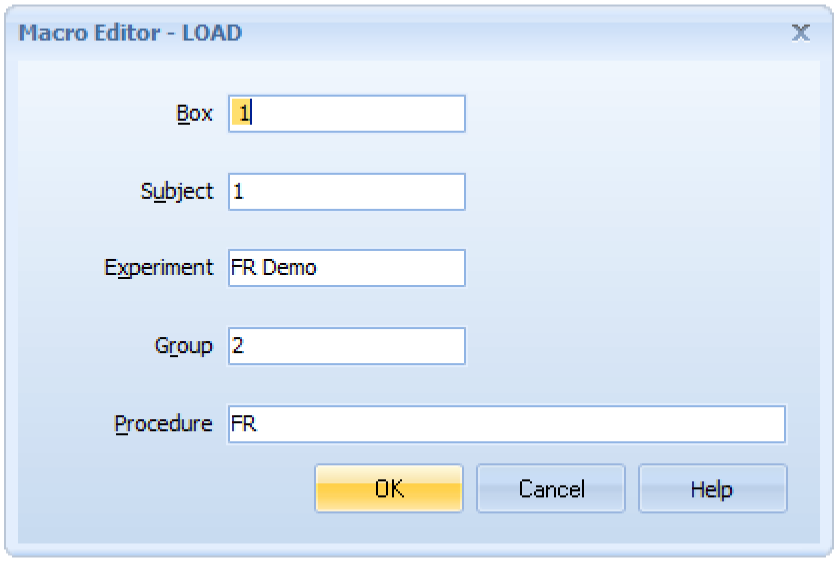
```
LOAD BOX 1 SUBJ 1 EXPT FR Demo GROUP 2 PROGRAM FR
DELAY 1000
SET A VALUE 20 MAINBOX 1 BOXES 1
```

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...** A screen similar to the one shown below will appear.

Figure 12.1 - Macro Editor - Load



Macro Editor - LOAD

Box: 1

Subject: 1

Experiment: FR Demo

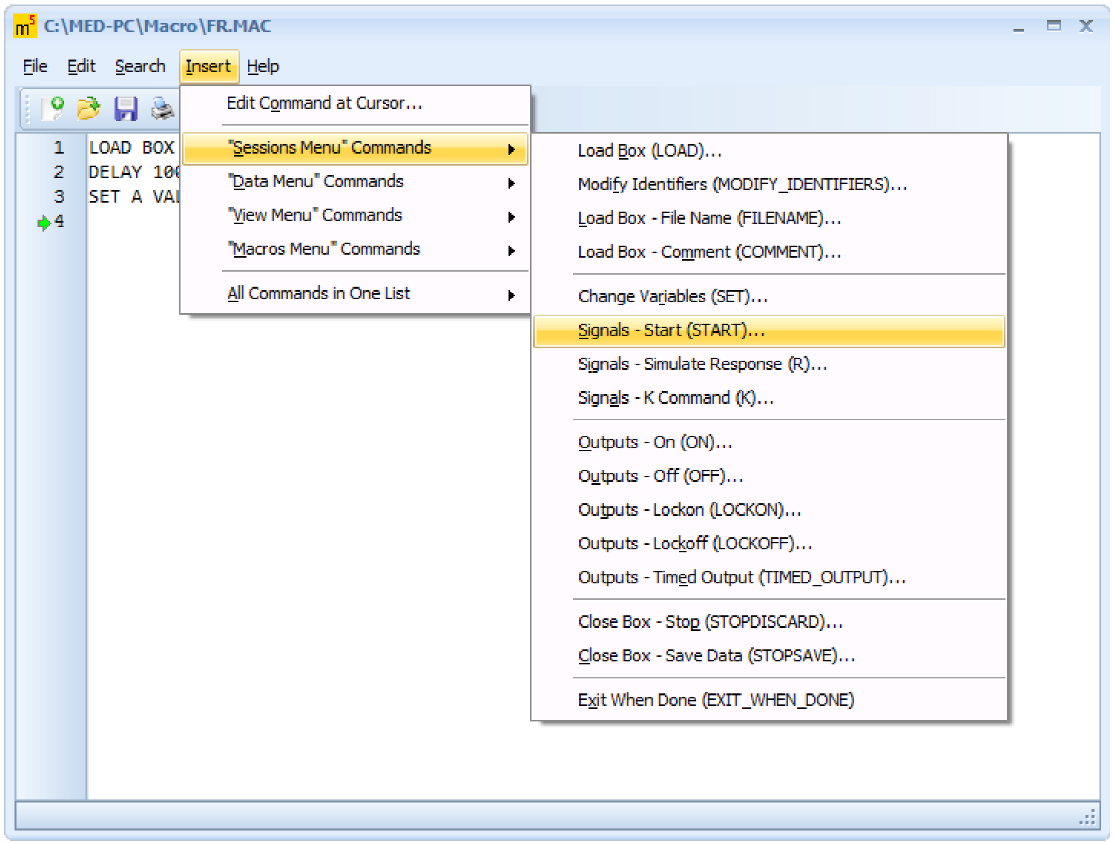
Group: 2

Procedure: FR

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 then select the **Insert** menu option. The menu will present a list of commands that may be inserted. See Figure 12.2. Choosing a command then presents a dialog box (similar to the example in Figure 12.1 above) to simplify creating the command.

Figure 12.2 - Insert Menu


C:\MED-PC\Macro\FR.MAC
File
Edit
Search
Insert
Help
LOAD BOX
DELAY 10
SET A VAL
Edit Command at Cursor...
"Sessions Menu" Commands
"Data Menu" Commands
"View Menu" Commands
"Macros Menu" Commands
All Commands in One List
Load Box (LOAD)...
Modify Identifiers (MODIFY_IDENTIFIERS)...
Load Box - File Name (FILENAME)...
Load Box - Comment (COMMENT)...
Change Variables (SET)...
Signals - Start (START)...
Signals - Simulate Response (R)...
Signals - K Command (K)...
Outputs - On (ON)...
Outputs - Off (OFF)...
Outputs - Lockon (LOCKON)...
Outputs - Lockoff (LOCKOFF)...
Outputs - Timed Output (TIMED_OUTPUT)...
Close Box - Stop (STOPDISCARD)...
Close Box - Save Data (STOPSAVE)...
Exit When Done (EXIT_WHEN_DONE)

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. (Ex: "Prompt to load the Subject" or to "Close the door to the lab.")
- Rather than entering parameters via the "Sessions | 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.
- Frequently used macros may be assigned to shortcut keys such as Alt+F1. This can make certain tasks such as issuing a free reinforcer easier.

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 | Record Macro**
2. Select **Sessions | Load Box...**
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. Now select **Macros | Insert 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
5. Select **Sessions | Change Variables...**
6. Select the **Box 1** checkbox
7. Change the "Session Time" to **5** minutes and the "Maximum Number of Reinforcers" to **15** and click on the **Apply** button
8. Close the "Change Variables" window by clicking on the **OK** button
9. Select **Sessions | Signals (#Start, #K, #R)...**
10. Select the **Box 1** checkbox and the **START** option and then click on the **OK** button
11. Select **Macros | Stop Recording**
12. 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 **Macros** | **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 "MAXIMUM NUMBER OF REINFORCERS" VALUE 15 MAINBOX 1 BOXES  1
SET "SESSION TIME" VALUE 5 MAINBOX 1 BOXES  1
START BOXES  1
```

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

CHAPTER 13 | INPUTSTOLEVEL AND INPUTSTOINVERT COMMANDS

The IC-124 OmniCtrl cards do not have any jumpers or switches for setting the Toggle/Level Mode or Normal/Invert Mode for inputs on the cards. When using these cards there are two special MedState Notation commands that must be used in order to change the default settings for the inputs.

NOTE: These two commands are for SOF-737 MED-PC VI or newer.

While the below definitions and examples reference a lever the information remains the same if the input was changed to an IR Beam or any other type of input.

Inputs in **Toggle Mode** will generate an input one time and one time only when the lever is pressed. Another input will not be generated until the lever is released and pressed a second time.

Inputs in **Level Mode** will generate an input every interrupt for as long as the lever is held down. The inputs will only stop when the lever is released. The Temporal Resolution for the interrupt can be set to 1ms or 10ms in the Hardware Configuration Utility.

Inputs in **Normal Mode** will generate an input in Toggle Mode or multiple inputs in Level Mode when the lever is pressed.

Inputs in **Invert Mode** will generate an input in Toggle Mode or multiple inputs in Level Mode when the lever is released.

The default settings for all inputs on the IC-124 OmniCtrl card are Toggle Mode and Normal Mode.

NOTE: These two commands should always be used before the START command. If a program changes several inputs to Level mode and/or Invert mode, then it is possible for the hardware to not complete the change before the program tries to use it.

INPUTSTOLEVEL (MED-PC VI OR NEWER)

When MED-PC is started or when a program is finished running in a Box all inputs that were assigned to that Box with the Hardware Configuration Utility are set back to the default setting of Toggle Mode. If a program needs an input to be in Level Mode, then that program must use the INPUTSTOLEVEL command.

Syntax: `INPUT: INPUTSTOLEVEL P1, P2, ..., Pn ---> NEXT`

Where: P1, P2, ..., Pn = Number

Comments: Number should be an input that has been declared in the Hardware Configuration Utility and that the current program is using.

Number must be a whole number in the range 1...80.

This command has no effect on a card that is not a IC-124 OmniCtrl. The hardware will ignore it so it is safe to have one Box that is using a DIG-716 SmartCtrl card or a

DIG-712 SuperPort card and another Box with a IC-124 OmniCtrl card. The same program with this command can be used in both Boxes.

Real code may look like this:

```
S.S.1,
S1,
  0.01": INPUTSTOLEVEL 1,2,3,4,5,6,7,8 ---> S2
```

This command will set inputs 1 through 8 to Level Mode in the Box running the code.

INPUTSTOINVERT (MED-PC VI OR NEWER)

When MED-PC is started or when a program is finished running in a Box all inputs that were assigned to that Box with the Hardware Configuration Utility are set back to the default setting of Normal Mode. If a program needs an input to be in Invert Mode, then that program must use the INPUTSTOINVERT command.

Syntax: `INPUT: INPUTSTOINVERT P1, P2, ..., Pn ---> NEXT`

Where: P1, P2, ..., Pn = Number

Comments: Number should be an input that has been declared in the Hardware Configuration Utility and that the current program is using.

Number must be a whole number in the range 1...80.

This command has no effect on a card that is not a IC-124 OmniCtrl. The hardware will ignore it, so it is safe to have one Box that is using a DIG-716 SmartCtrl card or a DIG-712 SuperPort card and another Box with a IC-124 OmniCtrl card. The same program with this command can be used in both Boxes.

Real code may look like this:

```
S.S.1,
S1,
  0.01": INPUTSTOINVERT 1,2,3,4,5,6,7,8 ---> S2
```

This command will set inputs 1 through 8 to Invert Mode in the Box running the code.

Tutorial 12: How to time the length of an Input

Timing the length of an input whether it is a lever press or a nose poke is a very common task to perform. This tutorial will demonstrate one way to accomplish this task.

Open Trans and enter the following text:

```
\ This program demonstrates timing the length of an input
\ Filename, Tutor12.mpc
\ Date: March 1, 2022


\ Inputs
^LeftLever = 1
```

```
\ Outputs
\  None


\ Variables
\  I   = Index into Array Z
\  T   = Response Timer
\  Y   = Input Timer
\  Z() = Array for Recording Response Lengths


DIM Z = 1000


\ Z-Pulses Used in this Program
\ ^Z_LeftBreak   = 3  \ Left Lever Press
\ ^Z_LeftRelease = 4  \ Left Lever Release


S.S.1,  \ RESPONSE TIMER
S1,     \ Set Input 1 to Level Mode
  0.01": INPUTSTOLEVEL 1 ---> S2   \ Always do this before the START command

S2,
  #START: SHOW 1,Last Resp Len,0, 2,Current Resp Len,0;
          SET Z(I) = -987.987 ---> S3

S3,     \ Wait for a signal that a Response has started
  #Z^Z_LeftBreak: SET T = 0 ---> S4

S4,     \ Wait for signal that the input has been released
        \ and Record the length of the Response
  0.01": SET T = T + 0.01;
         SHOW 2,Current Resp Len,T ---> S4
  #Z^Z_LeftRelease: SET Z(I) = T;
                    SHOW 1,Last Resp Len,Z(I);
                    ADD I; SET Z(I) = -987.987;
                    SHOW 2,Current Resp Len,0 ---> S3


S.S.2,  \ RESPONSE DEFINED - 20ms RESPONSE, 20ms RELEASE
S1,     \ Inputs in Level Mode generate an input "count" on each
        \ interrupt.  With a 10ms system resolution 2 counts will
        \ be reached in 20ms.  The Z^Z_LeftBreak pulse is used to
        \ signal a completed Response.  The second statement resets
        \ the counter every 20ms so that a partial Response of
        \ less than 20ms will not be counted.
  #R^LeftLever: ADD Y; IF Y >= 2 [@Response, @NoResponse]
                         @Response: SET Y = 0; Z^Z_LeftBreak ---> S2
                         @NoResponse: ---> SX
  0.02": SET Y = 0 ---> S1

S2,     \ As long as the input is broken the second statement
        \ causes a re-entry to this State.  This resets the
        \ internal 20ms timer so it never times out.  When the
        \ input is released for 20ms the timer times out and a
        \ Z^Z_LeftRelease pulse signals the release.
  0.02": Z^Z_LeftRelease ---> S1
  #R^LeftLever: ---> S2
```

In the above program S.S.1 looks for the Z3 pulse to indicate the beginning of a Response. When the Z-Pulse is received it starts a timer to record the length of the response. When the Z4 pulse is received the State Set records and displays the length of the response.

S.S.2 is used to track when a Response starts and ends. S1 tracks the start of a Response. When an input is in Level Mode it generates an input "count" on each interrupt. With a 10ms system resolution 2 input counts will be reached in 20ms. The Z3 pulse is used to signal a completed Response. The 0.02" timer in S.S.2, S1 resets the counter every 20ms so that a partial Response of less than 20ms will not be counted.

S2 tracks the end of a Response. As long as the input is broken the #R^LeftLever input causes a re-entry to this State. This resets the 0.02" (20ms) timer so it never times out. When the input is released for 20ms the timer times out and a Z4 pulse signals the release.

CHAPTER 14 | 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 10ms, the Boxes are serviced every 10ms. In the discussions that follow, it will be assumed that a system with 10ms resolution is applicable. These timed-based interrupts are generated by the DIG-704/DIG-750 interface card that plugs into the chassis of the PC or by the DIG-703/DIG-705 device connected to the computer via a USB cable.

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 #Rs presented through the menuing system since the initiation of the last processing sweep will be merged with any that were read from the input cards. For example, if a menu #R1 was recently generated for Box 3 and a hardware #R2 for Box 3 was also recorded, both the #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 menu 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, in practice, it is very unlikely that responses will be missed. Several factors make it unlikely the PC will miss a response including:

- Responses are latched in the hardware buffer until read (the response does not have to occur at the exact same time as the interrupt).
- A system resolution of 10 milliseconds is far faster than subjects can respond.
- Two responses occurring less than 10ms 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 a response on input 1 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, #X, 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 a #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, because computers are so fast, all of this is occurring in the FIRST MILLISECOND of each 10ms 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.	#X	Used as a time of day input HH:MM:SS, e.g., 131500#X (1:15 pm)
vii.	#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 (#Ks, #Rs, and #STARTs) 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 regardless 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 #Z1s 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 transitions 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 inputs, all other inputs are ignored.

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

   Stacked means more than one input in a single State. Example:

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

   In the event that the two Z-pulses are simultaneously received, the topmost statement will execute. In this example, if both the Z1 and Z2 received simultaneously, the Z1 will be processed because it is stacked above the Z2 input.

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 before the next interrupt is processed.

    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" that 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 be the new "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 a K1 is issued. This SHOW will display the label "A_Val" with a value of "1" in position 1 and "C_Val" incrementing in position 3. "B_Val" is never displayed and subsequent occurrences of K1 are not reflected in "A_Val" 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,
  #K1: ADD A; SHOW 1,A_Val,A ---> SX
  #Z1: Z2 ---> S2

S2,
  0.01": ADD B; SHOW 2,B_Val,B ---> SX  \ Never executed; #Z2 always
                                        \   occurs immediately
  #Z2: ---> S3                          \ #Z2 detected in same Clock
                                        \   Tick in which it is issued
S3,
  1": ADD C; SHOW 3,C_Val,C ---> SX


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

EXAMPLE B:

The following code is very similar to Example A. In this example the first statement in State Set 1 is changed so that a transition occurs to S1 (replacing SX). The result, however, is exactly the same as in Example A.

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

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

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


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

EXAMPLE C:

In this example, issuing a K1 causes a transition from S.S.1, S1 to S.S.1, S4. Hence, when the Z1 is issued in S.S.2, S1; S.S.1, S1 is no longer the active state; S4 has become the active state. As a result, "B_Val" and "C_Val" are not displayed when the K1 is issued.

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

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

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

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


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

EXAMPLE D:

In the following example, issuing a K1 causes a transition to S.S.1, S5. In S.S.2, it generates a Z1 pulse. During the Z-Pulse-processing phase, S.S.1 is in S5 so the Z1 required for transition is received and "E_Val: 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,
  #K1: ADD A; SHOW 1,A_Val,A ---> S5
  #Z1: Z2 ---> S2

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

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

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


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

Additional Commentary on Time Based Inputs

Time-based inputs (", ', #T, and #X) 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. Example:

   ```
   S5,
     0.001": ADD B ---> S5
     #R1:    ADD A ---> S6
   ```

   In the above code the #R1 will never get executed because the timed input criteria will always be met first.

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.

   ```
   S5,
     #R1:    ADD A ---> S6
     0.001": ADD B ---> S5
   ```

   In the above code if the #R1 happens at the exact same time as the timed interrupt, then the #R1 input will be processed first. Also note that if the temporal resolution is set to 10ms (0.01"), then the second statement will time 10ms and not 1ms. The system cannot time less than the selected temporal resolution.

4. In the following situation, imagine a system resolution of 10ms (0.01") and that a #R1 occurs and the 10" time duration times out on the same interrupt sweep (within the 10ms 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 10ms 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. DIG-704/DIG-750 interface card that plugs into the chassis of the PC or the DIG-703/DIG-705 device connected to the computer via a USB cable, generate the interrupt signal, with an accuracy of 0.005%. Keep in mind that all PCs 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. We call this the temporal resolution of the system.

Some users may wish to think of the temporal resolution 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 temporal resolution value, and time durations have been set equal to the temporal 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 on and off every 10ms. 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 9ms to process. Box 16's output would be toggled on 9ms after initiation of the sweep and the next sweep would occur 1ms later. If on the second sweep, Boxes 1 through 15 required very little time to process, such that Box 16 is serviced 1ms after initiation of the sweep, its output would be turned off 2ms after being turned on (rather than the 10ms 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 of all cycles" (displayed by going to **View | System Performance**) is appreciably less than the resolution value.

CHAPTER 15 | ADVANCED PROGRAMMING TECHNIQUES

IF Statements

The IF statement has multiple different syntaxes that can be used.

IF Statement Syntax A:

```
IF P1 Operator P2 [@Label1, @Label2]
   @Label1: Output Section ---> Transition
   @Label2: Output Section ---> Transition
```

Labels are arbitrary. Position of the labels is what matters

```
S1,
  #R1: ADD A; IF A = B [@True, @False]
                 @False: SET A = 0 ---> S2  \ True Transition
                 @True: ---> SX             \ False Transition
```

IF Statement Syntax B:

```
IF P1 Operator P2 [@Label1]
   @Label1: Output Section ---> Transition
```

If the comparison is false, then the transition is to SX

```
S1,
  #R1: ADD A; IF A = B [@True]
                 @T: SET A = 0 ---> S2  \ True Transition
```

IF Statement Syntax C:

```
IF P1 Operator P2 [Output Section] ---> Transition
```

If the comparison is false, then the transition is to SX

```
S1,
  #R1: ADD A; IF A = B [SET A = 0] ---> S2  \ True Transition
```

It should be noted that all three of the above if statements do the exact same thing. If A = B, then in all three if statements A will be reset to 0 and the program will transition to S2.

The significance of SX

What is the difference between how these two States Sets act?

```
S.S.1,
S1,
  #R1: ADD A ---> S1


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

The answer is there is no difference. The two State Sets act exactly the same. However, if a timed input is added to each State Set, then the behavior of the two State Sets becomes very different.

```
S.S.1,
S1,
  #R1: ADD A ---> S1
  1': ---> S2


S.S.2,
S1,
  #R1: ADD A ---> SX
  1': ---> S2
```

In S.S.1 when a response happens on input 1 it **resets** the one minute timed input. This is because the code is transitioning to S1. A transition to a specific State resets all input conditions to that State. So the program will not transition to S2 until one minute has passed and there has been no responses on input 1. This might be used as a punishment during a time out. The animal must not respond during the time out in order for the program to proceed.

In S.S.2 when a response happens on input 1 it **does not** reset the one minute timed input. This is because the code is transitioning to SX. A transition to SX sends the program back to the same State, but does not reset any of the input conditions for that State. So the program will transition to S2 after one minute has passed regardless of any responses that might have happened on input 1. This might be used when the number of responses during the time out need to be recorded, but there is no punishment for those responses.

How to Flash and Output

A common task is to flash an output (e.g. cue light) a variable number of times. The following example flashes an output 4 times in one second.

```
\ Inputs
^LeftLever = 1

\ Outputs
^LeftLight = 1


\ A() = Control Variables with Assigned Aliases as Defined
Var_Alias # of Times to Flash Light in 1s = A(0)  \ Default = 4 Times/second
Var_Alias Flash Duration (sec)            = A(1)  \ Default = 5 seconds


\ List Working Variables Here
\  E = Duration of each ON/OFF cycle
\      Output will be ON for E seconds and OFF for E seconds
\
\      Equation =    1   /    A(0)    /    2     *     1"
\                 second   # of Times   On & Off   convert to
\                            to Flash               MED Ticks
\
\      e.g. E = 1 / 4 / 2 = 0.125s   This value is then multiplied
\                                    by 1" to convert to MED Ticks


DIM A = 1


\ Z-Pulses Used in This Program
\  Z1 = Signal to Start the Flash
\  Z2 = Signal to End   the Flash


\***************************************************
\                   MAIN PROGRAM
\***************************************************
S.S.1,
S1,
  0.01": SET A(0) = 4, A(1) = 5 ---> S2

S2,
  #START: SET A(1) = A(1) * 1";
          SET E    = 1 / A(0) / 2 * 1" ---> S3

S3,     \ Wait for Response
        \ Signal to Flash the Light
  #R^LeftLever: Z1 ---> S4

S4,     \ Time the Flash Duration
  A(1)#T: Z2 ---> S3
```

```
\**************************************************
\                  FLASH THE LIGHT
\**************************************************
S.S.2,
S1,
  #Z1: ON ^LeftLight ---> S2

S2,     \ Light On Cycle
  E#T: OFF ^LeftLight ---> S3
  #Z2: OFF ^LeftLight ---> S1

S3,     \ Light Off Cycle
  E#T: ON ^LeftLight ---> S2
  #Z2: ---> S1
```

How to time the length of an input

Here are two different methods for timing the length of an input.

Inputs in **Toggle Mode** will generate an input **one time** and one time only when the lever is pressed. Another input will not be generated until the lever is released and pressed a second time.

Inputs in **Level Mode** will generate an input **every interrupt** for as long as the lever is held down. The inputs will only stop when the lever is released. The Temporal Resolution for the interrupt can be set to 1ms or 10ms in the Hardware Configuration Utility.

Inputs in **Normal Mode** will generate an input in Toggle Mode or multiple inputs in Level Mode when the lever is **pressed**.

Inputs in **Invert Mode** will generate an input in Toggle Mode or multiple inputs in Level Mode when the lever is **released**.

Method 1 uses only one input and that input is set to Level Mode. This method is almost identical to the code in Chapter 13 except that it does not use the InputsToLevel command. The input must be set to Level Mode using the jumper or switch on the card in the interface cabinet to permanently set the input to Level Mode.

```
\ Inputs
^LeftLever = 1  \ Level Mode


\ List Data Variables Here
\  Z() = Array for Recording Response Lengths


\ List Working Variables Here
\  I = Index into Array Z
\  T = Response Timer
\  Y = Left Lever Input Counter
\ Z-Pulses Used in this Program
^Z_LeftBreak   = 3
^Z_LeftRelease = 4


DIM Z = 1000


\**************************************************
\    RESPONSE DEFINED - 20ms BREAK, 20ms RELEASE
\**************************************************
S.S.3,
S1,     \ Inputs in Level Mode generate an input "count" on each
        \ interrupt.  With a 10ms system resolution 2 counts will
        \ be reached in 20ms.  The Z^Z_LeftBreak pulse is used to
        \ signal a completed Response.  The second statement resets
        \ the counter every 20ms so that a partial Response of
        \ less than 20ms will not be counted.
  #R^LeftLever: ADD Y; IF Y >= 2 [@Response, @NoResponse]
                        @Response: SET Y = 0; Z^Z_LeftBreak ---> S2
                        @NoResponse: ---> SX
  0.02": SET Y = 0 ---> S1

S2,     \ As long as the input is broken the second statement
        \ causes a re-entry to this State.  This resets the
        \ internal 20ms timer so it never times out.  When the
        \ input is released for 20ms the timer times out and a
        \ Z^Z_LeftRelease pulse signals the release.
  0.02": Z^Z_LeftRelease ---> S1
  #R^LeftLever: ---> S2


\**************************************************
\                 RESPONSE TIMER
\**************************************************
S.S.2,
S1,
  #START: SET Z(I) = -987.987 ---> S2

S2,     \ Wait for a signal that the input has been broken
  #Z^Z_LeftBreak: SET T = 0 ---> S3

S3,     \ Wait for signal that the input has been released
        \ Record the length of the Response
  0.01": SET T = T + 0.01 ---> S3
  #Z^Z_LeftRelease: SET Z(I) = T; ADD I; SET Z(I) = -987.987 ---> S2
```

Method 2 uses two inputs. A Y-Cable (SG-216C1) is needed to send the input from the lever to two inputs on the connection panel. Then on the card in the interface cabinet the first input is set to Normal Mode (Normal Mode is the default mode) and the second input is set to Inverted Mode.

```
\ Inputs
^LeftLeverN = 1  \ Normal Mode
^LeftLeverI = 2  \ Inverted Mode


\ List Data Variables Here
\  Z() = Array for Recording Response Lengths


\ List Working Variables Here
\  I = Index into Array Z
\  T = Response Timer


DIM Z = 1000


\****************************************************
\                  RESPONSE TIMER
\****************************************************
S.S.2,
S1,
  #START: SET Z(I) = -987.987 ---> S2

S2,     \ Wait for the input to been pressed
  #R^LeftLeverN: SET T = 0 ---> S3

S3,     \ Wait the input to been released
        \ Record the length of the Response
  #R^LeftLeverI: SET Z(I) = T; ADD I; SET Z(I) = -987.987 ---> S2
  0.01": SET T = T + 0.01 ---> S3
```

How to dispense multiple pellets

```
\ Outputs
^Pellet = 3


\ A() = Control Variable with Assigned Aliases as Defined
Var_Alias Number of Pellets to Dispense = A(0)  \ Default = 2

^NumPellets = 0


DIM A = 1

\ List Working Variables Here
\  P = Pellet Dispenser Counter


\ Z-Pulses Used in this Program
^Z_Reward = 1  \ Z1 = Signal Reinforcement


\****************************************************
\             PELLET DISPENSER CONTROL
\****************************************************
S.S.2,
S1,
  0.01": SET A(^NumPellets) = 2 ---> S2

S2,     \ Wait for Reinforcement Signal
  #Z^Z_Reward: ON ^Pellet; ADD P ---> S3
```

```
S3,     \ Check if delivered desired Number of Pellets
  0.05": OFF ^Pellet;
         IF P >= A(^NumPellets) [@Done, @More]
            @Done: SET P = 0 ---> S2
            @More: ---> S4

S4,     \ Must wait a certain amount of time before
        \ trying to dispense the next pellet.  This
        \ delay can be shortened for some Pellet
        \ Dispensers.
  0.5": ON ^Pellet; ADD P ---> S3
```

How to detect when a pellet has been removed from the pellet trough

In the example below there is an IR Beam in the pellet trough wired to input 4. When the pellet is in the trough the IR Beam is broken which generates constant inputs. When the pellet is removed from the trough the inputs stop and the program will release another pellet.

This example would need a trough type pellet receptacle with pellet detector (e.g. ENV-200R2MA).

```
\ Inputs
^PelletTrough = 4  \ Level Mode

\ Outputs
^Pellet = 3


\ List Data Variables Here
\  G = Pellet Counter


\ Z-Pulses Used in This Program
^Z_End = 32  \ Signal End of Session


\***************************************************
\              PELLET DISPENSER CONTROL
\***************************************************
S.S.2,
S1,
  #START: ---> S2

S2,     \ Time Acclimation Period
        \ Dispense Initial Pellet
  5': ON ^Pellet ---> S3

S3,
  0.05":    OFF ^Pellet ---> S4
  #Z^Z_End: OFF ^Pellet ---> S1

S4,     \ As long as the Pellet is in the Trough the
        \ input will reset the 500ms timer.  Once the
        \ Pellet has been removed the 500ms timer will
        \ time out and the next Pellet will be delivered.
        \
        \ Record the number of Pellets Taken
  #R^PelletTrough: ---> S4
  0.5": ADD G; ON ^Pellet ---> S3
  #Z^Z_End: ---> S1
```

How to control at what time an output turns on/off (MED-PC V or newer)

MED-PC V introduced the #X (Time of Day) input command which makes turning on and off an output at a specified time very easy to accomplish.

```
\ Outputs
^HouseLight = 7


\ A() = Control Variable with Assigned Aliases as Defined
Var_Alias HouseLight On Time  = A(0)  \ Default = 073000
                                      \          (07:30 Hours or 7:30AM)
Var_Alias HouseLight Off Time = A(1)  \ Default = 193000
                                      \          (19:30 Hours or 7:30PM)

DIM A = 1


\*********************************************************
\ S1 - Set Default Values
\  HouseLight On Time  (073000 or 07:30 Hours or 7:30AM)
\  HouseLight Off Time (193000 or 19:30 Hours or 7:30PM)
\*********************************************************
S.S.1,
S1,     \ Time is specified in Military Time
        \ The time desired is set by specifying Hours/Minutes/Seconds
        \   Hours  Minutes  Seconds
        \    07      30       00
  0.01": SET A(0) = 073000, A(1) = 193000 ---> S2


\****************************************************
\                 HOUSE LIGHT CONTROL
\****************************************************
S.S.4,
S1,
  #START: ---> S2

S2,     \ Check if the House Light Should be Turned On
        \
        \ First Statement: If the program is started and the computer
        \ time is anywhere between A(0) and A(1), then the House Light
        \ will be turned on.
        \
        \ Second Statement: If the program is started and the computer
        \ time is not between A(0) and A(1), then this statement will
        \ execute and the House Light will not be turned on.
  A(0),A(1)#X: ON ^HouseLight ---> S3
  0.01": ---> S4

\----------------------------------------------------

S3,     \ Wait for the Time when the HouseLight Should be Turned Off
  A(1)#X: OFF ^HouseLight ---> S4

\----------------------------------------------------

S4,     \ Wait for the Time when the HouseLight Should be Turned On
  A(0)#X: ON ^HouseLight ---> S3
```

How to write Master-Yoked programs

Here are two different ways to write Master-Yoked programs.

Scenario 1

Code for the Master program

```
^Pellet = 3


\**************************************************
\                  MASTER PROGRAM
\**************************************************
S.S.1,
S1,
  #R1: ON ^Pellet; K1 ---> S2

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

Code for the Yoked program

```
^Pellet = 3


\**************************************************
\                   YOKED PROGRAM
\**************************************************
S.S.1,
S1,
  #K1: ON ^Pellet ---> S2

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

In this scenario there are two separate programs. The first program is the Master program. When a response happens it dispenses a pellet and issues a K1 pulse to the Yoked program.

The second program is the Yoked program. It waits for a K1 pulse. When the pulse is detected it dispenses a pellet.

The biggest advantage of having two separate programs is that any Box can be the Master Box.

Master Yoked Scenario 2

```
^Pellet = 3


\**************************************************
\              MASTER/YOKED PROGRAM
\**************************************************
S.S.1,
S1,
  0.01": IF BOX = 1 [@Master, @Yoked]
            @Master: ---> S2
            @Yoked:  ---> S4

\--------------------------------------------------

S2,     \ Master Code
  #R1: ON ^Pellet; K1 ---> S3

S3,
  0.05": OFF ^Pellet ---> S2

\--------------------------------------------------

S4,     \ Yoked Code
  #K1: ON ^Pellet ---> S5

S5,
  0.05": OFF ^Pellet ---> S4
```

In this scenario there is only one MSN program. The program starts by checking the special identifier BOX. The special identifier BOX will contain the value of the Box that the program is running in. If the program is running in Box 1, then BOX will equal 1. If the program is running in Box 2, then BOX will equal 2. Etc.

The advantage of this scenario is that there is only one program to maintain and update and there can be multiple subjects yoked to the subject in Box 1, but the Master Box has to always be Box 1.

How to Time Bin Data

The following program counts the total number of lever presses that happened during the entire session as well as the number of lever presses that happen during each 5 minute time segment or "Bin."

```
\ Inputs
^Lever  = 1


\ List Data Variables Here
\  D() = Responses on Lever this 5 minute Time Bin
\  L   = Total Number of Lever Responses for Entire Session


\ List Working Variables Here
\  I = Index into Array D
```

```
DIM D = 5000


\**************************************************
\                 TIME BIN CONTROL
\**************************************************
S.S.1,
S1,
  0.01": SET D(I) = -987.987 ---> S2

S2,
  #START: SET D(I) = 0, D(I+1) = -987.987 ---> S3

S3,     \ Every 5 minutes increment the index
        \ into the Time Bin Array
  5': SET I = I + 1, D(I) = 0, D(I+1) = -987.987 ---> SX


\**************************************************
\              RECORD LEVER RESPONSES
\**************************************************
S.S.2,
S1,
  #START: ---> S2

S2,     \ Record the Total Responses in variable L and
        \ the number of Responses this Time Bin in D(I)
  #R^LeftLever:  ADD L, D(I)   ---> S2
```

How to Time Stamp Responses

This example records the time in seconds since the program was started of each left and right lever press. Two arrays are used, one for each lever's responses.

```
\ Inputs
^LeftLever  = 1
^RightLever = 2


\ List Data Variables Here
\  L() = List of Left  Lever Time Presses
\  R() = List of Right Lever Time Presses
\  S   = Session Timer


\ List Working Variables Here
\  J = Index into Left  Lever Time Press Array L
\  K = Index into Right Lever Time Press Array R


DIM L = 5000
DIM R = 5000


\**************************************************
\                   SESSION TIMER
\**************************************************
S.S.1,
S1,
  0.01": SET L(J) = -987.987;
         SET R(K) = -987.987 ---> S2
```

```
S2,
  #START: SHOW 1,Session Minutes,S/60 ---> S3

S3,
  0.01": SET S = S + 0.01;
         SHOW 1,Session Minutes,S/60 ---> S3


\****************************************************
\     RECORD TIME OF LEFT/RIGHT LEVER RESPONSES
\****************************************************
S.S.2,
S1,
  #START: ---> S2

S2,     \ Record the Time of Each Response
        \ The variable S is the Session Timer, but
        \ it can also be used record the time at
        \ which an event happened
  #R^LeftLever:  SET L(J) = S; ADD J;
                 SET L(J) = -987.987 ---> S2
  #R^RightLever: SET R(K) = S; ADD K;
                 SET R(K) = -987.987 ---> S2
```

How to Program a Task Pass Criterion

Many programs start the subject off with performing a simple task and when they master that task move them onto a harder task. This example shows how to program a Task Pass Criterion so that the subject must get a certain number of the trials correct before moving onto the harder task.

This example is not a full-fledged program. The code is meant to be copied and pasted into an existing program that is running the trials. The Z-pulses for Z_Correct and Z_Incorrect need to be issued by the existing program. The existing program also needs to look for the Z_TaskComplete so that it knows when to move onto the harder task.

```
Var_Alias Task Pass Criterion (out of 10) = A(0)  \ Default = 8


\ List Working Variables Here
\  J   = Value Drawn from LIST N
\  K   = Current Index into LIST N
\  N() = LIST of Indexes for Array R
\  R() = Array for Keeping Track of the number of Correct Trials
\        from the Last 10 Trials  (1 = Correct  0 = Incorrect)
\  W   = Total Number of Correct Trials for Last 10 Trials


DIM A = 1
DIM R = 9

\ Indexes into Array R
LIST N = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
\ Z-Pulses Used in this Program
^Z_Correct      = 3  \ Signal Correct Response
^Z_Incorrect    = 4  \ Signal Incorrect Response
^Z_TaskComplete = 5  \ Signal that the Task Pass Criterion has been met


\***************************************************
\    ADD COUNTERS FOR CORRECT & INCORRECT TRIALS
\***************************************************
S.S.10,
S1,    \ Subject must get 8 out of 10 Trials correct
       \ in order to move onto the next Task
  0.01": SET A(0) = 8 ---> S2

S2,
  #Z^Z_Correct:   LIST J = N(K); SET R(J) = 1 ---> S3
  #Z^Z_Incorrect: LIST J = N(K); SET R(J) = 0 ---> S3

S3,
  0.01": SUMARRAY W = R,0,9;
         IF W >= A(0) [@TaskPass, @NotYet]
            @TaskPass: ---> Z^Z_TaskComplete
            @NotYet:   ---> S2
```

Array R is meant to hold the results from the last 10 trials.

Array N contains a list of indexes into Array R. The values are drawn from LIST N in order. When the last element (9) is drawn, then it will automatically wrap Index K back around to the beginning and start again with the first element (0).

At the end of each trial the result (1 = Correct or 0 = Incorrect) is recorded into Array R. The program then sums up the values in Array R and puts the value into Variable W. If the Task Pass Criterion has been met, then the program will send a Z_TaskComplete Z-Pulse to signal the passing results.

The result from the last 10 trials are the only values kept. It does not matter how many trials it takes the animal to meet the Task Pass Criterion; the animal must get 8 (or whatever value is chosen) out of the last 10 trials for the Task Pass Criterion to be met.

In Line Pascal

MED-PC allows for the inclusion of Pascal source code directly inside the MedState Notation Program. Almost any Pascal command can be called directly just by placing ~'s (tildes) around the Pascal code. The ~'s let the Trans program know that this is Pascal code and it should be passed onto the Pascal compiler unchanged.

Below are examples of some very commonly used in line Pascal statements.

Example 1:

```
S.S.1,
S1,
  0.01": SET A = 1000;                  \ Set the Frequency on the
         ~SetFreq(MG, BOX, A);~ ---> S2  \ ANL-926 to 1000Hz
```

Example 2:

```
S.S.5,
S1,
  5#R^Lick: ~ShockOn(MG, BOX);~ ---> S2  \ After 5 responses on the
                                         \ Lickometer turn the ENV-413
                                         \ Shocker on

S2,
  0.5": ~ShockOff(MG, BOX);~ ---> S1     \ Turn the ENV-413 Shocker off
                                         \ after 0.5"
```

Some simple math routines that can be used in line:

Example 1:

```
S2,     \ Use the Pascal "Trunc" function
  #R^Left: ADD A; ~C := Trunc(A / 2);~;
           IF C = A / 2 [@Even, @Odd]
              @Even: ---> S3
              @Odd:  ---> S4
```

Example 2:

```
S2,     \ Use the Pascal "Round" function
  #R^Left: ADD A; ~C := Round(A / 4);~ ---> S3
```

Order in which things are processed

The first things to get processed are Responses, Timed Inputs, K-pulses, and START commands. They all have the same priority.

When MED-PC processes a State, it goes from the top to the bottom. It checks the first input and sees if its criteria has been met. If it has, then it does whatever commands are in the output section and arrows to whatever is next. If the first input criteria has not been met, then it will check the next input statement to see if its criteria has been met. MED-PC will do this until it finds an input criteria that has been met or it runs out of one of the inputs listed above. If an input criteria has been met, then MED-PC does not check the rest of the inputs for that State. Z-pulses are not processed at this time.

The important point is that the first input whose criteria has been met is the code that will execute. Take for example the following code:

```
S5,
  0.001": ADD B ---> S5
  #R1:    ADD A ---> S6
```

In the above code the #R1 will never get executed because the timed input criteria will always be met first.

After processing the Responses, Timed Inputs, K-pulses, and START commands MED-PC goes back for a second pass to see if any Z-pulses were issued. Again MED-PC processes things from the top to the bottom. The first Z-pulse whose input criteria is met is the one that will get executed and the ones that follow will be ignored.

After processing the Z-pulses from the second pass MED-PC will go back for a third pass to see if any more Z-pulses were issued.

Example:

```
S10,
  #R5: Z1 ---> S11  \ When a response is detected on input 5 the program
                    \ will issue a Z1 pulse and transition to S11

S11,
  #Z1: Z2 ---> S12  \ The program is now in S11 when MED-PC does its
                    \ first pass/check for Z-pulses.  Since a Z1 was
                    \ issued in S10 the program will respond to it and
                    \ issue a Z2 pulse and transition to S12

S12,
  #Z2: Z3 ---> S13  \ The program is now in S12 when MED-PC does its
                    \ second pass/check for Z-pulses.  Since a Z2 was
                    \ issued in S11 the program will respond to it and
                    \ issue a Z3 pulse and transition to S13
```

MED-PC will allow up to nine passes with Z-pulses issuing Z-pulses. If a tenth pass was created, then MED-PC would issue an error and the chain would be terminated. All of this happens within one interrupt.

Great care needs to be taken when having Z-pulses that issue Z-pulses. If the code takes too long, then the interrupt might run long which would affect the timing of the MED-PC system.

What would the value of A be after the response at the end of the interrupt?

Example 1:

Z-pulses get processed in the same interrupt.

```
S.S.5,
S1,
  0.01": SET A = 0 ---> S2

S2,
  #R1: ADD A; Z1 ---> S2


S.S.6,
S1,
  #Z1: ADD A ---> SX      \ Value would be 2 at the end of the interrupt
```

When the Response on input 1 happens the ADD A is executed and a Z1 pulse is issued. When the program goes back to process Z-Pulses, S.S.6 is looking for a Z1 pulse. Because the Z-Pulse is processed in the same interrupt along with its code at the end of the interrupt the variable A will contain the value of 2.

Example 2:

Z-pulses get processed in the same interrupt.

```
S.S.5,
S1,
  0.01": SET A = 0 ---> S2

S2,
  #R1: ADD A; Z1 ---> S3

S3,
  #Z1: ADD A ---> S2      \ Value would be 2 at the end of the interrupt
```

When the Response on input 1 happens the ADD A is executed and a Z1 pulse is issued. The program will then transition to S3 before checking if any Z-pulses were issued. Now with the program in S3 the program checks for any Z-pulses that were issued. Since a Z1 was previously issued in S2 the Z-Pulse is processed in the same interrupt along with its code so at the end of the interrupt the variable A will contain the value of 2.

Example 3:

K-pulses get processed on the next interrupt.

```
S.S.5,
S1,
  0.01": SET A = 0 ---> S2

S2,
  #R1: ADD A; K2 ---> S2  \ Value would be 1 at the end of the interrupt


S.S.6,
S1,
  #K2: ADD A ---> SX      \ K-pulses are processed on the next interrupt
```

When the Response on input 1 happens the ADD A is executed and a K2 pulse is issued. However, because K-pulses are processed on the next interrupt its code does not execute during this interrupt. So at the end of the interrupt the variable A will contain the value of 1.

It is important to note the distinction because this can and does affect when code will run. Sometimes it is important to have code that runs during the same interrupt so a Z-pulse must be used. Sometimes it is important to let certain code finish before starting the next step so a K-pulse might be used.

K-pulses are one of the few ways to communicate between Boxes so it is also important to understand that when a K-pulse is issued the receiving Box will not execute its code until the next interrupt.

APPENDIX A | MEDSTATE NOTATION COMMANDS

This appendix lists all MedState Notation commands and is presented in detail, with syntax, comments, examples and discussion. The commands have been grouped as follows:

A.1 Input 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: OUTPUT ---> NEXT`

Where: P1 = Number, Named Constant, variable, array element, or special identifier that indicates how many times the START input must occur

Comments: Since it is unclear when specifying a count would be useful, P1 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 issues a START command. 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: OUTPUT ---> NEXT`

Where: P1 = Number, Named Constant, variable, mathematical expression, array element, or special identifier that indicates how many time the response input must occur

P2 = Number, Named 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 MedPCConfiguration.xml (MED-PC V and later) or if applicable the older MPC2INST.DTA file (MED-PC IV and earlier)

[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.

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 the 5 is omitted, as in "#R3: ON 1 ---> SX", P1 defaults to 1 and the first response will turn on Output 1. The counter for 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 SHOWs 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
```

Explicit Time Inputs: " (seconds) and ' (minutes)

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

Syntax A: `P1": OUTPUT ---> NEXT`

Syntax B: `P2': OUTPUT ---> NEXT`

Where: P1 = Number or Named Constant

P2 = Number or Named 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 Named Constant.

When a fractional minute is specified, the decimal portion corresponds to fractional minutes, not seconds. So in example C above the 5.2' equals 5 minutes 12 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" when set to run with a 0.01 (10ms) resolution.

Counters for 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: OUTPUT ---> NEXT`

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 Z = 1", 2", 3"


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

S2,
  Z(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 Z(0) = 5" and Z(1) = 10" are equivalent:

```
LIST Z = 5", 10"


#START: SET Z(0) = 5", Z(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 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
```

In the above example S.S.1, S1 sets A = 5 and then when a START command is received in S2 the value in A is changed to the internal time units via the command SET A = A * 1".

Time Inputs Less Than the Resolution Value

The minimum amount of time that a program can time is equal to the resolution parameter declared during installation (temporal resolution). A system set up with 10ms resolution cannot time events less than 0.01". On a system with 10ms resolution, the following code would increment the variable A every 10ms, not every 1ms. If the resolution was 25ms, then A would increment every 25ms. 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 DIG-704, DIG-750, DIG-703, and/or DIG-705) is used for timing; since the PC's timer cannot time in units less than 15ms, 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 for accurate timing.

Internal Representation of Time

Time values in seconds (") or minutes (') are represented internally as the numeric value multiplied by the result of (1000 / the system's resolution value in milliseconds). For example, on a system with 10ms 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 Z = 1", 2", 3"


S.S.1,
S1,
  #START: RANDI A = Z;
          SHOWEX 1,Seconds,A/1",0, 2,MED Ticks,A,0 ---> SX
```

The sample code above will produce the results "Seconds 1 MED Ticks 100," or "Seconds 2 MED Ticks 200," or "Seconds 3 MED Ticks 300" each time a START command is issued.

Time of Day Inputs: #X

Time of day values may also serve as inputs. This may be accomplished by using #X preceded by a variable containing a time of day. The time must be specified in 24-hour format, including hours, minutes and seconds.

Syntax: `P1,P2#X: OUTPUT ---> NEXT`

Where: P1 = A variable, array element, Named Constant or mathematical expression that must be a valid time in the range of 000000 (midnight) to 240000 (midnight)

P2 = OPTIONAL. A variable, array element, Named Constant or mathematical expression that must be in the range of 000000 (midnight) to 240000 (midnight). If this parameter is used, it specifies the end of a time range during which the transition should occur.

Comments: The value of P1 & P2 must be specified in HHMMSS (HH:MM:SS) in 24 hour format

The time is coming from the MED-PC clock that is located in the lower left hand corner of the main window. The MED-PC clock is dependent on the DIG-704 card which has an accuracy that is better than +/- 0.005% over 24 hours

Example A:

```
S.S.1,
S1,
  #START: SET A = 131500 ---> S2

S2,     \ Output 1 turns on at 1:15 pm
  A#X: ON 1 ---> S3

S3,     \ Turn off Output 1 at 7 pm
  190000#X: OFF 1 ---> S2
```

In Example A, the variable A is set equal to 131500 (1:15 pm) and then used in State 2 as the parameter to #X. The effect of the procedure segment is to repeatedly turn output 1 on at 1:15 pm and off at 7 pm.

Example B:

```
S.S.1,
S1,
  #START: SET A = 131500, B = 90000 ---> S2

S2,     \ Output turns on anywhere from 1:15 pm to 9:00 am
  A,B#X: ON 1 ---> S3

S3,     \ Turn off at 11 am
  110000#X: OFF 1 ---> S2
```

In Example B, the output is turned on anywhere from 1:15 pm to 9:00 am. This ensures that the output is turned on within the stated time range. In contrast, in Example A, if the program is started at 1:16 pm, the output will not be turned on until almost 24 hours later.

#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 A.2 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: OUTPUT ---> NEXT`

Where: P1 = Number, Named Constant, variable, mathematical expression, array element, or special identifier that indicates how many times the Z-pulse must occur

P2 = Number, Named 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
```

In the above example S.S.1 will wait in S1 until a Z1 pulse arrives. When the pulse arrives, Output 1 will be turned on and the program will transition to S2.

#K (K pulses as inputs)

K pulses may be found in either the input or the output section of a statement. They 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, A.2). 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 a yoked experiment using the special identifier "BOX" as the P2 parameter. #K is unrelated to the variable "K."

Syntax: `P1#KP2: OUTPUT ---> NEXT`

Where: P1 = Number, Named Constant, variable, mathematical expression, array element, or special identifier that indicates how many times the K-pulse must occur

P2 = Number, Named 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.

> NOTE: Since #K0 (results of BOX-1 when the program is loaded into Box 1) is not valid this program will only run correctly in Box 2 or higher.

```
S.S.1,
S1,
  #K(BOX-1): ON ^Shock ---> S2  \ Wait for K pulse from Master Box

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

Example B: Free or Shaping Pellet

```
S.S.1,
S1,
  #START: SET A = 10, B = 20 ---> S2

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

S3,
  0.1": OFF 1; ADD C;
        IF C >= B [@End, @Continue]
           @End:  ---> STOPSAVE  \ Session Ends After B Reinforcers
           @Cont: ---> S2
```

Example C: K used to end program execution

```
S.S.1,
S1,
  #R1: ON ^Pellet ---> S2
  #K1: ---> STOPSAVE

S2,
  0.05": OFF ^Pellet ---> S1
  #K1:   OFF ^Pellet ---> STOPSAVE
```

! (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 concise 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
```

A.2 Output Commands

Overview

Output section commands fall between the colon and transition 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  \ Turn on Outputs 1 and 5, Add 1 to X
```

Controlling Outputs

ON

ON is used to turn outputs on. Turning on an already active output has no effect. Turning on an output that does not exist has no effect.

Syntax: `INPUT: ON P1 ---> NEXT`

Where: P1 = Number, Named Constant, variable, array element, or mathematical expression

Comments: Multiple outputs may be controlled with comma separation

Examples:

Activate a stimulus light (output 1) and grain hopper (output 3) for 4 seconds:

```
^Hopper = 3


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

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

OFF

OFF turns off the specified output. It is the opposite of ON. Turning off an output that is already off has no effect. Turning off an output that does not exist has no effect.

Syntax: `INPUT: OFF P1 ---> NEXT`

Where: P1 = Number, Named Constant, variable, array element, or mathematical expression

Comments: Multiple outputs may be controlled with comma separation

Examples:

```
S1,
  #R1: OFF 1, 2, ^Hopper, 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 shut off by either OFF or LOCKOFF, but an output turned on by LOCKON may only be shut off by LOCKOFF. OFF will not deactivate an output that had been activated by LOCKON. In contrast, LOCKOFF will shut off an output, regardless of whether it was activated by ON or LOCKON.

Syntax A: `INPUT: LOCKON P1 ---> NEXT`

Syntax B: `INPUT: LOCKOFF P1 ---> NEXT`

Where: P1 = Number, Named Constant, variable, array element, or mathematical expression

Comments: Multiple outputs may be controlled with comma separation

Discussion

If a given output has been turned on by both an ON and a LOCKON command, the same output is issued, however the output is upgraded from ON status to LOCKON status. Outputs activated by LOCKON remain on even after a Box is unloaded by STOPDISCARD or STOPSAVE and are not even turned off when the computer is turned off. The only way to shut off an output activated by LOCKON is by executing a LOCKOFF or by turning off the interface cabinet.

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 LOCKONed 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."

Overlapping Inputs and Outputs

It is an acceptable practice to have a given input or output on a card assigned to more than one logical Box. For example, all Boxes might have output one mapped to output card Port 780, Output 1. This kind of output sharing could be 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 retractable levers but some of the others do not. The chambers with retractable levers are wired to six outputs (^LeftLever = 1, ^RightLever = 2, ^Pellet = 3, ^LeftLight = 4, ^RightLight = 5, ^HouseLight = 7), whereas the remaining Boxes each have four

(^Pellet = 3, ^LeftLight = 4, ^RightLight = 5, ^HouseLight = 7) 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 and ALERTOFF

Syntax A: `INPUT: ALERTON ---> NEXT`

Syntax B: `INPUT: ALERTOFF ---> NEXT`

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

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 is 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 going to "Sessions | Shut Off Beeping."

It is also possible to produce beeps whenever all Boxes are shut off by enabling the "Sessions | Beep When All Sessions are Finished" 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 A.1 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.

Output Syntax: `INPUT: ZP1 ---> NEXT`

Input Syntax: `#ZP1: OUTPUT ---> NEXT`

Where: P1 = Number, Named 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 Named 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 delivery of 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 S.S.2, S2 or S3 (depending on whether or not 30" have elapsed), driving S.S.2 to S4. Simultaneously, the Z1 also serves as an input in S.S.3, S2 causing an output channel to turn on and transition to S3. After 2", a Z2 is generated, which then serves as an input to S.S.1, S3 and S.S.2, S4 driving them both back to S2.

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

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

S3,
  #Z2: ---> S2


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

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

S3,
  #R2: Z1 ---> S4
  #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 would produce a loop that the runtime system will treat as an error on the 10th iteration.

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 S.S.2, S1 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 below acceptable levels 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 R; SHOW 1,Reinforcers,R; 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,
  #START: Z1 ---> SX
  #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 log file, 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.

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.

Output Syntax: `INPUT: KP1 ---> NEXT`

Input Syntax: `#KP1: OUTPUT ---> NEXT`

Where: P1 = Number, Named Constant, variable, array element, or mathematical expression with value in the range 1...100

Warning: It is the programmer's responsibility to ensure that P1 is in range 1...100. If this range is not observed, particularly if violated with a Named Constant, variable array element, or mathematical expression, no warning message will be generated and unpredictable problems will result when programs are running.

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

^Hopper     = 1
^HouseLight = 2
^KeyLight   = 3


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

S2,     \ 20" Mean ITI.  Tell Yoked Box when ITI is over with K1-Pulse
  1": WITHPI = 500 [ON ^KeyLight; K1] ---> S3

S3,     \ Tell Yoked Box that the Hopper is on with K2-Pulse
  #R1 ! 8": OFF ^KeyLight; ON ^Hopper; K2 ---> S4

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

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

^Hopper     = 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 Reinforcement starts
  #K2: OFF ^KeyLight; ON ^Hopper ---> S4

S4,
  4": OFF ^Hopper ---> 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 10ms 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 microseconds), the K1 will not be presented to Box 2. Instead, when the next interrupt occurs, about 10ms 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 K1-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 Ks 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:

```
S.S.2,
S1,
  #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 selects which Boxes will receive the K-Pulse via the checkboxes. 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.

Using the Special Variable "BOX" with K-Pulses

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 to communicate with Box 2, 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

^Hopper     = 1
^HouseLight = 2
^KeyLight   = 3


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

S2,     \ 20" Mean ITI.  Tell Yoked Box when ITI is over with K-Pulse
        \ equal to this Box's BOX number
  1": WITHPI = 500 [ON ^KeyLight; K(BOX)] ---> S3

S3,     \ Tell Yoked Box that the Hopper is on with K-Pulse
        \ equal to this Box's BOX number plus 1
  #R1 ! 8": OFF ^KeyLight; ON ^Hopper; K(BOX+1) ---> S4

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



\ Auto-shaping Procedure for the "Yoked" Box

^Hopper     = 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 Reinforcement starts
        \ Look for the second K-Pulse of the preceding Box
  #K(BOX): OFF ^KeyLight; ON ^Hopper ---> S4

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

A.3 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: `INPUT: ADD P1 ---> NEXT`

Where: P1 = Variable or array element

Comments: Stringing variables with comma separation permissible

Examples:

```
S1,
  1": ADD C ---> SX              \ Every 1s 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 generally used to decrement a variable or array element by 1. It performs the same function as SET A = A - 1.

Syntax: `INPUT: SUB P1 ---> NEXT`

Where: P1 = Variable or array element

Comments: Stringing variables with comma separation permissible

Examples:

```
S1,
  1": SUB C ---> SX              \ Every 1s 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: `INPUT: LIMIT P1,P2,P3 ---> NEXT`

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

P2 = A variable, array element, Named 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.

Examples:

```
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 (Note: Less than 10)
  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 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.

Syntax A: `INPUT: SET P1 = P2 ---> NEXT`

Syntax B: `INPUT: SET P1 = P2 Operator P3 ---> 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 -)

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 Named Constant is not permissible. It will not produce a translator error but will produce a Pascal compiler error during compilation, typically of the form "Variable identifier expected."

In earlier versions of 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
```

This example can now be written as:

```
SET A = 1 + 2 * 10 / 4 - 3
```

Examples:

```
1':  SET A = A * 5, B = C(I) ---> SX  \ 2 SET commands strung together
#R3: SET A = (A + B) * 5 + C ---> SX  \ Note: Multiple Operations
#START: SET A = A * 1" ---> S2
```

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 size 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: `INPUT: BIN P1,P2,P3,P4,P5,P6 ---> NEXT`

Where: P1 = Array that 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 in seconds.

P4 = The size in units of each bin or cell of the distribution. If one is recording time, then P4 is the size in seconds.

P5 = Array element, variable, Named 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, Named 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 - 5.0	3
C(3)	5.1 - 10.0	5
C(4)	10.1 - 15.0	7
C(5)	15.1 - 20.0	8
C(6)	20.1 - 25.0	9
C(7)	25.1 - 30.0	6
C(8)	30.1 - 35.0	4
C(9)	35.1 - 40.0	3
C(10)	40.1 - 45.0	2

The aforementioned example serves to "BIN" inter-response times. Sometimes it is desirable to count how many responses happen during certain Time Bin intervals. The following example illustrates this with 5 minute Time Bin intervals. Note: This does not require the BIN command.

```
\ Sample program showing how to create incremental "Time Bins"

\ C(0) = Total Responses
\ C(I) = Total Responses this 5 Minute Time Bin
\ I    = Index into the Time Bin Array C
\ S    = Session Length with a Default Value of 60 Minutes


DIM C = 20  \ Arbitrary value.  In this example only 13 elements
            \ (0 - 12) are used, one for Total Response Count {C(0)}
            \ plus 12 five minute Time Bins {C(1) - C(12)}


S.S.1,
S1,     \ Set up Default Values
  0.01": SET I = 1, S = 60 ---> S2

S2,
  #START: SET C(I+1) = -987.987 ---> S3
  1": SHOW 1,Session Length,S ---> SX

S3,     \ Increment to the next Time Bin every 5 minutes.  Use a
        \ variable and #T for a more flexible program.  See S.S.3.
  5': ADD I; SET C(I) = 0, C(I+1) = -987.987;
      SHOW 1,Bin #,I, 2,Resp Count this Time Bin,C(I) ---> SX


S.S.2,
S1,
  #START: SHOW 1,Bin #,I,       2,Resp Count this Time Bin,C(I);
          SHOW 3,Tot Resp,C(0) ---> S2

S2,     \ Response Count and Display
  #R1: ADD C(0), C(I);
       SHOW 2,Resp Count this Time Bin,C(I), 3,Tot Resp,C(0) ---> SX


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

S2,
  S#T: ---> STOPSAVE
```

A.4 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: `INPUT: Command P1 = P2,P3,P4 ---> NEXT`

Where:	Command	= One of the statistical commands listed below
	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 0, 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 has been recorded.

Statistical commands:

ARITHMETICMEAN:	Computes the sum of all numbers in the array segment divided by the number of elements in the segment.
GEOMETRICMEAN:	Computes the nth root of the product of n numbers of the array segment. There must not be any zeros in the array segment.
HARMONICMEAN:	Computes the reciprocal of the arithmetic mean of the reciprocals of the array segment. There must 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 compute the variance of an array of 100,000 elements every 10ms 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 was recorded and pass this to the statistics command.

Example:

```
\ Outputs
^Pellet = 1

\ Z-Pulses
^Reinforcement = 1
^StartRatio    = 2


DIM C = 200


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

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

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


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

S2,
  0.1": SET C(I) = C(I) + 0.1 ---> SX
  #Z^Reinforcement: ---> S3

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.

A.5 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.

Syntax A:
```
INPUT: IF P1 Operator P2 [@Label1, @Label2]
          @Label1: OUTPUT SECTION ---> NEXT1
          @Label2: OUTPUT SECTION ---> NEXT2
```

Syntax B:
```
INPUT: IF P1 Operator P2 [@Label1]
          @Label1: OUTPUT SECTION ---> NEXT1
```

Syntax C:
```
INPUT: IF P1 Operator P2 [OUTPUT SECTION] ---> NEXT1
```

Where:		
	P1	= Named Constant, number, variable, array element, or special identifier
	P2	= Named Constant, number, variable, array element, or special identifier
	Operator	= 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
	OUTPUT SECTION	= Any legal output command(s)
	NEXT1 & NEXT2	= Any legal Transition such as SX, STOPDISCARD, STOPSAVE, 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 @.

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:

Because labels are arbitrary, spelling does not need to be consistent. If the comparison evaluates as TRUE, then the immediately following statement (i.e. @Label1) is executed. If the comparison is false, then the second following statement (i.e. @Label2) is executed.

```
S.S.1,
S1,
  #R1: ADD A; OFF 1; ON 2 ---> S2

S2,
  2": OFF 2; IF A >= 100 [@True, @False]
                @True: ---> STOPSAVE
                @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: ---> STOPSAVE
                @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 hopper and goes to S2. After 2", the hopper output is turned off and variable A (the reinforcement counter) is tested. If the value of A is now greater than or equal to 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 or @F 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 [@True, @False]
                @T ---> STOPSAVE  \ Missing ":" After Label
                @F: ON 1 ---> S1
```

Example C:

Careful attention to proper syntax is important. A common mistake that will cause a translator error is to have a transition arrow after labels. For example:

```
1": IF A >= 100 [@True, @False] ---> S2
                  \ Causes Error #50: IF statements with @ alternatives
                  \   may not be immediately followed by an arrow
```

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 @1stFalse statement, 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, if the logical comparison is TRUE, then output 1 will be turned on and a transition to State 2 will occur. If the logical comparison is FALSE, then a transition to SX automatically occurs.

Example F:

A final variation on the syntax permits labels for true and false conditions to be omitted completely. When this syntax is used, output commands are enclosed in square brackets following the logical comparison. If the logical comparison is TRUE then the output 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 syntactical format requires a pair of square brackets, even if they contain no output commands. For example:

```
S.S.1,
S1,
  10": ADD A; IF A >= 10 [] ---> S2

S2,
  0.01": ON 1 ---> S3

S3,
  0.1": SET A = 0; OFF 1 ---> S1
```

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.

The order of precedence for the operators is:

Operator	Precedence
NOT	Highest
AND NOT, *, /, AND	
OR NOT, +, –, OR	
=, <>, <, <=, >, >=	Lowest

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; Trans will not accept an unequal number. 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)
```
IF A = 1 OR B = 2 [] ---> S2         \ Illegal - No parentheses
IF (A = 1) OR (B = 2) [] ---> S2 \ Legal
```

2)
```
IF A = 1) OR (B = 2 [] ---> S2       \ Illegal - Unequal number of parentheses
IF (A = 1) OR (B = 2) [] ---> S2 \ Legal
```

3)
```
IF (A = 1) OR (2) [] ---> S2         \ Illegal - Each comparison must have 2 terms
IF (A = 1) OR (B = 2) [] ---> S2 \ 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. If the comparison is FALSE, a transition to SX occurs. 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 (P1 / 10000).

Syntax A:
```
INPUT: WITHPI = P1 [@Label1, @Label2]
             @Label1: OUTPUT SECTION ---> NEXT1
             @Label2: OUTPUT SECTION ---> NEXT2
```

Syntax B:
```
INPUT: WITHPI = P1 [@Label1]
             @Label1: OUTPUT SECTION ---> NEXT1
```

Syntax C:
```
INPUT: WITHPI = P1 [OUTPUT SECTION] ---> NEXT1
```

Where:		
	P1	= Named Constant, number, variable, or array element
	Label1 & Label2	= Any text label. Note, the @ must be present before the "Label" but the label itself is purely subjective
	OUTPUT SECTION	= Any legal output command(s)
	NEXT1 & NEXT2	= Any legal Transition such as SX, STOPDISCARD, STOPSAVE, or S1...S32 (given that S1...S32 is a valid State within the same State Set)

Comments: Nesting is permissible using the first Syntax.

@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).

Labels are arbitrary, and need not match, 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:

Because labels are arbitrary, spelling does not need to be consistent. If the probability gate evaluates as TRUE, then the immediately following statement (i.e. @Label1) is executed. If the comparison is false, then the second following statement (i.e. @Label2) is executed.

```
S.S.1,
S1,
  #R1: WITHPI = 5000 [@Reinforcement, @NoReinforcement]
          @RF: ON 1 ---> S2
          @NoRF: ---> SX

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

In Example A, every response in State 1 has a pseudo-randomly determined probability of 50% (5000 / 10000) 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 on input 1 would have a 10% (1000 / 10000) probability of resulting in transition to State 2.

Example B:

A common mistake which will cause a translator error is to have a transition arrow after labels. For example:

```
1": WITHPI = 1000 [@True, @False] ---> S2
                  \ Causes Error #50: IF statements with @ alternatives
                  \   may not be immediately followed by an arrow
```

A.6 FOR Loops

FOR loops can be used to repeatedly execute MSN code. In the simplest case the code is repeated a fixed number of times. FOR loops may be placed in the output section of an MSN statement and must always be paired with #END to mark the end of the loop. In addition there must be an exit condition that indicates how many times the loop should execute.

Syntax:
```
INPUT: FOR P1 = P2 TO P3;
           OUTPUT SECTION
       #END ---> NEXT
```

Where:	P1	= Variable or array element
	P2	= Named Constant, number, variable, or array element
	P3	= Named Constant, number, variable, or array element
	OUTPUT SECTION	= Any legal output command(s)
	NEXT	= Any legal Transition such as SX, STOPDISCARD, STOPSAVE, or S1...S32 (given that S1...S32 is a valid State within the same State Set)

Comments: Nesting is permissible

Semicolon is required after upper bound parameter P3

There is no colon ":" after the #END

Output commands may not follow the #END statement transition. For example:

```
#END ---> S2
```

is legal, whereas

```
#END: ON 1 ---> S2
```

is not permissible.

Examples and Discussion

Below is a simple example that illustrates the concept even if it does not do anything particularly realistic. After 1", the loop will execute 10 times on the same clock tick. The first time the loop executes, the "loop control variable", I, will equal 0, so C(0) will be set to 0. On the next tick, I will equal 1 and C(1) will be set to 1. When I = 9, C(9) will be set to 9, the exit condition will be met and the loop will terminate.

```
DIM C = 9


S.S.1,
S1,
  1": FOR J = 0 TO 9;
         SET C(J) = J;
      #END ---> S2
```

FOR loops may be nested - that is, FOR loops may contain other FOR loops. In the example below, the first time the outer loop executes, J = 1 and the inner loop controlled by K executes twice. The outer loop then executes again, J = 2, and the inner loop executes twice (and so on). After the transition to S2, the values of elements 0 through 6 of Array C are: 0,1,2,3,4,5,6.

```
DIM C = 6


S.S.1,
S1,
  1": FOR J = 1 TO 3;
         FOR K = 1 TO 2;
            ADD A; SET C(A) = J * K;
         #END;
      #END ---> S2
```

The special transition STAY

IF statements may be enclosed within FOR loops. However, it is often necessary to use the special transition, STAY, to avoid premature exiting from the loop. Consider the code below.

```
S.S.1,
S1,
  1": FOR I = 1 TO 3;
         IF I >= 3 [@T, @F]
            @T: SHOW 1,I=,I ---> S2
            @F: SHOW 1,I=,I ---> S1
      #END ---> S4
```

The very first pass through the loop, when I = 1, there will be a transition to S1 and the 1" input timer will be reset. No matter how long the program runs, I will always have a value of 1. The general problem is that IF statements always result in a transition to either another state or to the null state so that an IF embedded in a FOR statement always results in the loop executing only once. However, the special transition, STAY, provides a solution to this problem. The STAY transition allows the loop to continue to iterate after an IF statement has executed. In the example below, the pseudo transition STAY in the false (@F) alternative of the IF statement indicates that the loop should continue to execute until some other transition occurs. In this case, transition will be to S2 when I >= 3.

```
S.S.1,
S1,
  1": FOR I = 1 TO 3;
         IF I >= 3 [@T, @F]
            @T: SHOW 1,I=,I ---> S2
            @F: SHOW 1,I=,I ---> STAY
      #END ---> S4
```

The STAY transition may also be used with #END transitions when the transition is nested within another FOR loop. This situation can occur when an IF statement is embedded in a FOR loop and another FOR loop originates within one of the IF statement alternatives. Using the STAY transition to terminate the inner FOR loop allows the outer FOR loop to execute repeatedly.

```
S.S.1,
S1,
  1": FOR I = 1 TO 10;
         IF A >= 1 [@T, @F]
            @T: ADD X ---> STAY
            @F: ADD Y;
                FOR J = 2 TO 4;
                   IF B >= C [@T, @F]
                      @T: ---> STAY
                      @F: ---> S2
                #END ---> STAY  \ J will loop through values of 2, 3, and 4
                                \   The STAY command allows the program to
                                \   remain within the outer FOR loop
      #END ---> S4
```

WITHPI functions in the same manner as IF in terms of nesting and the use of the STAY pseudo transition.

A.7 Array Functions

Data that has been entered into arrays via a LIST declaration may be accessed via the commands LIST, RANDD, and RANDI. LIST sequentially draws values from an array, RANDD randomly selects without replacement from an array, while RANDI randomly selects with replacement from an array.

RANDD draws values from the array in a Dependent manner (hence the final "D" in RANDD), each value must be drawn from the array before an element can be repeated.

RANDI draws values from the array in a Independent manner (hence the final "I" in RANDI), that means that any previous access does not affect the next access. Each element has an equal chance of being selected with each RANDI call.

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.

Defining the Array.

Syntax A: `LIST P1 = P2, P3, ..., Pn`

Where:	P1	= The name of the array to be declared (A…Z)
	P2, P3, ..., Pn	= Number

Comments: The array must be defined before the first State Set.

A list declaration may be on more than one line, but each line must end on a comma. The last line should not end on a comma.

Selecting elements from the Array.

Syntax B: `INPUT: LIST P1 = P2(P3) ---> NEXT`

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: The value of P3 is automatically incremented by the LIST command.

Note: 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 a LIST or DIM statement.

Stringing is permissible.

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 Z = 1, 2, 3


S.S.1,
S1,
  1": LIST B = Z(I); ON B ---> S2

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

In the first LIST statement above, an array named Z 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 = Z(I)" sets B equal to the value of element I of array Z. Since all variables are automatically set to 0 at the beginning of program execution, the value of B is set equal to Z(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 Z(I) to B, 1 is automatically added to the value of I so that the next time the list statement is executed, the array index (I) for array Z 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 (I) is reset back to 0.

Example B:

```
LIST Z =  5, 10, 15, 20, 25, 30,  \ Note that each line must
         35, 40, 45, 50, 55, 60,  \   end in a comma except
         65, 70, 75, 80, 85, 90   \   for the last one
```

RANDD

RANDD is similar to LIST (as an output) and is used to automatically select data from an array created with the LIST command. The difference between the LIST and RANDD command is that while LIST pulls its values from the array sequentially, RANDD pulls them randomly without replacement. Think of a bucket full of numbers. When a number is drawn from the bucket, that number is now considered used and is set aside making it no longer available. That number cannot be drawn from the bucket again until all numbers have been drawn, at which point the bucket is refilled with all of the available numbers.

RANDD draws values from the array in a Dependent manner (hence the final "D" in RANDD), each value must be drawn from the array before an element can be repeated.

Syntax: `INPUT: RANDD P1 = P2 ---> NEXT`

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 Z = 1, 2, 3


S.S.1,
S1,
  1": RANDD B = Z; ON B ---> S2

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

RANDI

RANDI is closely related to RANDD with the difference being in that RANDI randomly selects from an array, but with replacement. Using the bucket analogy again, when a number is drawn from the bucket, that number is immediately put back into the bucket and becomes available to be drawn again.

RANDI draws values from the array in a Independent manner (hence the final "I" in RANDI), that means that any previous access does not affect the next access. Each element has an equal chance of being selected with each RANDI call.

Syntax: `INPUT: RANDI P1 = P2 ---> NEXT`

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.

Unlike the RANDD command, there is no limit to the number of elements that may be selected from an array declared with the LIST command.

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: 2, 2, 1, 3, 2, 1, 3, 3, 1, ...

```
LIST Z = 1, 2, 3


S.S.1,
S1,
  1": RANDI B = Z; 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, no elements will be copied and a MED-PC runtime error message will be generated.

Syntax: `INPUT: COPYARRAY P1,P2,P3 ---> NEXT`

Where: P1 = The source array from which data will be copied

P2 = The destination array into which data will be copied

P3 = The number of elements to copy from P1 into P2

Comments: P1 & P2 must be the name of an array

P3 may be a variable, Named 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 B to Array D every 5 minutes and then printed. This technique ensures that all data are printed from the same moment (see description of PRINT command). This also prints the cumulative or absolute total every 5 minutes.

```
DIM B = 2
DIM D = 2


PRINTVARS = D


S.S.1,
S1,
  #R1: ADD B(0) ---> SX
  #R2: ADD B(1) ---> SX
  #R3: ADD B(2) ---> SX


S.S.2,
S1,
  5': COPYARRAY B,D,3; PRINT ---> SX  \ Copy 3 elements (0, 1, 2)
                                      \   from B to D 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: `INPUT: ZEROARRAY P1 ---> NEXT`

Where: P1 = Must be the name of an array declared by LIST or DIM.

Example:

```
DIM B = 2
DIM D = 2


PRINTVARS = D


S.S.1,
S1,
  #R1: ADD B(0) ---> SX
  #R2: ADD B(1) ---> SX
  #R3: ADD B(2) ---> SX


S.S.2,
S1,
  5': COPYARRAY B,D,3; ZEROARRAY B; PRINT ---> SX
```

In this example, the current cumulative response totals on inputs 1-3 are transferred from Array B to Array D every 5 minutes, however the counter in Array B is then zeroed before Array D is printed. This example prints the relative total every 5 minutes.

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: `INPUT: INITCONSTPROBARR P1,P2 ---> NEXT`

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 Z = 1, 2, 3, 4, 5, 6, 7


S.S.1,
S1,
  1": INITCONSTPROBARR Z,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 Z = 1, 2, 3, 4, 5, 6, 7


S.S.1,
S1,
  1": INITCONSTPROBARR Z,10 ---> S2

S2,
  1": RANDD A = Z;
      SET A = A * 1";
      ~A := Round(A);~ ---> SX  \ Eliminate the Fraction of a "Tick"
```

A.8 Displaying, Printing, and Saving Data

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 Named 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: `INPUT: SHOW P1,Label,P3 ---> NEXT`

Where: P1 = Whole number in range 1…200 expressed as an array element, variable, Named Constant, or number indicating the position to use in SHOW display.

Label = 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, Named Constant, array element, or mathematical expression to be displayed in position P1.

Comments: There may be up to 200 SHOWs per Box.

The SHOW command is a secondary function and may be delayed while Boxes are processed; data is being saved, or being sent to print the manager.

Stringing is permissible.

Do not use the following characters in the descriptive label portion of a SHOW statement: { } \ ; " ' or a comma.

Numbers are normally displayed with 2 digits of precision to the right of the decimal point.

Use SHOWEX to specify more or less digits of precision.

Example A:

The following example displays the label "Session Time" in the fifth position on the SHOW screen. Every 10ms the value of S will increment by 0.01 and will be reflected on the screen.

```
S.S.1,
S1,
  0.01": SET S = S + 0.01;
         SHOW 5,Session Time,S ---> SX
```

Example B:

This example illustrates the stringing of parameters in one SHOW statement, the use of a constant number (5.01), the use of a Named Constant (^Value), and the use of a mathematical expression.

```
^Value = 5


LIST A = 2, 5


S.S.1,
S1,
  1": SHOW 1,Value1,5.01, 2,Value2,^Value, 3,Math,A(0)+A(1) ---> SX
```

When loaded to Box 1 this example creates the following display:

```
1) VALUE1   5.01     VALUE2   5.00     MATH   7.00
```

SHOWEX

SHOWEX may be used to display data on the screen while a procedure is running. This command is an extended version of SHOW as it has the ability to control the number of digits displayed after the decimal point. Each SHOWEX command takes four arguments. The first is the screen position (1 - 200), the second is the descriptive label, the third is a number, variable or Named Constant to be displayed on the screen, and the fourth argument specifies the number of digits displayed after the decimal point.

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: `INPUT: SHOWEX P1,Label,P2,P3 ---> NEXT`

Where: P1 = Whole number in range 1…200 expressed as an array element, variable, Named Constant, or number indicating the position to use in SHOW display.

Label = 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.

P2 = Number, variable, Named Constant, array element, or mathematical expression to be displayed in position P1.

P3 = Number of digits of precision to display to the right of the decimal point. MED-PC will display up to 8 digits after the decimal point. This value may be either a number, variable, Named Constant, or array element.

Comments: There may be up to 200 SHOWEXs per Box.

The SHOWEX command is a secondary function and may be delayed while Boxes are processed, data is being saved, or being sent to the print manager.

Stringing is permissible.

Numbers in MED-PC have 17 digits of precision, including the decimal point. If this total is exceeded then digits following the decimal point will likely be truncated and/or rounded. For example, 1234567890123.5678 will not display the very last digit "8" and the "7" might be rounded up. In normal circumstances this limitation should not be relevant.

Do not use the following characters in the descriptive label portion of a SHOWEX statement: { } \ ; " ' or a comma.

Example A:

The following example requires a 1ms resolution to have been set with the Hardware Configuration Utility. It displays the label "Session Time" in the tenth position on the show screen. Every 1ms the value of S will increment by 0.001 and will be reflected on the screen. The value will be displayed with three digits of precision to the right of the decimal point.

```
S.S.1,
S1,
  0.001": SET S = S + 0.001;
          SHOWEX 10,Session Time,S,3 ---> SX
```

Example B:

This example illustrates the stringing of parameters in one SHOWEX statement and what can happen when using the fourth parameter for specifying the digits of precision to the right of the decimal point.

```
^Value = 5


LIST A = 2.22, 5.71


S.S.1,
S1,
  1": SHOWEX 1,Value_1,5.09,1,   2,Value_1,5.09,2;
      SHOWEX 3,Value_2,^Value,0, 4,Math,A(0)+A(1),1 ---> SX
```

When loaded to Box 1 this example creates the following display:

```
1) VALUE_1   5.1     VALUE_1   5.09     VALUE_2   5     MATH   7.9
```

For the first SHOWEX field a constant number (5.09) is used. Notice that with only one digit of precision that the number was rounded up to 5.1

For the second SHOWEX field the same constant number was used with two digits of precision so the number was displayed fully.

For the third SHOWEX field the Named Constant (^Value) was displayed with zero digits after the decimal point.

For the fourth SHOWEX field a mathematical expression was used with only one digit of precision so the value was rounded down to 7.9.

Updating the SHOW Display

SHOW and SHOWEX commands are not displayed in real-time. When a Box issues a SHOW or a SHOWEX 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 SHOW/SHOWEX commands were issued. In practice, the SHOW/SHOWEX 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 a SHOW/SHOWEX output that appears on the screen as a discontinuously incrementing counter (perhaps incrementing as 3, 7, 8, 12, etc.):

Example A:

```
^Hopper = 3


S.S.1,
S1,
  20#R1: ADD Y; SHOW 2,Reinforcers,Y; ON ^Hopper ---> S2

S2,
  2": OFF ^Hopper ---> 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 subject 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.

```
^Hopper = 3


S.S.1,
S1,
  #START: SHOW 1,Responses,X, 2,Reinforcers,Y ---> S2

S2,
  20#R1: ADD Y; SHOW 2,Reinforcers,Y; ON ^Hopper ---> S3

S3,
  2": OFF ^Hopper ---> S1


S.S.2,
S1,
  #R1: ADD X; SHOW 1,Responses,X ---> SX
```

CLEAR - Remove data from SHOW display of runtime screen

CLEAR works in conjunction with SHOW and SHOWEX. As its name suggests, CLEAR blanks out SHOW and SHOWEX command outputs. 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 Named Constants as arguments.

Syntax: `INPUT: CLEAR P1,P2 ---> NEXT`

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

P2 = Number, variable, array element, or Named 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

P1 is the lowest show field to clear

P2 is the highest show field to clear

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, PRINTCOLUMNS, 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: `INPUT: PRINT ---> NEXT`

Comments: 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 variable A and possibly B 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" ---> STOPDISCARD
```

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 is 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 is 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 its 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 ---> STOPSAVE  \ Values will stop changing
                            \   when this line executes
```

2) Transfer the data to be printed to special arrays or variables that won't be updated during the time between when a printing request is issued and the generation of the printout happens. 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);  \ Freeze the data and Print
       PRINT ---> SX                     \   the values by issuing a
                                         \   K1 pulse from the keyboard
  1": ADD A(0), Z(2000) ---> SX
```

3) Transfer an entire array with the COPYARRAY command.

```
DIM B = 2
DIM D = 2


PRINTVARS = D


S.S.1,
S1,
  #R1: ADD B(0) ---> SX
  #R2: ADD B(1) ---> SX
  #R3: ADD B(2) ---> SX


S.S.2,
S1,
  5': COPYARRAY B,D,3; PRINT ---> SX  \ Copy 3 elements (0, 1, 2)
                                      \   from B to D and then Print
```

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.

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 a request to print the Box's data is still in the print queue.

Example:

```
60': PRINT ---> STOPDISCARD  \ Data will print, even though transition
                             \   is to STOPDISCARD because the print
                             \   request was issued before the
                             \   STOPDISCARD command
```

FLUSH

Users upgrading from MED-PC IV might remember the FLUSH command. This command has been deprecated. FLUSH has been replaced with the command WRITE. Programs that contain the old command will still continue to compile and run, however, it is strongly recommended that the new WRITE command is used instead.

WRITE

Under typical scenarios, MSN programs are designed such that data is written to the disk as the result of a transition to STOPSAVE. However, this command also terminates execution of the program. Under some circumstances it might be desirable to write data to the disk and then continue execution of the program. This may be accomplished by placing the WRITE command in the output section of an MSN statement.

Syntax: `INPUT: WRITE ---> NEXT`

Comments: WRITE takes no arguments

The data is written to disk without any user intervention, similar to the effects of STOPSAVE, 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 writes will cause data to be appended to the same file - multiple files will not be created, regardless of the number of writes 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': WRITE ---> 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.

A.9 Miscellaneous Commands

INPUTSTOLEVEL and INPUTSTOINVERT

The IC-124 OmniCtrl cards do not have any jumpers or switches for setting the Toggle/Level Mode or Normal/Invert Mode for inputs on the cards. When using these cards there are two special MedState Notation commands that must be used in order to change the default settings for the inputs.

While the below definitions and examples reference a lever the information remains the same if the input was changed to an IR Beam or any other type of input.

Inputs in **Toggle Mode** will generate an input one time and one time only when the lever is pressed. Another input will not be generated until the lever is released and pressed a second time.

Inputs in **Level Mode** will generate an input every interrupt for as long as the lever is held down. The inputs will only stop when the lever is released. The Temporal Resolution for the interrupt can be set to 1ms or 10ms in the Hardware Configuration Utility.

Inputs in **Normal Mode** will generate an input in Toggle Mode or multiple inputs in Level Mode when the lever is pressed.

Inputs in **Invert Mode** will generate an input in Toggle Mode or multiple inputs in Level Mode when the lever is released.

The default settings for all inputs on the IC-124 OmniCtrl card are Toggle Mode and Normal Mode.

NOTE: These two commands should always be used before the START command. If a program changes several inputs to Level mode and/or Invert mode, then it is possible for the hardware to not complete the change before the program tries to use it.

INPUTSTOLEVEL (MED-PC VI OR NEWER)

When MED-PC is started or when a program is finished running in a Box all inputs that were assigned to that Box with the Hardware Configuration Utility are set back to the default setting of Toggle Mode. If a program needs an input to be in Level Mode, then that program must use the INPUTSTOLEVEL command.

Syntax: `INPUT: INPUTSTOLEVEL P1, P2, ..., Pn ---> NEXT`

Where: P1, P2, ..., Pn = Number

Comments: Number should be an input that has been declared in the Hardware Configuration Utility and that the current program is using.

Number must be a whole number in the range 1...80.

This command has no effect on a card that is not a IC-124 OmniCtrl. The hardware will ignore it, so it is safe to have one Box that is using a DIG-716 SmartCtrl card or a DIG-712 SuperPort card and another Box with a IC-124 OmniCtrl Card. The same program with this command can be used in both Boxes.

Real code may look like this:

```
S.S.1,
S1,
  0.01": INPUTSTOLEVEL 1,2,3,4,5,6,7,8 ---> S2
```

This command will set inputs 1 through 8 to Level Mode in the Box running the code.

INPUTSTOINVERT (MED-PC VI OR NEWER)

When MED-PC is started or when a program is finished running in a Box all inputs that were assigned to that Box with the Hardware Configuration Utility are set back to the default setting of Normal Mode. If a program needs an input to be in Invert Mode, then that program must use the INPUTSTOINVERT command.

Syntax: `INPUT: INPUTSTOINVERT P1, P2, ..., Pn ---> NEXT`

Where: P1, P2, ..., Pn = Number

Comments: Number should be an input that has been declared in the Hardware Configuration Utility and that the current program is using.

Number must be a whole number in the range 1...80.

This command has no effect on a card that is not a IC-124 OmniCtrl. The hardware will ignore it so it is safe to have one Box that is using a DIG-716 SmartCtrl card or a DIG-712 SuperPort card and another Box with a IC-124 OmniCtrl card. The same program with this command can be used in both Boxes.

Real code may look like this:

```
S.S.1,
S1,
  0.01": INPUTSTOINVERT 1,2,3,4,5,6,7,8 ---> S2
```

This command will set inputs 1 through 8 to Invert Mode in the Box running the code.

How to reset the inputs to Toggle Mode and Normal Mode

Once a MSN program has set its inputs to Level Mode or Invert Mode the inputs will stay in that mode until one of three things happen.

1. The current program running in the Box is unloaded with one of the MSN STOP commands or the User ends the program manually by going to the Sessions | Stop Box menu.
2. MED-PC is closed.
3. The interface cabinet containing the IC-124 OmniCtrl cards is turned off and on.

DATE and TIME

These commands return information about the computer's Date and Time. Having access to this information may be useful in IF statements for setting experimental parameters on the basis of Time and/or Date information.

Syntax: `INPUT: DATE P1,P2,P3 ---> NEXT`

Where: P1 = Variable or array element

P2 = Variable or array element

P3 = Variable or array element

Information returned by DATE into the parameters:

P1 = Month
P2 = Day
P3 = Year

Syntax: `INPUT: TIME P1,P2,P3 ---> NEXT`

Where: P1 = Variable or array element
P2 = Variable or array element
P3 = Variable or array element

Information returned by TIME into the parameters:

P1 = Hours (24 hour format)
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 10, 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
```

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: `INPUT: GETVAL P1 = P2,P3 ---> NEXT`

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

Examples:

```
S.S.1,
S1,
  1": GETVAL A = 2,B ---> SX  \ Set A equal to the value of
                              \ variable B in Box 2


S.S.5,
S1,
  1": GETVAL A = 8,D(10) ---> SX  \ Set A equal to the value of
                                  \ array element D(10) in Box 8
```

Sometimes it is desirable to get a set of variables value from a prior day's run for the same Box. See Appendix C for an advanced programming technique to call a set of functions in the BACKPROC.PAS for this purpose.

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: `INPUT: BKGRND P1 ---> NEXT`

Where: P1 = A number from 1 to 10 inclusive

Example:

```
S.S.1,
S1,
  1": BKGRND 5 ---> S2
```

The above example will run background procedure 5. For more information on the background procedures please see Appendix C.

A.10 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 the Using the Special Variable "BOX" with K-Pulses section). 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 "Load Box..." 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 2022, STARTYEAR will equal 22[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.

[7] If a four digit year is desired, then use the Y2KCOMPLIANT command. For a discussion on this command, see the "Commands that come before the first State Set" section of this Appendix.

ENDMONTH
ENDDATE
ENDYEAR
ENDHOURS
ENDMINUTES
ENDSECONDS

These variables are set equal to the Date and Time when the Box's execution is terminated with STOPDISCARD or STOPSAVE. 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 (see next page). 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 current time as a 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, March 1, 2022 would be expressed as 220301.

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 10ms system, BTIME increments 100 times per second. On a 1ms system, BTIME increments 1000 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.

A.11 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 Subject 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 subject is being punished for responding during the ITI.

STOPKILL

Users upgrading from MED-PC IV might remember the STOPKILL command. This command has been deprecated. STOPKILL has been replaced with the command STOPDISCARD. Programs that contain the old command will still continue to compile and run, however, it is strongly recommended that the new STOPDISCARD command is used instead.

STOPDISCARD

This command is placed following a transition arrow and will cause the procedure to immediately stop executing. Any outputs currently turned on get shut off immediately unless they were turned on by the LOCKON command. In addition, the Box's status lines on the monitor are cleared. Any and all data from procedures terminated by STOPDISCARD is not recoverable - the data is not placed in the dump queue. STOPDISCARD automatically performs the same function as its manual equivalent on the "Stop Box..." window.

Syntax: `INPUT: OUTPUT ---> STOPDISCARD`

Example:

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

S2,
  2": OFF 1; IF A = 50 [] ---> STOPDISCARD
```

STOPABORT and STOPABORTFLUSH

Users upgrading from MED-PC IV might remember the STOPABORT and STOPABORTFLUSH commands. These commands have been deprecated. STOPABORT and STOPABORTFLUSH have both been replaced with the command STOPSAVE. STOPABORT has no direct equivalent and is now interpreted as STOPSAVE. Programs that contain the old commands will still continue to compile and run, however, it is strongly recommended that the new STOPSAVE command is used instead.

STOPSAVE

STOPSAVE is identical to STOPDISCARD in that it is a transition that turns off all outputs, except those that were turned on with the LOCKON command, and stops procedure execution. Additionally, this command causes the data to be written to disk automatically. File format is determined by the scheme selected by going to "Data | File Options..." and any DISK commands (DISKVARS, DISKCOLUMNS, DISKFORMAT, and DISKOPTIONS) used prior to State Set 1. File naming defaults to the scheme selected by going to "Data |File Options..." unless a custom file name is assigned from the "Load Box..." window.

Executing STOPSAVE will cause all data to be immediately written to the disk without any user intervention.

Syntax: `INPUT: OUTPUT ---> STOPSAVE`

Example:

```
S.S.1,
S1,
  #R1: ADD A ---> SX
  #R2: ADD B ---> SX


S.S.2,
S1,
  60' ---> STOPSAVE
```

The special transition STAY

STAY is a very special transition that is only used with a FOR loop that contains an IF or WITHPI statement. IF and WITHPI statements may be enclosed within FOR loops. However, it is often necessary to use the special transition, STAY, to avoid premature exiting from the loop. Consider the code below.

```
S.S.1,
S1,
  1": FOR I = 1 TO 3;
         IF I >= 3 [@T, @F]
            @T: SHOW 1,I=,I ---> S2
            @F: SHOW 1,I=,I ---> S1
      #END ---> S4
```

The very first pass through the loop, when I = 1, there will be a transition to S1 and the 1" input timer will be reset. No matter how long the program runs, I will always have a value of 1. The general problem is that IF statements always result in a transition to either another state or to the null state so that an IF embedded in a FOR statement always results in the loop executing only once. However, the special transition, STAY, provides a solution to this problem. The STAY transition allows the loop to continue to iterate after an IF statement has executed. In the example below, the pseudo transition STAY in the false (@F) alternative of the IF statement indicates that the loop should continue to execute until some other transition occurs. In this case, transition will be to S2 when I >= 3.

```
S.S.1,
S1,
  1": FOR I = 1 TO 3;
         IF I >= 3 [@T, @F]
            @T: SHOW 1,I=,I ---> S2
            @F: SHOW 1,I=,I ---> STAY
      #END ---> S4
```

The STAY transition may also be used with #END transitions when the transition is nested within another FOR loop. This situation can occur when an IF statement is embedded in a FOR loop and another FOR loop originates within one of the IF statement alternatives. Using the STAY transition to terminate the inner FOR loop allows the outer FOR loop to execute repeatedly.

```
S.S.1,
S1,
  1": FOR I = 1 TO 10;
         IF A >= 1 [@T, @F]
            @T: ADD X ---> STAY
            @F: ADD Y;
                FOR J = 2 TO 4;
                    IF B >= C [@T, @F]
                       @T: ---> STAY
                       @F: ---> S2
                #END ---> STAY  \ J will loop through values of 2, 3, and 4
                                \   The STAY command allows the program to
                                \   remain within the outer FOR loop
      #END ---> S4
```

WITHPI functions in the same manner as IF in terms of nesting and the use of the STAY pseudo transition.

A.12 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" cannot 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 total 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 11 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.

Defining the Array

Syntax A: `LIST P1 = P2, P3, ..., Pn`

Where: P1 = The name of the array to be declared (A…Z)

P2, P3, ..., Pn = Number

Comments: The array must be defined before the first State Set.

A list declaration may be on more than one line, but each line must end on a comma. The last line should not end on a comma.

Selecting elements from the Array

Syntax B: `INPUT: LIST P1 = P2(P3) ---> NEXT`

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: The value of P3 is automatically incremented by the LIST command.

Note: 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 a LIST or DIM statement.

Stringing is permissible.

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 Z = 1, 2, 3

S.S.1,
S1,
  1": LIST B = Z(I); ON B ---> S2

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

In the first LIST statement above, an array named Z 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 = Z(I)" sets B equal to the value of element I of array Z. Since all variables are automatically set to 0 at the beginning of program execution, the value of B is set equal to Z(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 Z(I) to B, 1 is automatically added to the value of I so that the next time the list statement is executed, the array index (I) for array Z 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 (I) is reset back to 0.

Example B:

```
LIST Z =  5, 10, 15, 20, 25, 30,  \ Note that each line must
         35, 40, 45, 50, 55, 60,  \   end on a comma except
         65, 70, 75, 80, 85, 90   \   for the last one
```

SEALED_ARRAY

Arrays declared with the SEALED_ARRAY command function like normal arrays except that the array is automatically truncated after the last non-zero value when a printout is generated or a data file is saved.

Syntax: `SEALED_ARRAY 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 total number of elements is always P2 + 1. The element's indices range from 0...P2

It is often difficult to predict how large a data array should be. Take for example a program recording an IRT Data Array. Depending on how long the program runs and how the subject responds there could be 50 responses or 150 responses. Thus, it is difficult to know how many elements to reserve for an array declared with the DIM command. A typical approach to this issue is to declare a very large array and then seal the array with -987.987 in order to prevent long printouts and large data files.

Example A:

```
DIM C = 49


S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX


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

S2,
  #R^LeftLever: IF I > 49 [@ArrayFull, @Continue]
                   @Full: ---> S1
                   @Cont: SET C(I) = T, T = 0; ADD I;
                          IF I > 49 [@ArrayFull, @SealArray]
                             @Full: ---> S1
                             @Seal: SET C(I) = -987.987 ---> SX
```

The SEALED_ARRAY command is a newer alternative to using -987.987 for truncating arrays.

Example B:

```
SEALED_ARRAY C = 49


S.S.4,  \ Increment Time (T) in 0.01 second intervals
S1,
  #START: ---> S2

S2,
  0.01": SET T = T + 0.01 ---> SX


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

S2,
  #R^LeftLever: IF I > 49 [@ArrayFull, @Continue]
                 @Full: ---> S1
                 @Cont: SET C(I) = T, T = 0; ADD I ---> SX
```

With the above code if the responses were as follows:

```
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
```

Only the first 42 numbers will be saved to the data file or printed on a printout:

```
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
```

Declaring Named Constants

The clarity of MSN programs may be enhanced through the use of Named Constants. Named Constants are user-defined meaningful names that may be used in place of whole numbers in MSN programs. Named Constants are particularly useful for providing logical names for input and output numbers and Z pulses. Named Constant names must always begin with a carat '^.'

Syntax: `^NAMED_CONSTANT = P1`

Where:	NAMED_CONSTANT	= The name being assigned
	P1	= The whole number being assigned to the Named Constant

Comments: Named Constants must be declared as having an integer value. For example, **^Hopper = 3.1** is illegal.

As Named Constants are represented as integer (whole) numbers, it is illegal to attempt to assign the value of a variable to a Named Constant during program execution.

Up to 2000 Named Constants may be declared in a single procedure and Named Constants are restricted to holding values within the range of –2,147,483,647 to 9,223,372,036,854,775,807.

It is acceptable to assign time values to Named Constants (e.g., ^FIVal = 30").

The following code illustrates several uses of Named Constants.

```
\ Inputs
^LeftPeck = 1

\ Outputs
^Hopper     = 3
^HouseLight = 7


DIM C = 5

\ Offsets into C Array
^RFS = 0  \ Counter 0 will count Reinforcers


\ Z-Pulses
^RFBegin = 1
^RFEnd   = 2


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

S2,
  10#R^LeftPeck: ON ^Hopper; ADD C(^RFS); Z^RFBegin ---> S3

S3,
  2": OFF ^Hopper; Z^RFEnd ---> S2
```

Named Variables (VAR_ALIAS Command)

MED-PC allows the user to create names for variables that will appear in the Session Parameters window of the Wizard for Loading Boxes and the Change Variables window of the runtime system. This allows users to set parameter values using meaningful labels.

Syntax: `VAR_ALIAS P1 = P2`

Where: P1 = A descriptive label for the variable or array element

P2 = The variable or array element

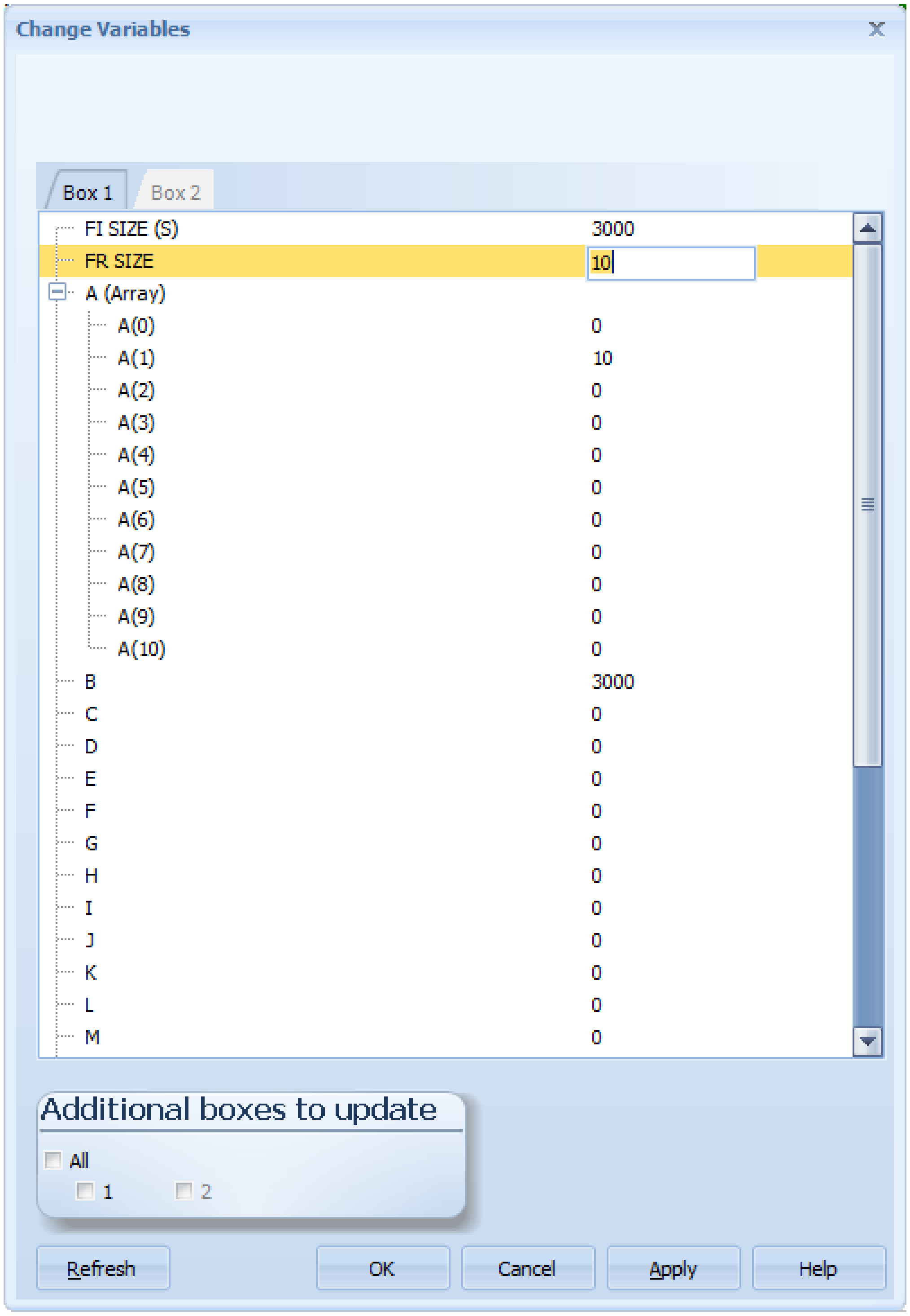
Comments: Variable aliases do not have any use within the body of MSN programs - they are simply directives placed before the first State Set that establish meaningful aliases (essentially synonyms) for program variables.

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 A(1).

```
\ Concurrent FR FI


VAR_ALIAS FR Size     = A(1)  \ Default = 10
VAR_ALIAS FI Size (s) = 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
```

Figure A.1 is an example of the standard dialog used in the runtime system to view and manipulate variables, Named Variables, and array elements. It was produced by the sample code above.

Figure A.1 - Change Variables Dialog with VAR_ALIAS



FI SIZE (S)	3000
FR SIZE	10
A (Array)	
A(0)	0
A(1)	10
A(2)	0
A(3)	0
A(4)	0
A(5)	0
A(6)	0
A(7)	0
A(8)	0
A(9)	0
A(10)	0
B	3000
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0
M	0

Equated Variables (EQUATE Command)

The EQUATE command may be placed before the beginning of the first State Set and is used to define meaningful names for variables that may be used within the MSN program.

Syntax: `EQUATE P1 = P2`

Where: P1 = A descriptive name at least two characters in length. Allowed characters: A...Z, a...z, _, 0...9

P2 = A variable, an array element, or an array name

For example, the following will work:

```
EQUATE Total_Responses = A


S.S.1,
S1,
  #R1: ADD Total_Responses ---> SX
```

This statement would be equivalent to:

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

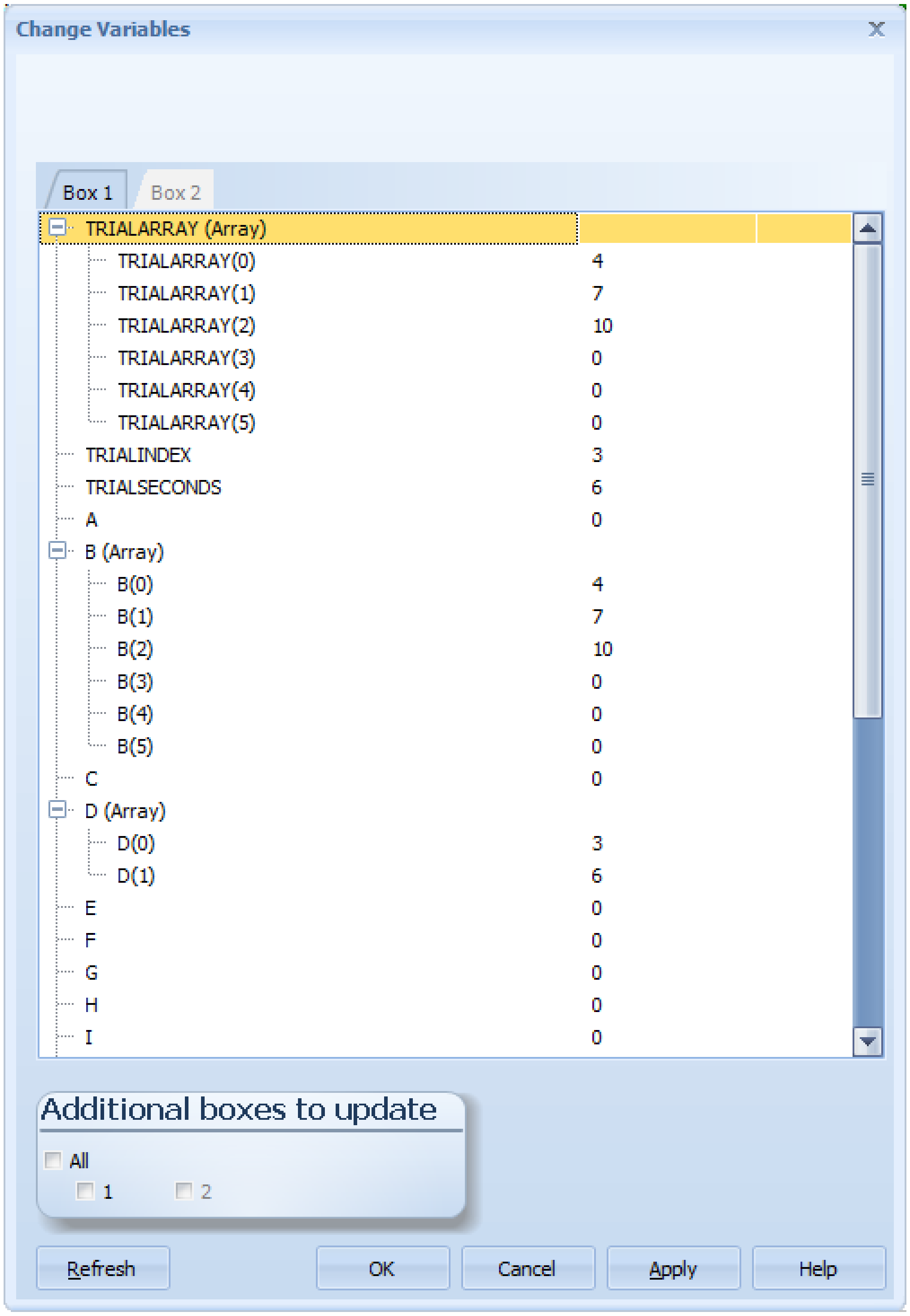
but it has the advantage of being easier to comprehend and maintain. In both cases, it is actually "A" that is incremented and a variable named Total_Responses will not appear on printouts or in data files. However, the Equated variable name is visible in the Change Variables dialog of the runtime system.

Examples:

```
EQUATE TrialArray   = B
EQUATE TrialIndex   = D(0)
EQUATE TrialSeconds = D(1)


DIM B = 5
DIM D = 1


S.S.1,
S1,
  #R1: SET TrialArray(TrialIndex) = TrialSeconds;
       SET TrialSeconds           = 0;
       ADD TrialIndex ---> S1
  1": ADD TrialSeconds ---> SX
```

Figure A.2 is an example of the standard dialog used in the runtime system to view and manipulate variables, Equated Variables, and array elements. It was produced by the sample code above.

Figure A.2 - Change Variables Dialog with EQUATE



TRIALARRAY (Array)	
TRIALARRAY(0)	4
TRIALARRAY(1)	7
TRIALARRAY(2)	10
TRIALARRAY(3)	0
TRIALARRAY(4)	0
TRIALARRAY(5)	0
TRIALINDEX	3
TRIALSECONDS	6
A	0
B (Array)	
B(0)	4
B(1)	7
B(2)	10
B(3)	0
B(4)	0
B(5)	0
C	0
D (Array)	
D(0)	3
D(1)	6
E	0
F	0
G	0
H	0
I	0

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
```

Comments: 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 State Set. In the following example, StopActivityChamber executes after STOPSAVE.

```
\ Stop the Activity Monitor chamber after
\ the Box has finished running.
FINALIZATIONCODE = ~StopActivityChamber(BOX);~


S.S.1,
S1,
  10': ---> STOPSAVE
```

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 and LIBRARY

Some portions of code tend to be used repeatedly. Code macros are code snippets that are defined and given a name prior to the first State Set 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 (*.LIB)" from the "Save as type:" dropdown list of the Save As dialog box. Library files use ".LIB" as their filename extension. 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 A: `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 B: `INPUT: %P1% ---> NEXT`

Where: P1 = The name of a macro. The name must be surrounded by "%" signs.

Example:

```
S.S.5,
S1,
  #Z2: %Outputs% ---> S2  \ Equivalent to #Z2: ON 1, 2, 3 ---> S2
```

Including code from a library file

Syntax C: `LIBRARY P1`

Where: P1 = The name of a file containing DEFINEMACRO statements.

Comments: 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.

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:\MED-PC\MPC\Constants.lib
```

Examples and Discussion:

Example A:

When issuing a reinforcer, a tone and the dispenser may be turned on and the House Light 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.

```
^Pellet     = 3
^Tone       = 6
^HouseLight = 7


DEFINEMACRO DeliverRF  = {OFF ^HouseLight; ON  ^Pellet, ^Tone}
DEFINEMACRO RFFinished = {ON  ^HouseLight; OFF ^Pellet, ^Tone}


S.S.1,
S1,
   10#R1: %DeliverRF% ---> S2

S2,
   1": %RFFinished% ---> S1
```

Example B:

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.

```
DEFINEMACRO SS1 = {
                   S.S.1,
                   S1,
                     #START: ON 1 ---> S2

                   S2,
                     10": ---> S3
                  }


%SS1%
```

Example C:

Macro definitions may themselves contain "nested" references to other macros. The following code expands properly.

```
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:

```
^Pellet     = 3
^Tone       = 6
^HouseLight = 7


S.S.1,
S1,
  10#R1: OFF HouseLight; ON ^Pellet, ^Tone ---> S2

S2,
  1": ON HouseLight; OFF ^Pellet, ^Tone ---> S1
```

If desired, the now expanded version of a file containing macros may be saved as a separate program.

It is often helpful to save code used in multiple programs as a macro in a library file. For example, a library file named Constants.lib might contain:

```
DEFINEMACRO PigeonConstants = {
                               ^Hopper     = 3
                               ^KeyLight   = 4
                               ^HouseLight = 7
                              }

DEFINEMACRO RatConstants = {
                            ^Pellets    = 3
                            ^Tone       = 6
                            ^HouseLight = 7
                           }
```

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 MSN program 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 = The variables and/or arrays A through Z to be printed

Comments: 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 ---> STOPSAVE  \ Print variables specified by PRINTVARS
                          \   from within MSN program 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 123.5

PRINTFORMAT = 7.2  \ 1234.67

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 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 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). 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: 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:

```
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


PRINTVARS    = A
PRINTFORMAT  = 9.3
PRINTOPTIONS = FULLHEADERS


S.S.1,
S1,
  1": PRINT ---> STOPDISCARD
```

The code above will produce a printout similar to the following:

```
Start Date: 03/01/16
End Date: 03/01/16
Subject: 0
Experiment: 0
Group: 0
Box: 1
Start Time: 14:11:32
End Time: 14:11:33
MSN: Print Sample
A:
     0:     1234.670     1234.670     1234.670     1234.670     1234.670
     5:     1234.670     1234.670     1234.670     1234.670     1234.670
    10:     1234.670     1234.672     1234.677
```

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

Comments: The default is 5 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


PRINTVARS    = C
PRINTCOLUMNS = 3


S.S.1,
S1,
  #K1: PRINT ---> SX
```

The code above will produce a printout similar to the following:

```
Start Date: 03/01/16
End Date: 03/01/16
Subject: 0
Experiment: 0
Group: 0
Box: 1
Start Time: 14:15:25
End Time: 14:16:03
MSN: Print Sample 2
C:
     0:       1.000        2.000        3.000
     3:       4.000        5.000        6.000
     6:       7.000        8.000        9.000
     9:      10.000
```

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 = The variables and/or arrays A through Z to be saved to disk

Comments: DISKVARS must be placed before the first State Set.

When the data format is "Stripped, with only the C array written to disk," DISKVARS has no effect unless C is omitted, in which case no variables or arrays would be written to the data file.

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

Comments: If DISKOPTIONS is not explicitly specified, the default is to write a full header (FULLHEADERS) to the data file.

Syntax: `DISKCOLUMNS = P1`

Where: P1 = The number of columns

Comments: This command only affects the annotated data format.

Example and Discussion:

Saving a data file with the WRITE command takes place in the background, which is to say that WRITE 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 the hard 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 Arrays A and B with elements 0...99
DIM A = 99
DIM B = 99


DISKVARS = B  \ Declare that only Array B will be written to disk


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 T so that the IRT 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) Copy the data to Array 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 Array A.  Also set I to 0 so that
        \         recording resumes at the beginning of A.
        \      C) Issue the WRITE command to request writing the
        \         elements of Array B to disk as time permits.
        \ 5) Transition is to S2 so that the 0.01" IRT timer is reset
        \    whenever a response occurs.
  #R1: SET A(I) = T, T = 0; ADD I;
       IF I = 100 [@Save, @NotYet]
          @Save: COPYARRAY A,B,100;
                 ZEROARRAY A;
                 SET I = 0;
                 WRITE ---> S2
          @NotYet: ---> S2
  0.01": SET T = T + 0.01 ---> SX  \ This line is the IRT timer
```

Y2KCOMPLIANT

The Y2KCOMPLIANT command gets placed before the first State Set and takes no arguments. The effect of this directive is to cause all years appearing in data files and printouts to be written with 4 digit numbers.

For example with the Y2KCOMPLIANT directive 2022 appears as "2022." In the absence of this directive 2022 appears as "22."

Syntax: `Y2KCOMPLIANT`

Example:

```
DISKOPTIONS  = FULLHEADERS
PRINTOPTIONS = FULLHEADERS


Y2KCOMPLIANT


S.S.1,
S1,
  1": PRINT ---> STOPSAVE
```

APPENDIX B| MACRO COMMANDS

All Macro commands are listed in detail, with syntax, comments, examples and discussion. Each command is indexed for convenience.

%PARENTBOX

%PARENTBOX is a special variable that may be used in macros to enable a single macro to work in multiple Boxes.

Consider the following macro command:

```
SET A VALUE 10 MAINBOX 1 BOXES 1
```

This macro sets the value of A to 10 in Box 1. If one wanted a macro that would set the value A to 10 in Box 2, it would be necessary to create another macro:

```
SET A VALUE 10 MAINBOX 2 BOXES 2
```

However, a single macro could be written that would change the value of A in any Box:

```
SET A VALUE 10 MAINBOX %PARENTBOX BOXES
```

The value of %PARENTBOX is set to the currently selected Box in the Box Status panel when a macro is run by either clicking the Play Macro button on the main toolbar, selecting "Macros | Play Macro..." from the main menu or pressing F5.

When a macro is run by clicking the Play Macro button on the toolbar of a Box panel, the value of %PARENTBOX is set to the Box panel's Box number.

LOAD

LOAD is the macro equivalent to the "Sessions | Load Box..." 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 name or number

P3 = The Experiment name or number

P4 = The Group name or 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 = The Box number

P2 = The 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
```

Set Box 3's filename to 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 = The Box number

P2 = The desired comment

Comments may be of any length, but the comment may be cutoff on printouts unless a sufficiently small font is used.

Example:

```
   COMMENT BOX 5 Morphine 10.0 mg/kg, 10 min pretreat.
```

Set Box 5's comment to "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 = The 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.

SAVE_MANUAL

This command has been deprecated. If this command is found in an existing macro it will be ignored. There is no equivalent replacement for this command.

SAVE_FLUSH

This command has been deprecated. If this command is found in an existing macro it will be ignored. There is no equivalent replacement for this command.

STOPABORT and STOPABORTFLUSH

Users upgrading from MED-PC IV might remember the STOPABORT and STOPABORTFLUSH commands. These commands have been deprecated. STOPABORT and STOPABORTFLUSH have both been replaced with the command STOPSAVE. STOPABORT has no direct equivalent and is now interpreted as STOPSAVE. Macros that contain the old commands will still continue to run correctly, however, it is strongly recommended that the new STOPSAVE command is used instead.

STOPSAVE

STOPSAVE is the macro equivalent to the MSN command "STOPSAVE" and the menu command "Sessions | Stop Box..." in conjunction with the "Stop, save data (StopSave)" option. This command ends an experimental session and automatically writes the data to disk.

Syntax: `STOPSAVE BOXES P1 P2 ... P16`

Where: P1...P16 = One or more Box numbers separated by spaces

Example:

```
STOPSAVE BOXES 2 5
```

Close the sessions in Boxes 2 and 5, then write the data to disk.

STOPKILL

Users upgrading from MED-PC IV might remember the STOPKILL command. This command has been deprecated. STOPKILL has been replaced with the command STOPDISCARD. Macros that contain the old command will still continue to run correctly, however, it is strongly recommended that the new STOPDISCARD command is used instead.

STOPDISCARD

STOPDISCARD is the macro equivalent to the MSN command "STOPDISCARD" and the menu command "Sessions | Stop Box..." in conjunction with the "Stop, discard data (StopDiscard)" option. This command ends an experimental session and abandons the data; the data cannot be subsequently saved to disk.

Syntax: `STOPDISCARD BOXES P1 P2 ... P16`

Where: P1...P16 = One or more Box numbers separated by spaces

Example:

```
   STOPDISCARD BOXES 2
```

Close the session in Box 2.

PRINT

Data may be printed under macro control using the macro command PRINT. Data will be printed only for boxes that are currently running.

If forms feeds is turned on (**Data | Printout Settings... | After Each Box**), then a page will be ejected from the printer after every PRINT command. If they are turned off, then the data will be sent to the printer and will remain queued in the printer until a final form feed is sent to the printer (**Data | Form Feed**).

Syntax: `PRINT BOXES P1 P2 ... P16`

Where: P1...P16 = One or more Box numbers separated by spaces

Example:

```
   PRINT BOXES 3 4
```

Print data for boxes 3 and 4.

BOX_PRINTER_SETTINGS

This command has been deprecated. If this command is found in an existing macro it will be ignored.

DEFAULT_PRINTER_SETTINGS

The appearance of printouts may be controlled with the macro command DEFAULT_PRINTER_SETTINGS. This command is equivalent to the "Data | Printout Settings..." menu command.

The default settings are those that MED-PC uses when a Box is loaded. The command will also change the settings for any running boxes except for those running programs that contain MSN commands for controlling the printout layout.

Syntax: `DEFAULT_PRINTER_SETTINGS PRINTWIDTH P1 PRINTDECIMALS P2 PRINTCOLUMNS P3 PRINTPOINTS P4 P5 P6`

Where: P1 = The print width controls 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.

P2 = The print decimals controls the number of digits displayed to the right of the decimal point. The default value of 3 specifies that 3 digits will be printed to the right of decimal points.

P3 = The print columns specifies the number of columns of data that will be printed on the printout. The default value is 5.

P4 = The print points specifies the point size of the characters. The default value is 12.

P5 = PORTRAIT or LANDSCAPE. Determines whether the paper is oriented in PORTRAIT (upright) or LANDSCAPE (sideways) mode. The default is PORTRAIT.

P6 = 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.

Example:

```
DEFAULT_PRINTER_SETTINGS PRINTWIDTH 10 PRINTDECIMALS 2 PRINTCOLUMNS 10
PRINTPOINTS 10 LANDSCAPE FORMFEEDS
```

In this example default printouts are set to 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 (1234567.89). The paper will be in sideways (LANDSCAPE) orientation and a page will be ejected after each Box is printed (FORMFEEDS).

BOX_PRINTER_NAME

This command has been deprecated. If this command is found in an existing macro it will be ignored and an error recorded in the log.

DEFAULT_PRINTER_NAME

The printer used to produce printouts may be selected with the macro command DEFAULT_PRINTER_NAME. This command is equivalent to "Data | Printer Selection..." menu command.

The default settings are those that MED-PC uses when a Box is loaded. The command will also change the settings for any running boxes.

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.

To get the names of available printers start MED-PC and go to "**Data | Printer Selection...**" and a list of installed printers will appear in the drop down list.

SET

SET is a macro command that may be used to change the value of variables, array elements, named variables (VAR_ALIAS), and equated variables (EQUATE). This command is equivalent to the "Session |Change Variables..." menu command. SET alters the values of variables in currently running boxes only.

Syntax: `SET P1 VALUE P2 MAINBOX P3 BOXES P4 P5 ... P19`

Where:	P1	= The variable, array element, named variable, or equated variable to alter
	P2	= The value to which P1 should be set
	P3	= The Box to be affected
	P4...P19	= One or more Box numbers separated by spaces whose variables should be affected.

Comments: Note that the BOXES argument must be specified, but the parameters P4 - P19 can be left blank or it can be set to the same value as MAINBOXES.

Example A:

```
SET X VALUE 10.1 MAINBOX 3 BOXES 3
```

Variable X is set to 10.1 in Box 3.

Example B:

```
SET X VALUE 10.1 MAINBOX 3 BOXES
```

Variable X is set to 10.1 in Box 3 with BOXES parameter left blank.

Example C:

```
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 E:

```
SET FRSize VALUE 10 MAINBOX 1 BOXES 1
```

Set named variable FRSize to 10 in Box 1.

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

\ A = FR Size


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

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

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

This program arranges a simple FR. 100 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 FR
SET A VALUE 20 MAINBOX 1 BOXES 1
```

When this macro is run, it will load the program FR into Box 1 and then variable "A" will be set to the value of 20 almost immediately. The FR program will also start running as soon as it is loaded, but S.S.1, S1 will not execute until 0.1" (100 milliseconds) after the program is loaded which will cause the variable "A" to be set back to the value of 10.

The following macro avoids this problem by introducing a delay of 1000 milliseconds (1s) after the Box is loaded before trying to set the value of A.

```
LOAD BOX 1 SUBJ 1 EXPT FR Demo GROUP 2 PROGRAM FR
DELAY 1000
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 "Sessions | Signals (Start, #K, #R)..."

START BOXES

Sends a START signal to one or more Boxes.

Syntax: `START BOXES P1 P2 ... P16`

Where: P1...P16 = One or more Box numbers separated by spaces

Example:

```
START BOXES 5 6 9
```

Send a START signal to Boxes 5, 6 and 9

K

Sends a K pulse to one or more Boxes.

Syntax: `K P1 BOXES P2 P3 ... P17`

Where:	P1	= A K signal
	P2...P17	= One or more Box numbers separated by spaces

Example:

```
K 5 BOXES 1
```

Send K5 to Box 1.

R

Sends a simulated Response to one or more Boxes.

Syntax: `R P1 BOXES P2 P3 ... P17`

Where:	P1	= A R signal
	P2...P17	= One or more Box numbers separated by spaces

Example:

```
R 5 BOXES 1
```

Send R5 to Box 1.

NOTE: The "SYNCH" parameter formerly used as an optional parameter for these commands is now ignored because MED-PC now synchronizes all signals (i.e., they are all issued simultaneously).

Controlling Outputs with 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 "Sessions | Outputs..."

ON

Turns the specified Output on.

Syntax: `ON P1 BOXES P2 P3 ... P17`

Where:	P1	= An Output number
	P2...P17	= One or more Box numbers separated by spaces

Example:

```
ON 3 BOXES 5 6 9
```

Turn on Output 3 in Boxes 5, 6, and 9.

OFF

Turns the specified Output off.

Syntax: `OFF P1 BOXES P2 P3 ... P17`

Where:	P1	= An Output number
	P2...P17	= One or more Box numbers separated by spaces

Example:

```
OFF 5 BOXES 1
```

Turn off Output 5 in Box 1.

LOCKON

Locks the specified Output on.

Syntax: `LOCKON P1 BOXES P2 P3 ... P17`

Where:	P1	= An Output number
	P2...P17	= One or more Box numbers separated by spaces

Example:

```
LOCKON 3 BOXES 5 6 9
```

Lock Output 3 on in Boxes 5, 6, and 9

LOCKOFF

Locks the specified Output off.

Syntax: `LOCKOFF P1 BOXES P2 P3 ... P17`

Where:	P1	= An Output number
	P2...P17	= One or more Box numbers separated by spaces

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 P4 ... P18`

Where:	P1	= An Output number
	P2	= The number of seconds during which Output should be on
	P3...P18	= One or more Box numbers separated by spaces

Example:

```
   TIMED_OUTPUT 2 DURATION 2.50 BOXES 1
```

Turn on Output 2 in Box 1 and then turn it off 2.5 seconds later.

INPUTBOX

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). 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 = A quoted string containing the text of the input prompt that appears in the middle of the dialog box

P3 = A quoted string containing the 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.

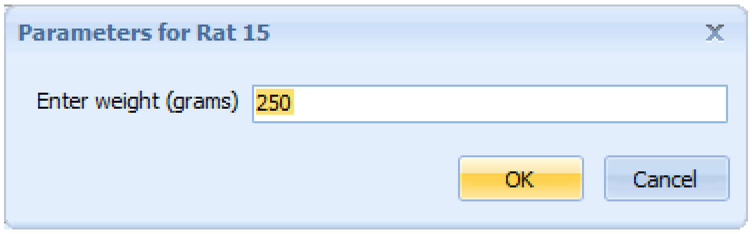
If the response will be used to set a variable value, then the NUMERICINPUTBOX should be used instead (see below).

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 W 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 B.1 - Dialog Box

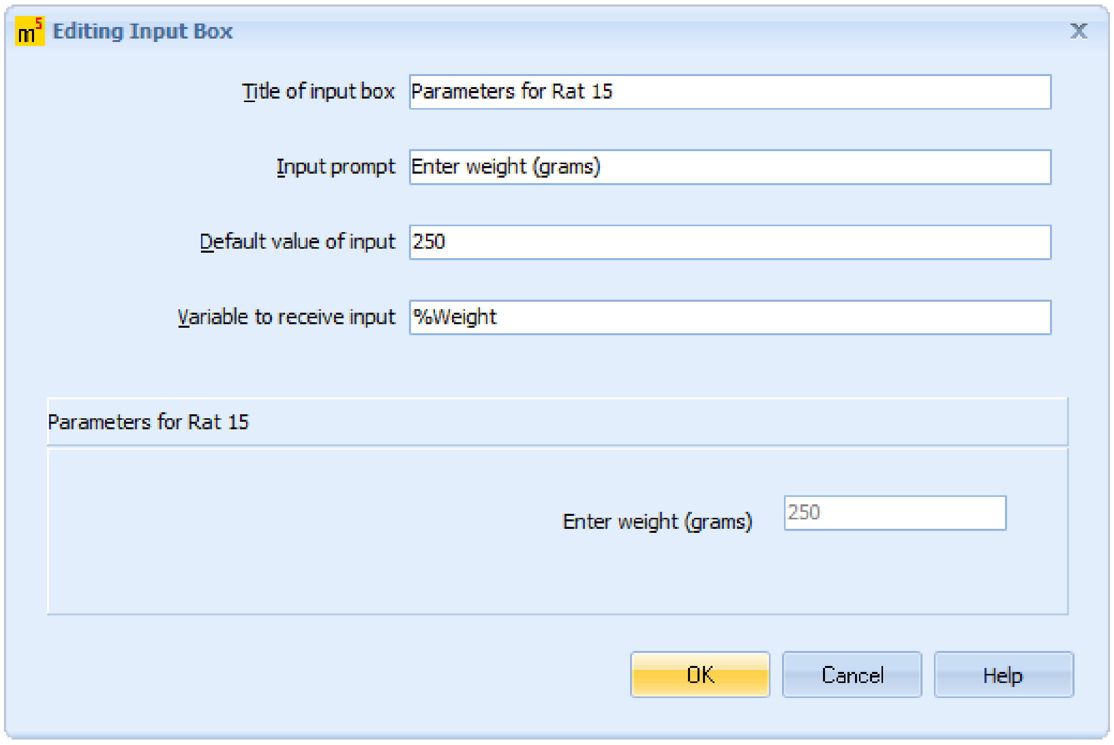


A key thing to notice about the INPUTBOX command above is that the last parameter is %Weight. %Weight is a variable that could have been named anything and is not displayed on the screen. However, the variable is used to store the value that was entered into the input field by the operator. It is this variable (%Weight) that is then used in the SET command as the value to be assigned to the MSN variable "W."

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 B.2:

Figure B.2 - Input Box


Editing Input Box

Title of input box: Parameters for Rat 15

Input prompt: Enter weight (grams)

Default value of input: 250

Variable to receive input: %Weight

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.

Syntax: `NUMERICINPUTBOX 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 = A quoted string containing the text of the input prompt that appears in the middle of the dialog box

P3 = A quoted string containing the 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: 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.

Syntax: `TEXTINPUTBOX 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 = A quoted string containing the text of the input prompt that appears in the middle of the dialog box

P3 = A quoted string containing the 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: 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.

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 quoted string containing the text to be displayed

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.

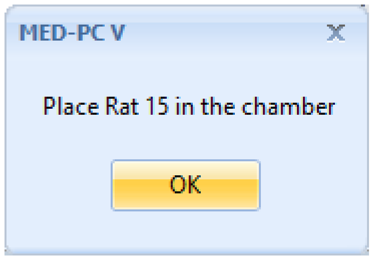
Example:

The following macro loads Rat 15 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 1 SUBJ Rat 15 EXPT FR GROUP One PROGRAM Concurrent FR FI
SHOWMESSAGE "Place Rat 15 in the chamber"
START BOXES 2
```

Figure B.3 is the show message produced by the macro:

Figure B.3 - Show Message Produced by the Macro



Place Rat 15 in the chamber

PLAYMACRO

PLAYMACRO is an extremely useful macro command that provides for nested playback of macros.

Syntax: `PLAYMACRO P1`

Where: P1 = The full path and file name of the macro to be played

Example:

```
PLAYMACRO C:\MED-PC\Macro\FR.mac
```

Play a macro named C:\MED-PC\Macro\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:\MED-PC\Macro\RAT1.MAC
PLAYMACRO C:\MED-PC\Macro\RAT2.MAC
PLAYMACRO C:\MED-PC\Macro\RAT3.MAC
PLAYMACRO C:\MED-PC\Macro\RAT4.MAC
```

When recording a macro MED-PC will automatically record the full path and file name of any macro that is played during that recoding session.

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.

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 "Sessions | Exit When All Sessions 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 (STOPDISCARD and STOPSAVE) 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.

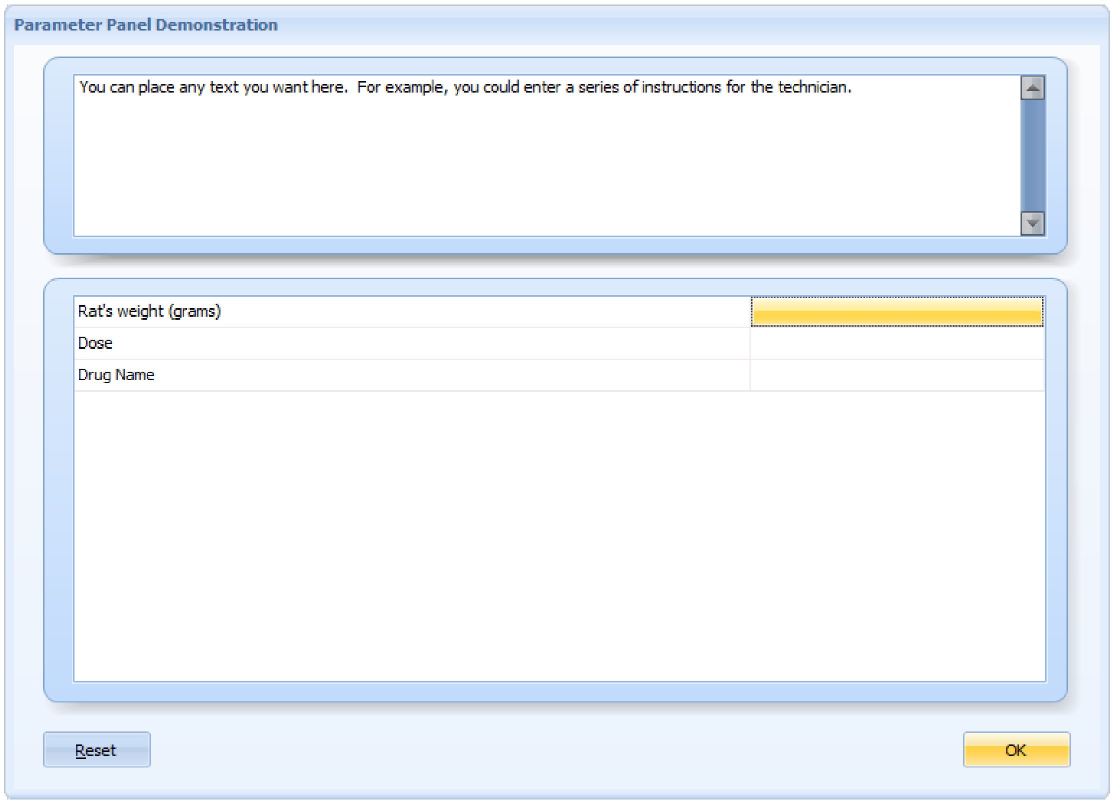
Syntax: `RESET_ERRORS`

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.

PARAMETER_PANEL

Parameter panels provide a way for macros to launch highly customized dialog boxes that can be used to set the value of macro variables. Below is a screenshot of a parameter panel that might be used in an experiment involving drug administration.

Figure B.4 - Parameter Panel



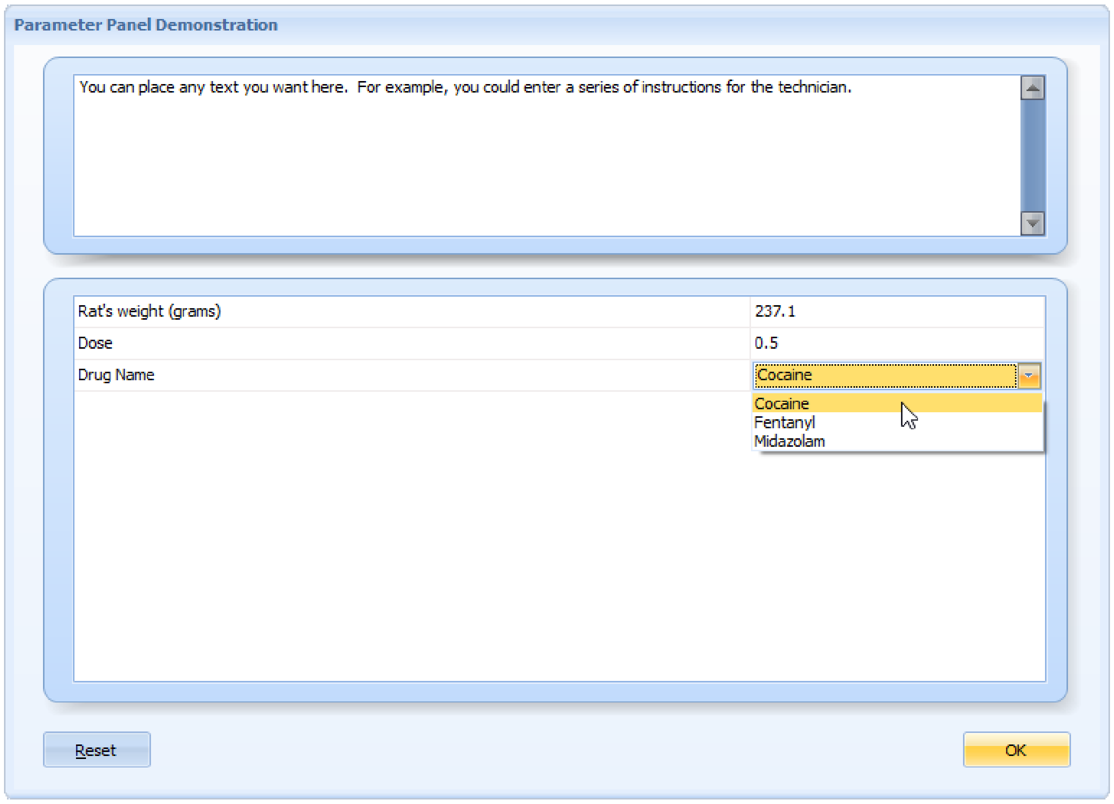
Parameter Panel Demonstration

You can place any text you want here. For example, you could enter a series of instructions for the technician.

Rat's weight (grams)	
Dose	
Drug Name	

The dialog can have a custom title, instructions or other text, and a grid where data can be entered in the second column. There is the ability to restrict data entry to specific values or types of data. For example, in the example below, the Drug Name field is restricted to Cocaine, Fentanyl, and Midazolam and the value is selected from a dropdown list.

Figure B.5 - Parameter Panel with Dropdown List



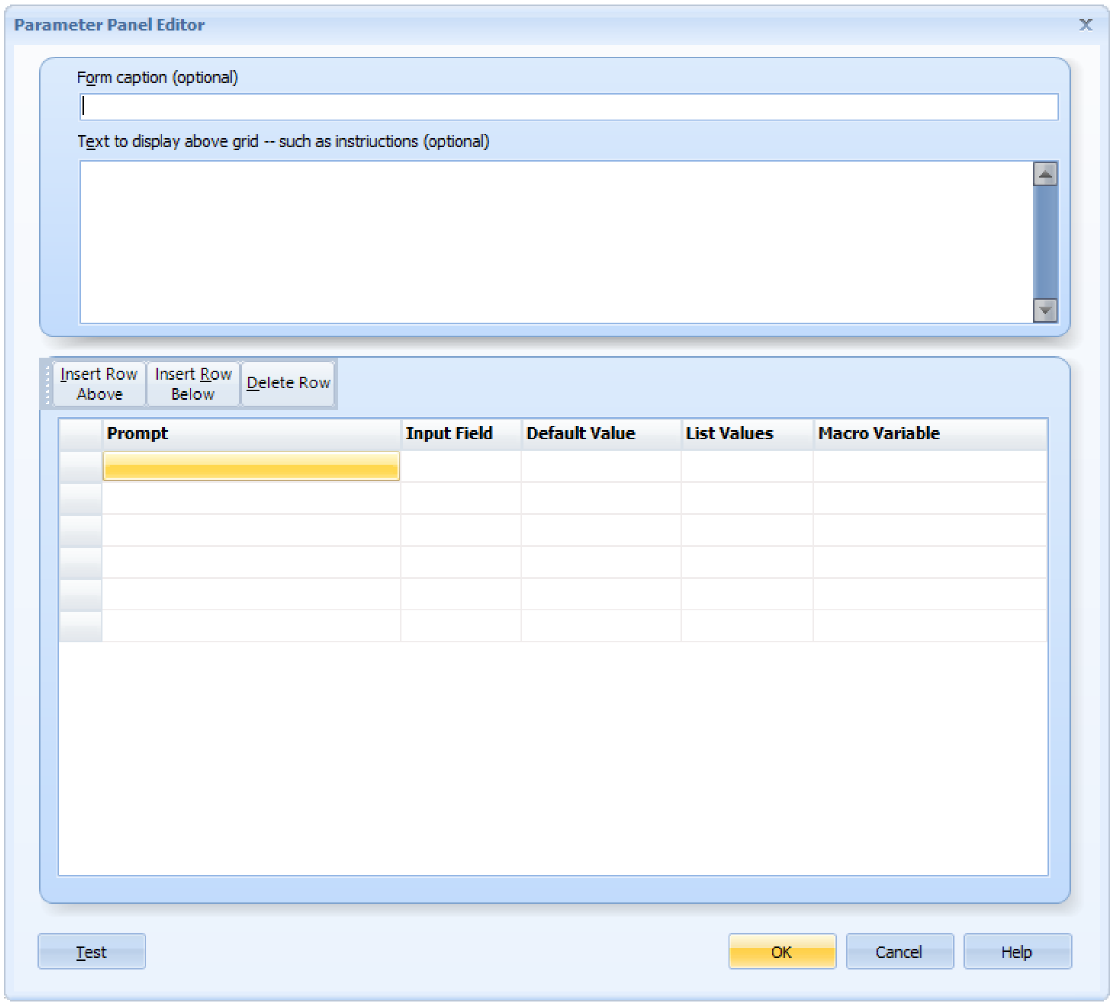
You can place any text you want here. For example, you could enter a series of instructions for the technician.

Rat's weight (grams)	237.1
Dose	0.5
Drug Name	Cocaine

Cocaine
Fentanyl
Midazolam

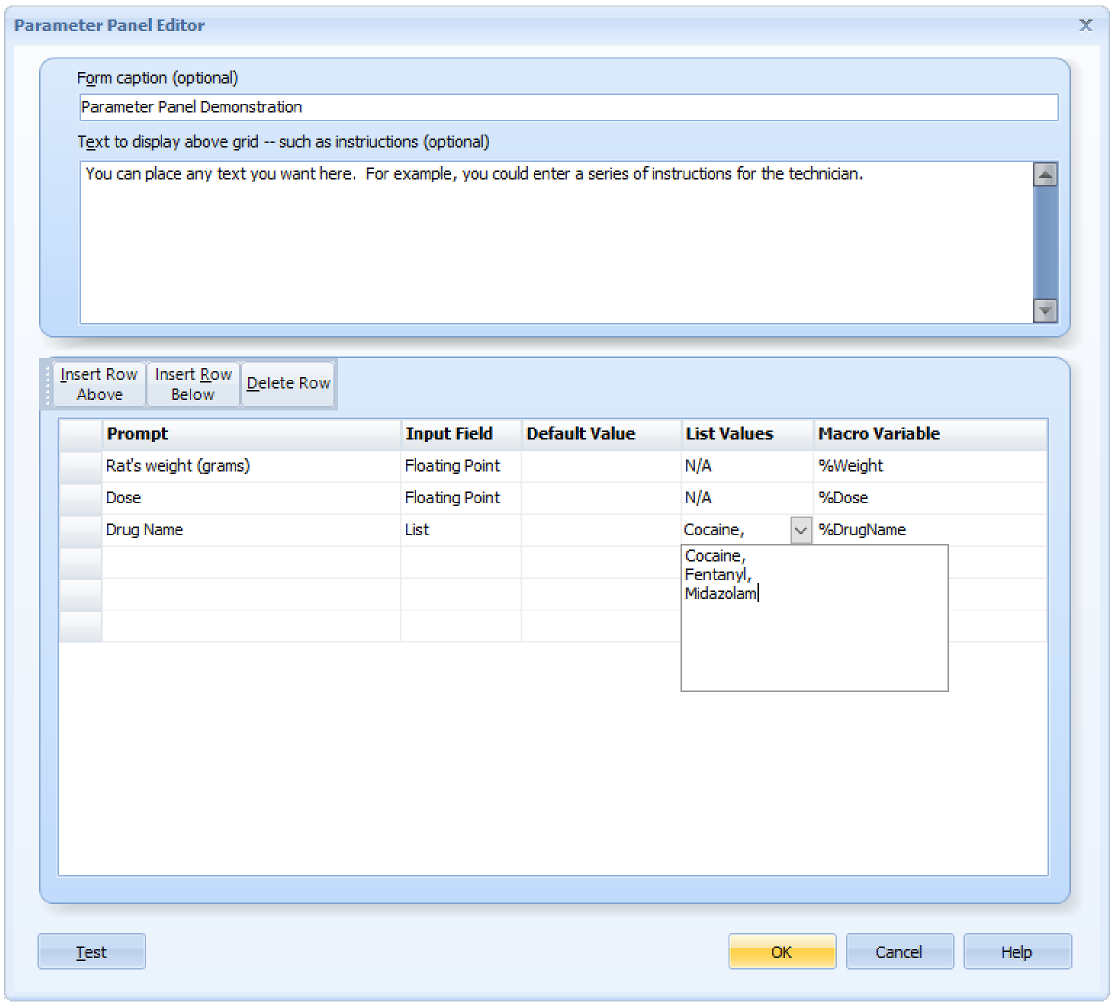
Parameter panels cannot be directly recorded - they must be created from within the macro editor "Macros | Editor" by right-clicking within the editor and selecting "Insert a Command | Parameter Panel (PARAMETER_PANEL)..." The Parameter Panel Editor is displayed below.

Figure B.6 - Parameter Panel Editor



The parameter panel displayed at the top of this section was created by filling in the editor as follows:

Figure B.7 - Editing Parameter Panel Fields



Form caption (optional)

Parameter Panel Demonstration

Text to display above grid -- such as instriuctions (optional)

You can place any text you want here. For example, you could enter a series of instructions for the technician.

	Prompt	Input Field	Default Value	List Values	Macro Variable
	Rat's weight (grams)	Floating Point		N/A	%Weight
	Dose	Floating Point		N/A	%Dose
	Drug Name	List		Cocaine,	%DrugName

Cocaine,
Fentanyl,
Midazolam

Prompt: Text entered in this column is displayed as a prompt or label in the parameter panel.

Input Field: This column determines what type of data may be entered or selected by the user of the parameter panel.

Normal indicates that any type of value is acceptable (text, symbols, digits, etc.).

Integer indicates that only whole numbers may be entered.

Floating Point accepts either a whole number or a decimal number.

List accepts only values in the dropdown list. The values for the dropdown list are entered in the "List Value" column separated by commas.

Editable List is similar to List but the user is allowed to either select a value from the dropdown list or type in a value not on the list.

None indicates that no value can be entered; this can be used to insert a blank line or lines containing only the text displayed in the prompt column.

Default Value:	Any value entered here will be displayed on the parameter value as the initial (default) value.
List Values:	This contains the values for the dropdown list when the Input Field Type is either "List" or "Editable List."
Macro Variable:	Contains the name of the variable that will hold the value entered by the user when the parameter panel is used in a macro. The macro variable can be used as parameters in other macro commands. For example, placing the SET command "SET A VALUE %Weight MAINBOX 1 BOXES 1" after the Parameter Panel code would set the value of variable A to whatever value was enter by the user.

The Test button allows one to interactively test the macro.

Parameter panels should only be edited using the built-in macro editor; this can be done by clicking on the beginning of the code for the parameter panel <Parameter_Panel> and selecting **Edit Command at Cursor...** from the context menu.

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 tilde (~) 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
                           \ NOTE: The Pascal assignment operator is :=
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.

Writing INLINE Pascal Procedures

INLINE Pascal procedures consist 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 MED-PC Global parameter). 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 (MED-PC
                                   \   Global paramter) and Variables
                                   \   A & B, Return with Value C

S2,
  1": SHOW 1,A=,A, 2,B=,B, 3,C=,C ---> S1
```

2. The procedure 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: Real;
                Var Result: Real);

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 at the top of the USER.PAS file, as follows:

```
INTERFACE

Procedure ADDAB(MPCGlobal: MPCGlobalPtr;
                Num1, Num2: Real;
                Var Result: Real);

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:

```
^Pellet = 3


LIST A =  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
         11, 12, 13, 14, 15, 16, 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 ^Pellet ---> S5

S5,
  0.05": OFF ^Pellet; 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 command. As soon as the call to ConstantVI in USER.PAS is executed the dummy values are replaced with appropriate values for the desired VI.

Syntax: `INPUT: ~ ConstantVI(MG, P1, P2, P3);~ ---> NEXT`

Where:	MG	= The MED-PC Global Parameter. This is required as the first parameter.
	P1	= Name of the array
	P2	= Number of elements in the array. In the example above, this number was calculated automatically.
	P3	= Mean Variable Interval 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 the variable X within the ConstantVI procedure 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: Real);

Var
  I:     Integer;
  Term1: Real;
  Term2: Real;
  Tn:    Array [1..1000500] of Real 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}
```

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,
      //   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 Variable 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 ChangeSubjectNum(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;  {ChangeSubjectNum}
```

Warning: Array Accesses must be Within the Declared Range of the Array!

Windows does not tolerate User-written code that reads beyond the ends of an array and it could cause a GPF (General Protection Fault). 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];  \ Trying to access X[11] is an error
```

Background Procedures

There are a variety of routines that should not be done in Inline Pascal or the USER.PAS due to the fact that they are very time consuming. Complex calculations or tasks such as writing to the disk or printer may take an indeterminate amount of time. However, the MSN command BKGRND along with the file BACKPROC.PAS allows the User to accomplish these tasks..

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 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 following MSN sample code 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 called and then the program transitions to S2 occurs. Note that the program will stay in S2 until the variable 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.

Sample MSN code illustrating the use of the 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 the values for demonstrations purposes
  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(0); SHOW 1,B(0)=,B(0) ---> SX
  #K2: ADD B(1); SHOW 2,B(1)=,B(1) ---> SX
  #K3: ---> S5  \ Let K3 simulate some basis for ending the session
  1":  ADD B(2); SHOW 3,B(2)=,B(2) ---> SX

S5,
  0.1": SET D = 0; BKGRND 1 ---> S6

S6,     \ Don't go immediately into STOPDISCARD or STOPSAVE
        \ Wait until Background Proc says done via D = 1
  0.1": IF D = 1 [] ---> STOPDISCARD
```

The MSN code also illustrates writing to a data file at the end of the session. Importantly, notice that before BKGRND 1 was called that the variable "D" was set back to 0 and that the transition to STOPDISCARD does not occur until "D" has been set to 1. This ensures that the procedure is executed before the STOPDISCARD command.

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 extremely important, because it informs MED-PC that the procedure has finished executing for the requested box.

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: C:\MED-PC\DATA\subject 4500.txt).

The following is the code for BKGRND procedure 1. This procedure will generate an error if it cannot write to the data file.

```
{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\Subject ' +
                              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 SETS 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}
```

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.

```
{$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\Subject ' +
                                   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 B 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 SETS 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}
{$I+}
```

Compiling Background Procedures

The BACKPROC.PAS needs to be recompiled after any changes are made. This is done automatically by translating and compiling any MPC program. Doing this will cause MED-PC to recognize that the BACKPROC.PAS has been changed and do an automatic compilation of the file.

Guidelines for writing and calling BKGRND procedures:

- BKGRND procedures are called from the output section of MSN statements.

  Syntax: `INPUT: BKGRND P1 ---> NEXT`

  Where: P1 = A number from 1 to 10 inclusive

- 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 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 its second subscript is properly defined.

- 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. Copy any custom code from the old USER.PAS file to the new USER.PAS file supplied with the present version of MED-PC.

3. Copy any custom code from the old BACKPROC.PAS file to the new BACKPROC.PAS file supplied with the present version of MED-PC.

   NOTE: In some select cases it may be possible to just copy the BACKPROC.PAS from the old MED-PC IV folder into the new MED-PC folder.

APPENDIX D|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 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 for a .MPC procedure named TEST1.MPC.

```
Type
  MPCProgramType = packed 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 : Real;
    G : Packed Array [0..10] of Real;
  End;

  MPCProgramArrType = Packed Array [1..16] of ^MPCProgramType;

Var
  MPCProgramRec:MPCProgramArrType;
```

The procedure has one array, "G," declared by DIM G = 10. Each .MPC procedure has an array of pointers to data structures. MPCProgramRec 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 Description is of type DescriptionRec and Header is of type and HeadRec. DescriptionRec and HeaderRec are both defined in the file MEDTYPES.PAS.

The following record definitions are found within MEDTYPES.PAS and are used to define data structures referenced in the data definitions of individual MSN procedures:

```
DescriptionRec = packed record
  RecordSize: longint;
  Vars: packed Array ['A' .. 'Z'] of VarRec;
  PPrintoutDescription, DPrintoutDescription: TUserPrinterSettings;
  ThePrinter: TUserPrinterSettings;
  PA_H, PI_P, PQ_X, PY_Z: byte;
  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;
  VarAliasListString: widestring;
  VarEquateListString: widestring;
  FlushPermission: Boolean;
  LastJournalEntry: Integer;
  RngCheckErrors, ZErrorCount: Integer;
End;
```

DescriptionRec contains miscellaneous information about the .MPC procedure and the Box running the procedure.

DescriptionRec has the following fields:

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. 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 MPCProgramArrType.

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 MPCProgramRec[3]^.Description.VARS['C'] (assuming Box 3 is loaded with TEST1.MPC). The elements of VARS are of type VarRec. The fields of VarRec are defined is as follows:

```
VarRec = packed record
  VarKind, HasList: byte;
  Size: Integer;
  StartOffset: Integer
  Sealed: Boolean;
End;
```

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 (8 bytes after the start). Note that this value is expressed in terms of data elements, not bytes.

In MPCProgramRec, Variable A has a StartOffset of 0 and Variable B has a StartOffset of 1, etc. So the variables go from 0...24 and because Array G begins immediately after the simple variables, its StartOffset is 25 and goes to 34 (25...34 is ten elements).

Of course, if another array is added that started after G, its StartOffset would not be 35, but rather 34. With two arrays there would now be only 24 variables that go from 0...23, the G array would now go from 24...33, and the new array would start at 34. 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.

Sealed: Indicates whether the data element is a sealed array.

PPrintoutDescription, **DPrintoutDescription**, and **ThePrinter**: 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 is as follows:

```
PUserPrinterSettings = ^TUserPrinterSettings;

TUserPrinterSettings = packed Record
  FWidth: Integer;
  Decimals: Integer;
  Columns: Integer;
  Points: Integer;
  Orientation: Integer;
  CondensedHeaders: Integer;
  FormFeeds: Integer;
  FontName, PrinterName: widestring;
  DiskFormatSpecified: Boolean;
End;
```

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 variables 'A,' 'B,' and 'H.'

PI_P, **PQ_X**, and **PY_Z**: These variables are analogous to PA_H, but for variables I – P, Q – X, and Y – Z respectively.

DA_H, **DI_P**, **DQ_X**, and **DY_Z**: These variables are analogous to PA_H through PY_Z except that they control the internal structure of disk files.

TimeDurs, **Times**, **Ins**: These variables are presently not used and may be dropped in subsequent releases.

RecordH and **RecordT**: These variables are presently not used and may be dropped in subsequent releases. The structure of these records is as follows:

```
RecordRecPtr = ^RecordRec;

RecordRec =  Record
  St: widestring;
  Next: Pointer;
end;
```

AddRecord: This variable is 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.

VarAliasListString: This is a WideString that contains the names of all variable aliases and the variables or array elements to which they refer.

VarEquateListString: This is a WideString that contains the names of all equated variables 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.

LastJournalEntry: Used internally.

RngCheckErrors and **ZErrorCount**: Used internally.

The following fields comprise the HeadRec:

```
HeadRec = record
  Source: widestring;
  StartMonth, StartDate, StartYear,
  EndMonth,   EndDate,   EndYear,
  SubjectNumber, ExpNumber, GroupNumber, BoxNumber,
  StartHours, StartMinutes, StartSeconds,
  EndHours,   EndMinutes,   EndSeconds,
  Y2KStartYear: real;
  SubjectSt, ExpSt, GroupSt: widestring;
  TheLibHandle: THandle;
  TheBoxesFreeProc: FreeMPCProcType;
  FileName: widestring;
  Comments: widestring;
  Started: Boolean;
End;
```

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**, **EndSeconds**, and **Y2KStartYear**: These variables comprise the special identifiers described in detail in Appendix A.

SubjectSt, **ExpSt**, **GroupSt**: String representations of the Subject, Experiment and Group identifiers.

TheLibHandle, **TheBoxesFreeProc**: Used internally.

Filename: The name under while the file will be saved.

Comments: Session comments.

Started: Indicates whethe the box received one or more start signals within the runtime system.

DataRec - Universal Access to a Box's Data

The Definition of DataRec

```
DataRec = packed record
  Description: DescriptionRec;
  Header: HeadRec;
  BigArr: packed Array [1 .. 1000500] of real;
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 E | 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 or call us at 1-802-527-2343.

INDEX