

Proc File Basics

Proc File Basics

Written by the Operating Systems Department

© Copyrighted by D.D.M.S., Inc.
Printed in the United States of America.

Contents

Section 1: An Introduction to Procs	5
How are Procs Created	5
How to Run Procs	6
Section 2: Procedure Commands - A Brief Description:	7
Section 3: Procedure Commands - Quick Command Descriptions	10
Section 4: Replaceable Variables	13
Section 5: Procedure Commands - Detailed Descriptions	14
* (Remark Character)	14
ASSIGN	14
BEEP	15
BEGIN	15
CHAIN	16
CLEAR	16
DATECALC	16
DATEDOW	17
DSKCHKON and DSKCHKOFF	17
ENDTEXT	18
ERASE	18
EXIT	19
FINDFILE	19
FLUSH	20
GET\$, GET% and GET#	20
GETCHAR	21
GETMAIL	22
GETUSAGE	23
GOSUB	23
GOTO	24
INC and DEC	24
IF, IF# - THEN and ENDIF	25
KEYON and KEYOFF	27
KEYPRESSED	28
LOCK and UNLOCK	29
MENU and MENUEND	29
PAUSE	30
POSITIONXY, POSITIONXYC, POSITIONXYS	31
PRETTY	32
PRINT	32
PRINTWP	32
RANDOM	33
REBOOT	33
RECLen	33
RENAME	34
RESET	35
RESTORE	35
RETURN	36
RUN	36

Proc File Basics

SAVE	37
SENDMSG	38
SETDEVICE	39
SETHI	39
SETLO	40
SETMAIL	40
SHOW	40
SHOWAT	41
SHOWLN	41
STRMOVE	41
TEXT	43
TRACEON	43
TRACEOFF	44
WAIT	44
WAITUNTIL	44
Section 6: Error Messages	46
Section 7: Error Message Descriptions	47
Section 8: A Proc Demonstration	49

Section 1: An Introduction to Procs

Procs, sometimes called procedure or batch files, are a method of combining a series of operations or programs into a single operation. The new proc system is improved over the old way of setting up procs which required that programs be programmed specifically to handle procs. The new system allows virtually any program to be "proc'd". This should greatly increase the flexibility of the DDMS software and usage.

HOW ARE PROCS CREATED

Procs are created using the editor or word processor supplied with the DDMS system. Each line of the document is a proc "COMMAND". Proc commands tell the proc executor what to do next whether it is to run a program, get input from the user or display a message on the screen. Since the proc is a normal word processor type of file means that they may be changed readily and quickly. In computer lingo the proc executor program is called an "INTERPRETER", which means that each line of the proc is read and acted upon by the proc executor. The lines of the proc are read in order sequentially unless the executor comes to a line that sends it to somewhere else in the proc. Procedure commands and their parameters may be placed anywhere on the line with at least one space between the procedure command and the parameter.

Example:

Correct Way :

```
col. 123456789 etc.
|-----+
  SHOWLN Hello World
```

Also Correct :

```
      SHOWLN Hello World
^^^^^^- Notice that spaces are in front of SHOWLN
```

Another Correct Variation:

```
      RENAME          MYFILE 4
```

Continuation Character

A "+" character placed in the column 80 of the line allows the next line in the proc to be a continuation of the current line. This is useful for long "RUN" commands that have extensive lists of input characters.

Example:

```
Column 80 -----+
      |
      v

RUN ;NEW:R 4  THIS IS A LONG SEQUENCE OF INPUT CHARACTERS .....+
THIS IS A CONTINUATION OF THE PREVIOUS LINE
```

HOW TO RUN PROCS

Procs are executed through the utilities master screen by entering a "B" - "6" which is the command execute procedure file. The following is displayed:

```
PROCEDURE FILE EXECUTIVE --- VERSION 1.0  
ENTER PROCEDURE FILE  
ENTER FILE NAME:      ENTER UNIT:
```

You then enter the proc name which you created through the editor or word processor and give the appropriate unit number. The procedure file executive will now execute the commands until there are no more commands left to execute.

Section 2: Procedure Commands - A Brief Description:

This is a brief summary of commands currently available with the procedure executive:

COMMAND	DESCRIPTION
*	- denotes a proc remark - no action is performed
ASSIGN	- allow proc program to set a replaceable parameter
BEEP	- beep the display terminal
BEGIN	- execute a program in the background (i.e. BATCH)
CHAIN	- run another program thus exiting the proc executive
CLEAR	- clear the display terminal screen
DATECALC	- allows the proc to add/subtract days from/to a date
DATEDOW	- returns the day of week for any date
DEC	- decrements one of the user variables by 1
DSKCHKON	- turns file error checking ON for RENAME and ERASE
DSKCHKOFF	- turns file error checking OFF for RENAME and ERASE
ENDIF	- used with the IF command
ENDTEXT	- used with TEXT command to display text on screen
ERASE	- allows a file to be deleted
EXIT	- unconditionally stop execution of a proc
FINDFILE	- searches all hard drives for a file and returns the unit
FLUSH	- flushes the internal screen/program cache memory
GET\$	- allows input of text into one of the variables
GET%	- allows input of text with a given length
GET#	- get numbers only into one of the variable parameters
GETCHAR	- get a single character into one of the variable parameters
GETMAIL	- gets a number from one of the 16 internal mailboxes
GETUSAGE	- returns the percentage in use on any disk unit
GOSUB	- execute a routine and return to where it left off
GOTO	- have the executive start execution at different place
IF	- have the executive perform command based on a condition

Proc File Basics

COMMAND	DESCRIPTION
IF#	- perform command based on a numeric condition
INC	- increments one of the user variables by 1
INITIALIZE	- allows a proc to initialize an unit.
KEYON	- allows procs to be aborted if a key is pressed
KEYOFF	- disables key checking while proc is running
KEYPRESSED	- allows a proc to check if a key is pressed on keyboard
LOCK	- allows a proc to lock the system program loader.
MENU	- displays a menu and lets the user select an item
MENUEND	- denotes the end of menu selections
PAUSE	- temporarily stop execution of proc for a period of time
POSITIONXY	- position cursor to a particular spot on terminal
POSITIONXYC	- same as POSITIONXY except clear to the end of the line
POSITIONXYS	- same as POSITIONXY except clear to the end of the screen
PRETTY	- allows a program to print a program source.
PRINT	- makes a hard copy of a document file to a printer
PRINTWP	- uses new version of print program to print a file
RANDOM	- returns a random number for use with diagnostic routines
REBOOT	- allows the proc to reboot the system
RECLEN	- returns the record length given the file and unit
RENAME	- allows a file to be renamed to a new name
RESET	- allows the proc to reset the screen table area
RESTORE	- allows a parameter to be restored from save area
RETURN	- returns to where a GOSUB was called from
RUN	- execute a program
SAVE	- allow a replaceable parameter to be saved to save area
SENDMSG	- allows a proc to send a message to a device
SETDEVICE	- allows a proc to set the device to use for RUN commands
SETHI	- cause subsequent text displayed to be hi-lighted
SETLO	- cause subsequent text displayed to be normal intensity
SETMAIL	- allows a proc to set one of the 16 internal mailboxes

COMMAND	DESCRIPTION
SHOW	- display a message on the terminal
SHOWAT	- displays text at specific location on the terminal
SHOWLN	- display a message on the terminal followed by cr/lf
STRMOVE	- allow extraction or insertion a string into a string
SYSRES	- allows the proc to change the current system resident unit
TEXT	- displays following lines of text on screen until ENDTEXT
TRACEON	- turns proc file debug on
TRACEOFF	- turns proc file debug off
UNLOCK	- unlocks the system after a lock command
WAIT	- wait for user to press a key on the terminal
WAITUNTIL	- stop execution until a certain time has elapsed

Section 3: Procedure Commands - Quick Command Descriptions

This is a quick summary of commands and parameters currently available with the procedure executive. For more detailed information on these commands, consult the COMMAND DESCRIPTIONS section of this documentation.

COMMAND	PARAMETERS REQUIRED
*	
ASSIGN	&<variable#> <contents>
BEEP	
BEGIN	<program> <unit/volser> <device>
CHAIN	<program> <unit/volser>
CLEAR	
DATECALC	<date> <amount to add/subtract> &<variable#>
DATEDOW	<date> &<variable#>
DEC	&<variable#>
DSKCHKON	
DSKCHKOFF	
ENDTEXT	
ERASE	<filename> <unit/volume>
EXIT	
FINDFILE	<filename>
FLUSH	
GET\$	&<variable#>
GET%	<length> &<variable#>
GET#	<length> &<variable#>
GETCHAR	&<variable#>
GETMAIL	<whichone (1-16)>,&<variable#>
GETUSAGE	<unit/volume>
GOSUB	<label>
GOTO	<label>

COMMAND	PARAMETERS REQUIRED
IF	<NOT> <str1> <condition> <str2> <COMMAND>
IF#	<NOT> <str1> <condition> <str2> <COMMAND>
INC	&<variable#>
INITIALIZE	<unit/volume>
KEYON	
KEYOFF	
KEYPRESSED	&<variable#>
LOCK	
MENU	<col> <row> <promptchar>
MENUEND	
PAUSE	<ticks/seconds> SECONDS
POSITIONXY	<col> <row>
POSITIONXYC	<col> <row>
POSITIONXYS	<col> <row>
PRINT	<filename> <unit/volume> <printer> <copies>
PRINTWP	<filename> <unit/volume> <printer> <copies> <a/o/e>
RANDOM	<bounds> &<variable#>
REBOOT	
RECLN	<filename> <unit/volume>
RENAME	<filename> <unit/volume> <newfilename>
RESET	
RESTORE	&<variable#>
RETURN	
RUN	<program> <unit/volume> <input characters>
SAVE	&<variable#> <contents>
SENDMSG	<device> <message>
SETDEVICE	<device>
SETHI	
SETLO	
SETMAIL	<whichone (1 - 16)> <to what (0 - 65535)>

Proc File Basics

COMMAND	PARAMETERS REQUIRED
SHOW	<text>
SHOWAT	<col> <row> <text>
SHOWLN	<text>
STRMOVE	<position1> <string1> <position2> <string2> <length>
SYSRES	<unit/volume>
TEXT	
TRACEON	
TRACEOFF	
UNLOCK	
WAIT	
WAITUNTIL	<hh:mm:ss> <day of week/@date>

Section 4: Replaceable Variables

Replaceable variables are used to insert certain text into the commands being processed. They are "expanded" before the command is executed. These variables are:

&1 - &99 are used to hold input from the GET\$ or ASSIGN command. System variables: These are used to give the proc writer convenient access to commonly used things such as the date, time etc. These CANNOT be changed within the proc executive. Like the replaceable variables these will be "expanded" in the proc before it is executed.

- &D** - inserts the current system date into the proc command
- &T** - inserts the current system time into the proc command
- &M** - inserts the current system month into the proc command
- &W** - inserts the current system day into the proc command
- &Y** - inserts the current system year into the proc command
- &H** - inserts the current system hour into the proc command
- &N** - inserts the current system minutes into the proc command
- &S** - inserts the current system seconds into the proc command
- &C** - inserts the type of OPD system currently operating (i.e. ASST for the ASSISTANT or OPD for the standard OPD-DEALER)
- &V** - inserts the current version of the Operating System (i.e. B115)
- &E** - This variable is used in conjunction with the ERASE or RENAME, RECLLEN command. &E contains a YES if a disk error has occurred, otherwise, it is blank. Some programs when used with the RUN command will also set this variable. (i.e. Z-C-1 and Z-C-3)
- &F** - contains the percentage of disk space used from the last invocation of the GETUSAGE command.
- &R** - contains the results from the last invocation of the RECLLEN command.
- &B** - This variable is spaces. It allows the proc to determine if one of the other variables is blank. (i.e. IF &1 = &B SHOWLN THAT FIELD IS BLANK)
- &K** - This contains the text from the last KEYPRESSED operation - YES if a key is available or NO if no key is waiting.
- &U** - This variable contains the unit number after a call to the FINDFILE command. If the file was not found it contains blanks if the file was located on multiple units it contains a 99 otherwise it contains the unit number of found file.
- &Z** - This contains the number of the menu item that the user
- &P** - This contains the O/S type: R for real mode, P for protected mode

Section 5: Procedure Commands - Detailed Descriptions

The following pages contain a detailed description of each proc command and the appropriate usage and syntax of each command. Where necessary, examples were provided to add clarity.

* (REMARK CHARACTER)

PURPOSE: Allow remarks to be imbedded in a proc file.

USAGE: *

Example:

```
*  
* THIS IS A PROC TO PERFORM MY MONTH END FUNCTIONS  
*
```

Remarks placed in a procedure file are for internal information purposes to make the proc more readable to proc writers. Placing remarks in a proc help to show what the proc writer has in mind as to the function of various parts of the proc. When the proc executive encounters a remark, no function is performed. The remark lines are simply skipped over.

Note: The asterisk character (*) must be located before any other text on the line in order to function properly.

ASSIGN

PURPOSE: To allow the proc to set one of the nine replaceable parameters to a given value.

USAGE: ASSIGN &<variable#> <contents>

Example:

```
*  
* This is an example of the ASSIGN command  
*  
ASSIGN &1 Hello World  
SHOWLN &1
```

This example will assign the text "Hello World" to the user variable &1. The SHOWLN command following the ASSIGN displays the &1 variable to the display. The ASSIGN command allows the proc writer a means of storing information in the replaceable variables without using the GET\$ to do it. (See also GET\$)

Another function of the ASSIGN command could be to set flags as in many programming languages to alter flow of control in a proc.

Example:

```

*
ASSIGN &1 FLAG1
----- more processing here
----- may set &1 to something else
IF   &1 = FLAG1 GOTO :LABEL1
IF   &1 = FLAG2 GOTO :LABEL2
IF   &1 = FLAG3 GOTO :LABEL3
IF   &1 = FLAG4 GOTO :LABEL4

```

(See also IF, GOTO)

BEEP

PURPOSE: To create an audible noise on the terminal. Usually to signify the completion of some previous command or event.

USAGE: BEEP

Example:

```

* this is a proc to test the BEEP function.
SHOW PRESS ANY KEY WHEN READY
BEEP
WAIT

```

This proc displays the message "PRESS ANY KEY WHEN READY", then beeps at the user, then waits for the user to press a key on the keyboard. (See also "SHOW" and "WAIT")

BEGIN

PURPOSE: Starts a program in the background on a batch, utility, commline or a printer device. Generally used for programs that do not require any user intervention such as sorting a file or printing a report.

USAGE: BEGIN <program> <unit/volser> <device>

Where <program> is the name of the program being executed and <unit> is the disk unit or volume serial in which the program is located. NOTE: for the executive to recognize that a volume serial is being used, it must be preceded by a @ character.

Example:

```

SHOWLN NOW RUNNING DAY END REPORT
BEGIN ;RPT:DAYEND 4 B:
or
BEGIN ;RPT:DAYEND @SR?????? B:

```

This will start program ";RPT:DAYEND" on unit 4 in the first case and on SR?????? in the second case on the first available batch device (B:).

CHAIN

PURPOSE: Transfers control to another program, replacing the proc executive. This is like an EXIT command only it exits the proc to the program of your choice instead of MASTER.

USAGE: CHAIN <program> <unit/volser>

Where <program> is the name of the program to execute and <unit> is the disk unit or volume serial in which the program is located.

Note: For the executive to recognize that a volume serial is being used, it must be preceded by a @ character.

Example:

```
SHOWLN NOW EXITING THE PROC TO RUN A CONVERSION PROGRAM
CHAIN ;SPC:CNVT 4
or
CHAIN ;SPC:CNVT @SR??????
```

This will transfer control to program ";SPC:CNVT" on unit 4 in the first case and on SR?????? in the second case.

Note: Any lines following the CHAIN command will never be executed. The proc is essentially terminated.

CLEAR

PURPOSE: Clears the terminal screen. This makes display less cluttered and easier to read.

USAGE: CLEAR

Example:

```
* proc to test CLEAR screen function
CLEAR
SHOWLN END OF PROC -- PRESS ANY KEY
WAIT
```

This example clears the crt screen displays the message "END OF PROC -- PRESS ANY KEY" and waits for the user to press a key. (See also "SHOWLN" and "WAIT")

DATECALC

PURPOSE: To allow the user to calculate a new date from a given date.

USAGE: DATECALC <date> <amount to add> <new date variable>

Where <date> is the date you wish to calculate from and must be in the format of MM/DD/YY or MM/DD/YYYY. If the date is specified in any other way then a SYNTAX ERROR will be generated.

<amount to add> is the number of DAYS that will be added to or subtracted from the first parameter, the date. If this number is positive (>0) then the days will be added to the date creating a date in the

future of date. If the number is negative (<0) then the days will be subtracted from the date yielding a date that is before the date specified.

<new date variable> is one of the 99 user variables in which to place the newly calculated date. If any thing other than a user variable is specified then a SYNTAX ERROR will be generated. The date returned will be either in the format of MM/DD/YY or MM/DD/YYYY depending on which date format was used in the first parameter.

Example:

```
* This is a simple proc describing the use of the
* DATECALC command.
CLEAR
* Calculate a date 10 days from today.
DATECALC  &D,10,&1
SHOWLN The date that is 10 days from today is &1
WAIT
* Now calculate a date that is 10 days in the past
DATECALC  &D,-10,&1
SHOWLN The date that was 10 days ago is &1
WAIT
*End of Proc
```

DATEDOW

PURPOSE: To allow the user to get the day of the week text from any date.

USAGE: DATEDOW <date> <user variable>

Where <date>, in the format of MM/DD/YY or MM/DD/YYYY, is the date to return the day of the week for. <user variable> is one of the 99 user variables to return the day of the week in.

Example:

```
* This is an example of the DATEDOW command
CLEAR
DATEDOW  &D,&1
SHOWLN  The day of the week for today is ==>&1
WAIT
*End of proc
```

The purpose of this proc is to get the day of the week (i.e. Monday, Tuesday, Wednesday etc.) for the current system date. The day of the week is stored in &1.

DSKCHKON AND DSKCHKOFF

PURPOSE: To allow error checking to be disabled or enabled in conjunction with the ERASE and RENAME command. (See ERASE and RENAME command)

USAGE: DSKCHKON

or

DSKCHKOFF

Proc File Basics

Example:

```
*
* This is an example of the DSKCHKON command
*
DSKCHKON
* From this point on file errors on ERASE, RENAME and RECLEN will be
* reported as error messages.
DSKCHKOFF
* Now error checking and reporting is OFF. Errors are still,
* however, contained in the system variable &E.
```

These two commands will give the proc writer flexibility in handling the ERASE and RENAME functions. Sometimes it is necessary for a proc to disregard file errors in processing a file command (i.e. ERASE and RENAME). For example, you may want to ERASE a file if it exists and continue processing if it doesn't.

ENDTEXT

PURPOSE: To stop displaying text when used in conjunction with the TEXT command. (See TEXT command)

USAGE: ENDTEXT

Example:

```
*
* Proc to show the use of the ENDTEXT command
*
TEXT
These lines will display on the screen
until we come to an ENDTEXT command
ENDTEXT
```

The TEXT/ENDTEXT combination will display the text between these commands up to but not including the ENDTEXT command.

ERASE

PURPOSE: To allow a file to be erased from the disk unit.

USAGE: ERASE <filename> <unit/volume>

Where <filename> is the name of the file you wish to erase. <unit/volume> is the disk unit or volume serial the file is on.

Note: As with the BEGIN function, if a volume serial is specified, it must be preceded with the character "@". (see also "BEGIN")

Example:

```

*
* This is a proc to test the ERASE function
*
ERASE MYFILE 4
*      or
ERASE MYFILE @SR??????

```

This command will erase the file "MYFILE" located on disk unit 4 or as in the next line will delete it off volume SR??????. If the operation is successful the file will be deleted, otherwise, an error message will be displayed.

NOTE WELL: Utmost care must be taken in order not to delete files which are important to the operation of the system (i.e. Libraries, user data files, etc.). This operation completely and totally removes the specified file.

EXIT

PURPOSE: To allow the program to stop execution at some point in the procedure file. This command may be used if the proc is interacting with the user and they type "QUIT" and you want to terminate the proc.

Note: If the proc does not have an EXIT command in it then the proc will terminate after the last executable command in the file. So usage of the EXIT command is optional in some cases.

USAGE: EXIT

Example:

```

SHOW ENTER PROGRAM TO EXECUTE====>
GET$ &1
IF &1 = QUIT EXIT

```

This proc snippet will compare the user variable &1 to the text "QUIT" and will exit the proc if this is true. (see "GET\$" command and "IF" command)

FINDFILE

PURPOSE: This command allows a proc to determine the location of a file, thus eliminating the need for multiple lines of commands in a proc.

USAGE: FINDFILE <filename>

Where <filename> is the name of the file you wish to locate. The command works by searching all hard drives for the specified file and returns the result into one of the system variables (&U). If the file has been located on one of the disk units then the number of that disk unit is placed into &U. If the file was not found at all then &U is blank and can be tested with the &B (blank) system variable. If the file was found on multiple disk units then &U will contain a 99.

Proc File Basics

Example:

```
*
* This is a proc to demonstrate the FINDFILE command
*
SHOW Enter a file to search for ===>
GET% 10 &1
SHOWLN
SHOW Searching.....
FINDFILE &1
SHOWLN
IF   &U = &B THEN
    SHOWLN That file was not found
    WAIT
    EXIT
ENDIF
IF   &U = 99 THEN
    SHOWLN That file was located on multiple disk units
    WAIT
    EXIT
ENDIF
SHOWLN
That file was found on unit &U
WAIT
```

This example performs the simple function of prompting the user for a file name, displays the message "Searching....." and proceeds to try to locate the file using the FINDFILE command. The results of the command are displayed for all the conditions that arise using this command. The file was either not found at all or it was found or it was found in more than one place.

FLUSH

PURPOSE: To allow the proc to flush the screen/program cache available on PGD386 version of the operating system.

USAGE: FLUSH

This proc command is useful if you are writing procs that add new screens or programs to a library. This command causes all resident programs or screens to be reloaded from disk so that any one accessing the programs will get a fresh copy loaded into memory.

Note: This command is only useful for the PGD386 operating system.

GET\$, GET% AND GET#

PURPOSE: To get input from the user into one of nine 30 character variables with variations that allow length of the text to be entered and numbers only to be entered.

USAGE: GET\$ &<variable#>
 or
 GET% <length> &<variable#>
 or
 GET# <length> &<variable#>

Where &<variable#> is one of the nine input variables. The procedure executive will wait until the user enters some text (up to 30 characters long). The input variables are such that &1 is the first variable &2 is the second and so forth up to &99. The contents of the variables will remain intact for the life of the proc or until another GET\$ or ASSIGN command replaces them. (See also ASSIGN)

Example:

```
* proc to demonstrate the GET$ function
CLEAR
SHOW ENTER YOUR NAME =====>
GET$ &1
SHOWLN
SHOWLN YOUR NAME IS &1
SHOW PRESS ANY KEY
WAIT
```

This example clears the screen and displays the message "ENTER YOUR NAME =====>". the proc then lets the user type in their name which gets stored into variable &1. the proc now displays the message "YOUR NAME IS <NAME>" where <NAME> is the text the user typed in. The proc displays the message "PRESS ANY KEY" and waits for the user to press any key on the keyboard.

If you want to specify how large the input field is you can do this by using the GET% command or the GET# command. The GET% command works by allowing you to specify how many characters the user may type. For example:

```
GET% 10 &1
```

This allows the user to enter up to 10 characters which are stored into variable number 1. The GET# works the same way as the GET% but it only accepts numbers from the user. This is useful in retrieving a unit number for a file operation.

GETCHAR

PURPOSE: To get input from the user into one of nine 30 character variables.

USAGE: GETCHAR &<variable#>

Where &<variable#> is one of the nine input variables. The procedure executive will wait until the user enters a character on the terminal. The input variables are such that &1 is the first variable &2 is the second and so forth up to &99. The contents of the variables will remain intact for the life of the proc or until another GETCHAR replaces them.

Proc File Basics

Example:

```
* proc to demonstrate the GETCHAR function
CLEAR
SHOW -- ENTER Y or N =====>
GETCHAR &1
SHOWLN
SHOWLN THE KEY YOU PRESSED IS =====> &1
SHOW PRESS ANY KEY
WAIT
```

This example clears the screen and displays the message "ENTER Y or N =====>". the proc then lets the user type a key on the keyboard which gets stored into variable &1. The proc displays the message "THE KEY YOU PRESSED IS <KEY>" where <KEY> is the key the user typed. The proc then displays the message "PRESS ANY KEY" and waits for the user to press any key on the keyboard.

GETMAIL

PURPOSE: To get a value stored in one of the sixteen internal mailboxes.

USAGE: GETMAIL <whichone> &<variable#>

Where <whichone> is a number from 1 to 16 which is one of the internal mailboxes to retrieve the value from. The user variable is used to place the value contained at the specified mailbox. The values contained in these mailboxes range in the value of -32768 to 32767 and are strictly numeric. Some programs use these mailboxes to store error return numbers when the program abnormally terminates. The GETMAIL command allows a proc to use a similar mechanism. SEE ALSO SETMAIL command.

Example:

```
* This is an example of the GETMAIL command.
CLEAR
* First lets set a value in one of the mailboxes
SETMAIL 10,100
* Now lets retrieve the value we just set and display it
GETMAIL 10,&1
SHOWLN The value we saved in mailbox #10 was =>&1
WAIT
*End of proc
```

The purpose of this proc was to describe how to use the GETMAIL command in conjunction with the SETMAIL command. After clearing the screen, we set mailbox #10 with the value of 100.

Next we use the GETMAIL command to retrieve this value and subsequently we displayed this value on the screen.

Note: There are 16 total mailboxes in the system and these mailboxes are shared by ALL applications. So if you have multiple procs running on multiple devices then it is possible for other devices to change the value of the mailboxes without notice, thus giving inconsistent results. However, this could also allow for multiple procs to "cooperate" with each other by sharing a mailbox between them.

GETUSAGE

PURPOSE: To determine the capacity of a disk unit.

USAGE: GETUSAGE <unit/volume>

Where <unit/volume> is the unit number or volume serial to find the disk capacity on. The capacity of disk unit specified will be expressed as a percentage used of the total capacity.

Example:

```
* This is an example of the GETUSAGE command
CLEAR
GETUSAGE @SR
IF &F > 90 THEN
    SHOW THAT DISK IS TOO FULL!
    WAIT
ENDIF
```

The purpose of this proc is to determine the disk usage of drive volume "SR". All it does is clear the screen then calls the GETUSAGE command to retrieve the disk usage. Next it will use the IF command to figure out if the disk usage is more than 90%. If the usage of that disk is over 90% then it will display a "Too Full" message.

GOSUB

PURPOSE: To call a routine in the proc and return back to where the routine was called. The GOSUB command may be nested 4 levels deep. GOSUB's are useful when you have a chunk proc code that needs to be executed more than once. So, instead of having duplicate sections of proc code doing the same thing, you only need one GOSUB routine.

USAGE: GOSUB <label>

Where <label> is the name of the routine to call. Labels must start with the character ":" and may be followed by any sequence of characters. The end of the routine **MUST** be terminated with a RETURN command. (See RETURN)

Example:

```
*
* This is an example proc to demonstrate the GOSUB command
*
CLEAR
SHOWLN This is an example to show how GOSUB is used.
GOSUB :DISPLAY_PROMPT
CLEAR
SHOWLN This is some more text
GOSUB :DISPLAY_PROMPT
EXIT
* We now start the subroutine
:DISPLAY_PROMPT
POSITIONXY 1 23
SHOW Press any key to continue
WAIT
RETURN
```

Proc File Basics

This example proc simply displays some text messages on the screen and calls the routine :DISPLAY_PROMPT which prompts the user to "Press any key to continue". The subroutine technique allows you to use less commands for commonly used functions. (See also RETURN)

GOTO

PURPOSE: To transfer control of the procedure executive to another command line of the proc.

USAGE: GOTO <label>

Where <label> is the position in the proc in which to transfer control to. Labels must start with the character ":" and may be followed by any sequence of characters.

Example:

```
*
* this is a proc to show how to use the GOTO feature
*
* Clear the terminal screen
CLEAR
* the following command ":START" is a label
:START
* now display a message on the screen
SHOWLN HERE I AM - PRESS A KEY
* let the user terminate the proc by hitting escape if they want
WAIT
* heres a GOTO command now
GOTO :START
```

What this short proc does is simply clear the screen and display the words "HERE I AM - PRESS A KEY" on the display and wait for the user to press a key on which case the cycle starts over and the same message is displayed again. The proc is terminated when the user presses escape.

INC AND DEC

PURPOSE: These commands are used to provide a way of letting procs count up or down.

USAGE: INC &<variable#>

Where &<variable#> is one of the user variables to increment up or decrement down. NOTE: It is up to the proc writer to make sure that a user variable (&1 - &99) contains a number and not text. If the INC or DEC is used on a text variable the results will be undefined.

Example:

```

*
* This is an example to show the use of the INC and DEC commands
*
CLEAR
ASSIGN    &1    1
:COUNT_UP
    IF NOT &1 = 99 THEN
        SHOWAT    10 10  &1
        INC    &1
        GOTO :COUNT_UP
    ENDIF
:COUNT_DOWN
    IF NOT &1 = 1 THEN
        SHOWAT    10 10 &1
        DEC    &1
        GOTO :COUNT_DOWN
    ENDIF
:DONE

```

This example first clears the screen with the CLEAR command. Then we assign a "1" to the user variable &1 to initialize it to a known value. It is generally a good programming practice to initialize variables before we use them. Then we loop around and display the user variable &1 at column 10, row 10 on the screen and then increment it by 1. We keep incrementing until the user variable is equal to 99 in which case we go into a similar loop this time decrementing and displaying the user variable.

IF, IF# - THEN AND ENDIF

PURPOSE: This command allows procedure file commands to be execute if a condition is true (or false).

USAGE: IF <NOT> <str1> <condition> <str2> <COMMAND>

or

IF <NOT> <str1> <condition> <str2> THEN

<COMMAND>

<COMMAND>

ENDIF

or

IF# may be substituted in the above expressions

Where <str1> and <str2> are the text strings to compare and <COMMAND> is a valid procedure file command to execute if the comparison is true. <condition> is the type of comparison to make. Valid conditions are =, < or >. The only difference between the IF command and IF# variation is that instead of comparing two text strings, a numeric comparison is performed. The two variables compared must be whole numeric values and less than or equal to ten digits long.

Proc File Basics

If you want to see if two words equal then you would use a line like this:

```
IF string1 = string2 GOTO :EQUAL_TO
```

If you want to check if one string is alphabetically before another string then you would have a line like this :

```
IF string1 < string2 GOTO :LESS_THAN
```

If you want to check if a string is located alphabetically after another string then your line might look like this :

```
IF string1 > string2 GOTO :GREATER_THAN
```

If the keyword THEN is found instead of a valid command then the commands following the IF command are executed. This allows for faster processing of the IF command if multiple commands are required.

Note: If the IF - THEN form of the command is used then it is necessary to use the ENDIF command after all of the commands.

If the <NOT> keyword is used, as shown, then <COMMAND>(s) will only be executed if the condition is false.

Example:

```
*
* this is a simple proc to show how to use the "IF" command
*
:START
* clear the screen
CLEAR
* ask user to type in their name
SHOW ENTER YOUR NAME (OR QUIT TO EXIT) ==>
* input their name into variable 1 (&1)
GET$ &1
* see if they type "QUIT" and if not, start over
IF NOT &1 = QUIT GOTO :START
SHOWLN
SHOWLN THIS IS THE END - BYE!
```

This simple proc clears the screen and prompts the user for their name or to type "QUIT" if they want to stop. It then saves their keystrokes into the variable "&1". We now check to see if they typed "QUIT" with the line:

```
IF NOT &1 = QUIT GOTO :START
```

which will see if the variable &1 is the text "QUIT" and if not it will start the proc over again.

Another example:

```

*
* Proc to show the IF - THEN - ENDIF usage
*
:START
* clear the screen
CLEAR
* ask user to type in their name
SHOW ENTER YOUR NAME (OR QUIT TO EXIT) ===>
* input their name into variable 1 (&1)
GET$ &1
* see if they type "QUIT" and if not, start over
IF &1 = QUIT THEN
    SHOWLN
    SHOWLN THIS IS THE END - BYE!
    WAIT
    EXIT
ENDIF
GOTO :START

```

This example is essentially the same as the previous example only it shows how the IF - THEN construction is used. It is common in programming languages to indent the commands following the IF <exp> THEN as to show the nesting of commands. IF - THEN commands are virtually unlimited in the nesting capability.

Example:

```

IF <exp> THEN
    IF <exp> THEN
        IF <exp> THEN
            IF <exp> THEN
                <command>
                <command>
            ENDIF
            <command>
        ENDIF
    ENDIF
ENDIF

```

KEYON AND KEYOFF

PURPOSE: These commands determine whether you will allow a proc to be aborted while it is running. Normally, by default, the KEYON command is active which means that during execution of the proc file if a user presses a key a message will be displayed allowing the user to continue processing the proc or abort. Sometimes, with complex menu displays, the user could disrupt the way a proc displays on the screen by pressing a key at the wrong time. The KEYOFF command will make the proc executive ignore keypresses until activated by a KEYON command.

USAGE: KEYON

or

KEYOFF

Proc File Basics

Example:

```
*
* This is proc to show the KEYON and KEYOFF command
*
KEYOFF
- - - - - display a menu or something here with keystroke
- - - - - trapping disabled
KEYON
* reenale keystroke trapping
```

The KEYON command provides a convenient way for "runaway" procs to get under control. For example if you had an endless "loop" in your proc you could press a key and you will be allowed to abort the proc. If you had set the KEYOFF command, however, the proc will run its endless loop without the user able to intervene.

Note: The KEYOFF feature will significantly show a speed increase for proc file processing because keystrokes do not have to be checked in between every line of the proc.

KEYPRESSED

PURPOSE: To allow a proc to determine if a key has been pressed on the keyboard or not.

USAGE: KEYPRESSED &<variable#>

This command is useful for performing background tasks while waiting for the user to press a key on the keyboard. If the user HAD pressed a key then that keystroke is stored in variable parameter you specify. The proc can test to see if a key was pressed by looking at the &K system variable. This variable always contains the result of the last KEYPRESSED operation. If a key was available then the &K variable contains YES otherwise if a key was not available then it contains a NO.

Example:

```
*
* This is a proc to demonstrate the KEYPRESSED command
*
SHOWLN
SHOWLN Press a key to pause the time display
SHOWLN
:WAIT_FOR_KEY
    KEYPRESSED &1
    IF &K = NO GOTO :DISPLAY_TIME
    SHOWLN
    SHOWLN You pressed a key! That key you pressed was &1
    WAIT
GOTO :WAIT_FOR_KEY
* Display the time while not doing anything
:DISPLAY_TIME
    POSITIONXY 10 10
    SHOW &T
GOTO :WAIT_FOR_KEY
```

What this proc does is loop around checking for someone to press a key on the keyboard. Anytime a key has not been pressed, it displays the current system time on column 10, row 10 of the terminal screen. If a key was pressed it displays a message.

LOCK AND UNLOCK

PURPOSE: To allow the proc writer to lock the operating system program loader to prevent other programs from loading.

USAGE: LOCK

or

UNLOCK

This is a system level command of the proc executive which causes the operating system to cease from letting any other terminals or devices load programs. This is a dedicated sort of operation and should be used with care. If an UNLOCK command is not generated sometime before the proc exits then the system could potentially stay locked until the system is rebooted.

MENU AND MENUEND

PURPOSE: To allow the proc writer to generate easy to use selection menus.

USAGE: MENU <col> <row> <promptchar>

<menu selection 1>

<menu selection 2>

<menu selection 3>

<menu selection n>

MENUEND

Where <col> is the column position to start display of the menu selections and <row> is the line position to start displaying. The <promptchar> parameter is optional. This parameter is used to display a character of your choice beside a menu item as the user scrolls through the menu selections. It exists mainly if the user has a terminal that doesn't have a low or hi-intensity capability. The lines in the proc following the MENU key word are the actual menu selections that are displayed on the screen. The maximum number of menu selections is 23. If you try to use any more than 23 a syntax error will result. The key word MENUEND tells the proc executive when to stop displaying menu selections. It is MANDATORY to end your menu selections with this command.

When the menu command is used, it will display the menu selections in a columnar format and highlight the first menu selection. The proc program controls the screen intensity at this point so any SETHI or SETLO commands issued before this will have no effect. The user picks menu selections by pressing the <TAB> key, the <BACKSPACE> key, the letter "U" or the letter "D". These keys will cause the next or previous menu selection to be highlighted, showing the user which option they are picking. The <ESC> key will let the user abort the menu if you choose. The <ENTER> or <RETURN> key will cause the current hi-lighted selection to be chosen.

The results of the menu selection process are stored in the variable &Z. The menu selections are numbered starting at 1 for the first menu selection up 23 for the last menu selection. The last menu selection varies with the number of selection you have in your menu.

Proc File Basics

If the user presses the <ESC> key during the menu sequence, the &Z variable will be blank, which allows you complete control over menu process. MENU and MENUEND cont'd

Example:

```
*
* This is a proc to demonstrate the MENU and MENUEND commands
*
CLEAR
MENU 30 10
    THIS IS MENU SELECTION 1
    THIS IS MENU SELECTION 2
    THIS IS MENU SELECTION 3
    THIS IS MENU SELECTION 4
MENUEND
IF &Z = &B    SHOWAT 1 23 THEY PRESSED ESCAPE!
IF &Z = 1    SHOWAT 1 23 THEY PICKED MENU SELECTION 1
IF &Z = 2    SHOWAT 1 23 THEY PICKED MENU SELECTION 2
IF &Z = 3    SHOWAT 1 23 THEY PICKED MENU SELECTION 3
IF &Z = 4    SHOWAT 1 23 THEY PICKED MENU SELECTION 4
WAIT
```

This proc example first clears the screen then issues a MENU command to tell the proc executive that the next lines up to the MENUEND command are menu selections. The menu selections will start being displayed at column 30, row 10. The next 4 lines after the MENU command will be displayed on the screen with the first selection highlighted. The user can now scroll around these selections and pick one with the <ENTER> or <RETURN> key. The selection that the user made will be displayed at the bottom of the screen. Optionally you could have used the prompt character capability to make the scrolling even more visible to the user.

An example of this format is:

```
MENU 30 10 >
. . . .
. . . .
MENUEND
```

This has the same effect as the other format only a ">" character will be displayed before each currently highlighted menu selection.

PAUSE

PURPOSE: To cause the execution of the procedure file to wait for specific amount time.

USAGE: PAUSE <ticks/seconds> SECONDS

Where <ticks/seconds> is the number of clock ticks or optionally the number of seconds you want the proc to delay.

Note: If seconds instead of ticks is desired then the keyword "SECOND(S)" must follow the number of seconds requested.

Example:

```

*
* proc to demonstrate the PAUSE function
*
PAUSE 36
* or
PAUSE 5 SECONDS

```

The first example will cause the proc to wait 36 clock ticks.

Note: There are approximately 18 ticks in 1 second, so 36 ticks will delay the proc about 2 seconds. The second example will cause the proc to delay for about 5 seconds. The usage of ticks versus seconds is that specifying ticks adds a slightly greater level of control over the length of the delay.

POSITIONXY, POSITIONXYC, POSITIONXYS

PURPOSE: To place the cursor at a specific location on the CRT screen. This command is useful for displaying messages at a specific location on the CRT screen. Variations of this command clear to the end of the line or clear to the end of the screen.

USAGE: POSITIONXY <col> <row>

or

POSITIONXYC <col> <row>

or

POSITIONXYS <col> <row>

Where <col> is the screen column and <row> is the screen row to place the cursor. Most CRT screens have 80 columns by 24 rows.

Example:

```

*
* proc to demonstrate the POSITIONXY function
*
POSITIONXY 10 1
SHOW HELLO WORLD
WAIT

```

This proc excerpt will place the cursor at column 10 row 1 of the CRT screen and display the message "HELLO WORLD"

Variations:

The POSITIONXYC command acts like the POSITIONXY command except that after it positions the cursor to the coordinates specified, it clears off any text from the coordinates to the end of the line.

The POSITIONXYS command like POSITIONXYC command positions the cursor and clears text from that point to the end of the display screen.

PRETTY

PURPOSE: To print a program source file using the pretty print program.

USAGE: PRETTY <filename> <unit/volume> <printer>

Where <filename> is the name of the program file located on <unit/volume>. The printer device is specified by <printer>. Replaceable variables may be used in all of the parameters. This program requires that the ;CMP library exists and the library name "PRETTY" exists in the library.

All options available in the pretty print program will use the defaults.

PRINT

PURPOSE: To print a document file to the printer of choice.

USAGE: PRINT <filename> <unit/volume> <printer> <copies>

Where <filename> is the name of the document file located on <unit/volume>. The printer device is specified by <printer>. For multiple copies of the document, specify in <copies> the number of copies desired. Replaceable variables may be used in all of the parameters for the PRINT command.

Example:

```
*
* This is a proc to test the PRINT command.
*
PRINT MYFILE @SR????? P1 3
WAIT
```

This proc example will cause the document file "MYFILE" located on volume serial SR?????? to be printed on printer "P1". It specifies 3 copies be made of this document.

PRINTWP

PURPOSE: To print a document file to the printer of choice using the new document formatter and printer.

USAGE: PRINTWP <file> <unit/volume> <printer> <copies> <pages>

Where <filename> is the name of the document file located on <unit/volume>. The printer device is specified by <printer>. For multiple copies of the document, specify in <copies> the number of copies desired. The <pages> parameter is used to tell the print program whether to print ALL pages ODD pages or EVEN pages. If this is left off, then the default is ALL pages. Replaceable variables may be used in all of the parameters for the PRINTWP command.

Example:

```
*
* This is a proc to test the PRINTWP command.
*
PRINTWP MYFILE @SR????? P1 3 O
WAIT
```

This proc example will cause the document file "MYFILE" located on volume serial SR?????? to be printed on printer "P1". It specifies 3 copies be made of this document. Only ODD pages will be printed.

RANDOM

PURPOSE: To allow the user to generate a pseudo random number.

USAGE: RANDOM <bounds> <user variable>

Where <bounds> is the maximum value to generate the random number for. <user variable> is the user variable to store the random number result into. This command can be useful for diagnostic work where a random value is need to add a "humanizing" factor to a test. The random number generated will range from 0 to <bounds>.

Example:

```
* This is an example of how to use the RANDOM command
CLEAR
* Lets generate a random number between 0 and 10
RANDOM    10,&1
SHOWLN   The number is =>&1
WAIT
*end of proc
```

This examples uses the RANDOM command to generate a number between 0 and 10 then displays the result on the screen. Possible uses for the random number generator include writing diagnostic or test procs where the need for random data is essential.

REBOOT

PURPOSE: To allow the user to reboot the system.

USAGE: REBOOT ALL

or

REBOOT LOCAL

Where ALL tells the proc executive to reboot all of the machines in a network configuration and LOCAL just reboots the machine the proc is running on. If the system is not a networked system then both commands have the same effect.

RECLEN

PURPOSE: To allow the proc executive to determine the record length of a given file.

USAGE: RECLEN <filename> <unit/volume>

Where <filename> is the name of the file you wish to find the record length for and <unit/volume> is the disk unit or volume serial the file is located. The record length for the file is put into the system variable &R. If the file was not found and disk error checking is ON then an error message will be

Proc File Basics

produced. If disk error checking is OFF then &E system variable will be set to YES if the file is not found. (See also DSKCHKON and DSKCHKOFF command descriptions)

Note: as with the BEGIN function, if a volume serial is specified, it must be preceded with the character "@". (See also "BEGIN")

Example:

```
*
* This is a proc to show the RECLEN command
*
CLEAR
SHOW Enter the name of the file to show the record length for ::
GET$ &1
SHOWLN
SHOW Enter the unit # for that file ::
GET$ &2
RECLEN &1 &2
SHOWLN
SHOW The record length for &1 on unit &2 is &R
```

This example prompts the user for a file name and unit and then uses the RECLEN command to determine the record length of the file.

RENAME

PURPOSE: To allow a file to be renamed to a new name.

USAGE: RENAME <filename> <unit/volume> <newfilename>

Where <filename> is the name of the file you wish to rename and <unit/volume> is the disk unit or volume serial the program is on. <newfilename> is the new name you want to call the file.

Note: as with the BEGIN function, if a volume serial is specified, it must be preceded with the character "@". (See also "BEGIN")

Example:

```
*
* This is a proc to show the usage of the RENAME function.
*
RENAME MYFILE 4 YOURFILE
or
RENAME MYFILE @SR?????? YOURFILE
```

This command will rename the file "MYFILE" located on disk unit 4 to the name of "YOURFILE" or as in the second line will rename "MYFILE" to "YOURFILE" located on volume SR??????. If the RENAME command is successful then the file will be renamed to the new name specified, otherwise, an error message will be issued.

RESET

PURPOSE: To allow the user to reset the special screen area.

USAGE: RESET

This command is used mainly in conjunction with the SAVE and RESTORE command to reset the special screen area that is used for inter-program communication. The RESET command clears this area out leaving it in a default condition to be used by other SAVE or RESTORE commands. SEE ALSO SAVE, RESTORE.

Example:

```
* This is an example of the RESET command
CLEAR
RESET
RUN  SOMEPROG 4
WAIT
* End of proc
```

This proc is an example of the RESET command. It is used mainly for some programs that require that the special screen area be zeroed out in order to function properly.

RESTORE

PURPOSE: Allows the proc writer the ability restore variables from the special program save area. (See also SAVE)

USAGE: RESTORE &<variable#>

Example:

```
*
* This is an example to show the RESTORE command
*
SHOWLN INPUT TEXT BE SAVED
GET$ &1
* now lets save it away
SAVE &1
* now were going to blow that variable away
SHOWLN INPUT MORE TEXT
GET$ &1
SHOWLN you have just input ==> &1
* now lets bring it back to life
RESTORE &1
* we will now display what the user input originally
SHOWLN &1
```

What this snippet of proc does looks complicated but is really simple. The first thing it does is display a message and waits for the user to type in a message. The message is now saved in a special place in program memory. The proc requests input from the user again using the same variable previously. The RESTORE command is brought into action restoring what was saved in the special program memory.

Note: The SAVE and RESTORE commands are a little tricky to use and understand. Normally, these commands are used for inter- program communication if two programs agree on the SAVE and RESTORE sequence. In other words SAVE and RESTORE provide a means for one program located on one device to pass information to another program located on a different device. In the future application program may use this feature to make the proc executive integrated into application programs.

When a proc issues a SAVE command the information will be there until the next SAVE command which will silently overwrite any information previously saved there. RESTORE may repeated be used to restore information put there by a SAVE.

RETURN

PURPOSE: Used with the GOSUB command to return to the point in the proc from where the GOSUB was called

USAGE: RETURN

Example:

See GOSUB command for an example of the RETURN command.

RUN

PURPOSE: To execute a program from a proc with either supplied keystrokes for automatic execution or no supplied keystrokes for interactive programs.

USAGE: RUN <program> <unit/volume> <input characters>

Where <program> is the name of the program you wish to execute. <unit/volume> is the disk unit or volume serial the program is on and <input characters> are the supplied keystrokes for the program.

Note: As with the BEGIN function, if a volume serial is specified, it must be preceded with the character "@". (see also "BEGIN")

Example:

```
RUN ;NEW:MA 4 YACY ~
```

This command will execute the program ";NEW:MA" located on disk unit 4 with the input keystrokes "YACY <ESC>". NOTE: In order to accommodate special keys such as <ESC> and <RETURN> and <TAB> they are given special characters:

```
<ESC>      = ~
<RETURN>   = ^^
<TAB>      = ,
```

If the input characters are not specified then the program will run like any normal application, users will be prompted for the keystrokes where needed.

SAVE

PURPOSE: Allows the proc writer the ability save variables from the special program save area. (See also RESTORE)

USAGE: SAVE &<variable#> <contents>

Example:

```
*
* This is an example to show the SAVE command
*
SHOWLN INPUT TEXT BE SAVED
GET$ &1
* now lets save it away
SAVE &1
* now were going to blow that variable away
SHOWLN INPUT MORE TEXT
GET$ &1
SHOWLN you have just input ==> &1
* now lets bring it back to life
RESTORE &1
SHOWLN &1
```

What this snippet of proc does looks complicated but is really simple. The first thing it does is display a message and waits for the user to type in a message. The message is now saved in a special place in program memory. The proc requests input from the user again using the same variable previously. The RESTORE command is brought into action restoring what was saved in the special program memory.

Note: The SAVE and RESTORE commands are a little tricky to use and understand clearly. Normally, these commands are used for inter- program communication if two programs agree on the SAVE and RESTORE sequence. In other words SAVE and RESTORE provide a means for one program located on one device to pass information to another program located on a different device. In the future, application program may use this feature to make the proc executive integrated easier into application programs.

When a proc issues a SAVE command the information will be there until the next SAVE command which will silently overwrite any information previously saved there. RESTORE may repeatedly be used.

Application Tip:

Normal use of the save command is used to pass application specific information such as the master location or the order writer number to the program being invoked. In fact, most of the online applications use this technique. You can assign values to these variables using the ASSIGN command that can be used by the **SAVE** command in the proc.

Proc File Basics

The layout of this record follows:

Variable Name	Length	Type	Position	Description
GLOBAL START				
TERMINAL	2,0	Binary	1	Terminal ID
UNIT_NBR	2,0	Binary	3	System Libraries Unit
MASTER_LOC	2,0	Numeric	5	Master Location
INQUIRY	1	Alpha	7	Inquiry Action
REQUEST	3	Alpha	8	Program request
WHO	4	Alpha	11	Order writer
END GLOBAL				
				Total Record Size = 14 Bytes

For example:

```
ASSIGN &1  \      99      \
*          ^^^^^^^^^^^^^^^
*          12345678901234
SAVE &1
RUN `;NEW:SR' 4 <put keystrokes here>
```

Now when &1 is used with the **SAVE** command, it has the location '99' stored at the appropriate location (position 5) in the record. Thus, when the program ';NEW:SR' is run it will have the location '99' passed to it as was desired. This is the same as if the program was being run on a terminal that was set up as location 99.

SENDMSG

PURPOSE: To allow text to be sent to a specific device or group of devices.

USAGE: SENDMSG <device> <text>

Where <device> is the logical name of the device to send the text to (i.e. T1, T5, T: etc.) and <text> the textual message to send.

Example:

```
*
* This is an example of the SENDMSG command
*
* Send a message to all terminals
SENDMSG 'T:', 'Starting Backup on &D at &T'
.....<Command to backup system here>.....
IF &E = NO SENDMSG 'T:', 'Backup finished on &D at &T'
IF &E = YES SENDMSG 'T:', 'ERROR occured during backup on &D at &T'
```

This example will send the text 'Starting Backup on <current date> at <current time> to all "T:" devices. The proc executive then executes the commands to backup the system (commands omitted for clarity). Next, based on the result of the error return variable (&E) a message is sent to all "T:" devices indicating the results.

Note: This command works only with programs that intercept mail messages. (i.e. the MASTER programs)

SETDEVICE

PURPOSE: To allow a proc set the device to use for all RUN commands.

USAGE: SETDEVICE <device>

Where <device> is the logical name of the device to use for all subsequent RUN commands (i.e. B1, B:, P: etc.)

Example:

```
*
* This is an example of the SETDEVICE command
*
SETDEVICE  'B:'
RUN  .....<Command to run here>
```

This example causes the RUN command to use the device "B:" to run the program on instead of the default "DY" device.

SETHI

PURPOSE: To allow subsequent text displayed on the screen to be displayed with a hi-intensity attribute. (See also SETLO)

USAGE: SETHI

Example:

```
*
* this will demonstrate the SETHI function
*
SETLO
SHOWLN THIS IS NORMAL INTENSITY
SETHI
SHOWLN THIS IS HI-INTENSITY
```

This example will display the message "THIS IS NORMAL INTENSITY" then it will set hi-intensity and display the message "THIS IS HI- INTENSITY".

SETLO

PURPOSE: To allow subsequent text displayed on the screen to be displayed in normal (low) intensity.

USAGE: SETLO

See "SETHI" command for an example of the usage.

SETMAIL

PURPOSE: To allow the proc to set one of the sixteen internal mail boxes to a values.

USAGE: SETMAIL <mailbox# (1 - 16)> <new value>

Where <mailbox#> is which mailbox to set. This may be in the range of 1 to 16. <new value> is a number in the range of -32768 to 32767 in which to set in the specified mail box.

Example:

```
* This is an example of the SETMAIL command.  
CLEAR  
* First lets set a value in one of the mailboxes  
SETMAIL 10,100  
* Now lets retrieve the value we just set and display it  
GETMAIL 10,&1  
SHOWLN The value we saved in mailbox #10 was =>&1  
WAIT  
*End of proc
```

The purpose of this proc was to describe how to use the SETMAIL command in conjunction with the GETMAIL command. After clearing the screen, we set mailbox #10 with the value of 100.

Next we use the GETMAIL command to retrieve this value and subsequently we displayed this value on the screen.

SHOW

PURPOSE: To display text on the crt screen.

USAGE: SHOW <text>

Where <text> is any text that is to be displayed on the crt.

Example:

```
*  
* This is an example of the SHOW command  
*  
SHOW hello world
```

This will display the message "HELLO WORLD" at the current cursor position on the CRT screen. This command is often used after a "POSITIONXY" command which places the cursor at a specific row and column on the CRT screen.

SHOWAT

PURPOSE: To display text on the CRT screen at a specific location.

USAGE: SHOWAT <col> <row> <text>

Where <col> is the column number and <row> is the row number of the display terminal in which to position the cursor before displaying <text>. Where <text> is any text that is to be displayed on the CRT.

Example:

```
*
* This is an example of the SHOWAT command
*
SHOWAT 10 23 hello world
```

This will display the message "HELLO WORLD" at column 10 row 23 of the display terminal. This command is a degree faster than using POSITIONXY then SHOW commands because the lines don't have to read from the procedure file then executed.

SHOWLN

PURPOSE: To display text on the CRT screen followed by a carriage return.

USAGE: SHOWLN <text>

Where <text> is any text that is to be display on the crt.

Example:

```
*
* This is an example of the SHOWLN command
*
SHOWLN HELLO WORLD
SHOWLN THIS LINE COMES AFTER HELLO WORLD
```

This will display the text "HELLO WORLD" with the line "THIS LINE COMES AFTER HELLO WORLD" displayed on the next line.

STRMOVE

PURPOSE: To allow the user to do surgical operations on a string.

USAGE: STRMOVE <pos1> <string1> <pos2> <destvar> <length>

This command gives you powerful abilities to extract characters from anywhere in a string and insert them anywhere in another string. The parameters are described as follows:

<pos1> is the position to start copying characters from in string1. This number starts at 1 for the beginning of the string and cannot be larger than 160.

<string1> this is string to copy the characters from and can contain any variables needed.

Proc File Basics

<pos2> this is the position in the destination string in which to start copying the characters to. This number can range from 1 to 160.

<destvar> this is the user variable to copy the characters to. This has to be one of the 99 user variables.

<length> this is the number of characters to move from the source string to the destination variable.

Example:

```
* This is an example of the STRMOVE command.
* What we're going to do here is extract the year
* from the system date and display it.
* The system date is in the form of :
* MM/DD/YY
* 12345678 <- This is the position of each character in the date.
* So, what we want to do is to copy 2 characters starting
* from character number 7.
CLEAR
STRMOVE 7,&D,1,&1,2
SHOWLN The current system date year is =>&1
* End of proc
```

What this proc does is simply copy the characters starting from character number 7 for 2 characters. So if the system date is 11/27/88 then this proc will display an 88 on the screen.

A breakdown of the command follows:

```
STRMOVE 7,&D,1,&1,2
Position to start copying from ———┐
This is the system date variable —┐
This is where to copy characters to —┐
This is the user variable to store result —┐
This is the number of characters to copy —┐
```

To extract the day out of the date string you would simply need to modify the STRMOVE command to start copying at the fourth position of the date string:

STRMOVE 4,&D,&1,2 would copy the two day characters from the system date and place them in first user variable.

In these examples the characters copied from the system date have only been placed in the first position of the user variable. You could, however, have placed them anywhere in the user variable. For example, if you have a complex keystroke response line set up for a RUN command you could have placed the portion of the date we extracted above anywhere in that keystroke response line by changing the position to copy the characters to from 1 to any other position.

TEXT

PURPOSE: To display a large block of text on the screen. To stop display of the ENDTEXT command must be used. (See also ENDTEXT)

USAGE: TEXT

Example:

```
*
* Proc to demonstrate the TEXT command
*
CLEAR
TEXT
THE TEXT COMMAND WILL DISPLAY THIS TEXT
UPTO BUT NOT INCLUDING THE ENDTEXT COMMAND
ENDTEXT
```

This example displays the text between the TEXT/ENDTEXT commands. This is useful for displaying large amounts of textual information.

TRACEON

PURPOSE: To give the proc writer a means in which to debug a proc The TRACEON causes the name of the proc command currently executing to be display on the display terminal and will continue to display commands until the TRACEOFF command is reached. (See also TRACEOFF)

USAGE: TRACEON

Example:

```
*
* This is a proc to show the usage of the TRACEON command.
*
SHOWLN PRESS ANY KEY WHEN READY ----
WAIT
TRACEON
SHOWLN TRACING THE PROC NOW....
TRACEOFF
```

This excerpt prompts the user with the message "PRESS ANY KEY WHEN READY ----" and waits for the user to press any key. When the user presses a key this begins the trace mode of the proc executive. If a proc command has multiple parameters it will display these as well, allowing a person to verify what the proc executive really sees when executing a proc.

When this proc is executed the following is displayed on the terminal display:

```
PRESS ANY KEY WHEN READY ----
(User presses a key here)
TRACE IS NOW ON
SHOWLN
TRACING THE PROC NOW....
TRACE IS NOW OFF
```

TRACEOFF

PURPOSE: To give the proc writer a means in which to stop debugging a proc. It is used in conjunction with the TRACEON command described elsewhere. If no TRACEON command was previously issued then the TRACEOFF command will have no effect. (See also TRACEON)

USAGE: TRACEOFF

Example:

(See the TRACEON command for an example of TRACEOFF usage)

WAIT

PURPOSE: To cease execution of the proc until the user presses a key on the terminal.

USAGE: WAIT

Example:

```
*
* This is an example to show the WAIT command
*
SHOWLN PRESS ANY KEY WHEN READY ----
WAIT
```

This excerpt prompts the user with the message "PRESS ANY KEY WHEN READY ----" and waits for the user to press any key.

WAITUNTIL

PURPOSE: To delay execution of the subsequent proc statements until a given time and/or date has elapsed.

USAGE: WAITUNTIL <hh:mm:ss>

or

WAITUNTIL <hh:mm:ss> <day of week>

or

WAITUNTIL <hh:mm:ss> <@date>

Where <hh:mm:ss> is a valid military style time. The second parameter is optional and can be either the day of the week or an actual date. The parameter for the day of the week is the text name of the day spelled out (i.e. MONDAY, Tuesday, wednesday, THURSDAY, Friday. Note that the case of the letters is not important.) Using a day of the week will cause the execution of the proc to suspend until the day of the week specified is reached. To use an actual date as the parameter you must precede the date with an "@" character to indicate that a date is being used. The date specified can be either a standard date format (i.e. @05/31/92; mm/dd/yy) or an extended form of the date with the full year specified (i.e. @05/31/1992)

A special feature of using the date format is that it is possible to "wildcard" on certain digits of the date. This is useful if you want the proc to execute on say, the fifteenth of every month. To use the wildcard feature you simply replace the digits you wish to wildcard on with a question mark character (?). An example of this date in a proc would be "@??/15/????". This will cause the proc to suspend until the day of the 15th of any month or any year arrives. Or an entry of "@05/??/??" will make the proc execute on any day of the month of May.

Example:

```
*
* This is an example of the WAITUNTIL command.
*
WAITUNTIL 01:00:00
* resumes processing here
```

This will cause the proc to wait until 1:00 a.m. at which point it will resume execution. NOTE: it is very important that the time be formatted properly, i.e. with leading zeros and colons. otherwise it will be an error or the proc could hang and never execute.

Another example:

```
*
* This is another example of the WAITUNTIL command.
*
WAITUNTIL 19:00:00 MONDAY
* resumes here on monday
```

This example causes the proc to wait until 5 o'clock on Monday.

Another example:

```
*
* This is an example of using a date with the WAITUNTIL command.
*
WAITUNTIL 19:00:00 @??/10/92
* resumes here
```

This example causes the proc executive to execute at 5 o'clock on the tenth day of every month in the year of 1992.

Section 6: Error Messages

When the proc executive encounters an error it cannot handle in the procedure file it will display the general nature of the error, then it will show the line of the proc the error was found on and prompt the user with a message.

Example:

suppose the following line is in your proc-

```
RUN ;NEW:MA 56
```

Note that an invalid unit (56) is specified for this program. When the procedure executive tries to execute this line it will encounter it as an error and display the following:

```
INVALID UNIT SPECIFICATION  
ERROR ON LINE : 12  
RUN ;NEW:MA 56  
ABORT THIS PROC? Y/N
```

You will be asked to enter a "Y" or "N" to continue or abort the proc completely. If you say "N" to not abort the proc the following message is displayed:

```
RE-TYPE THIS LINE ? Y/N
```

In which case you may opt to either let the proc continue at the next instruction or you may correct the error and the proc executive will do the line over again.

```
REPLACE LINE IN PROC FILE? Y/N
```

Responding "Y" to this questing will update the procedure file with the newly typed in correction for this line so the next time the procedure interpreter encounters this line it will be correct. It allows a proc to be fixed while the proc writer is setting up a new proc without having to go into the editor, make the change and run the proc again.

Section 7: Error Message Descriptions

The following is a list of messages displayed and their probable cause:

```
MESSAGE:
"INVALID FILE SPECIFICATION OR FILE NOT FOUND"
CAUSE:
```

The procedure executive tried to run a program that could not be located on the specified disk. Check the disk for the file and modify the proc appropriately.

```
MESSAGE:
"INVALID UNIT OR VOLUME SPECIFICATION"
CAUSE:
```

The unit or volume serial specified for a "RUN" or "BEGIN" command was invalid. Change the proc to reflect the correct unit or volume serial and re-run the proc.

```
MESSAGE:
"NO ENDTEXT COMMAND SPECIFIED"
CAUSE:
```

This is encountered when the proc executive ran to the end of file when displaying text used with the TEXT command. You must use the ENDTEXT command to terminate the display of text.

```
MESSAGE:
"GOSUBS NESTED TOO DEEP - MAXIMUM IS 4 LEVELS"
CAUSE:
```

This message is encountered when calling a routine using the GOSUB command. GOSUB commands are restricted to 4 levels of nesting.

```
MESSAGE:
"RETURN COMMAND WITHOUT GOSUB"
CAUSE:
```

This message is caused when the proc executive encounters a

```
MESSAGE:
"GOTO/GOSUB LABEL NOT FOUND!!!"
CAUSE:
```

The proc specified a GOTO or GOSUB label that did not exist anywhere in the proc. Make sure that all GOTO/GOSUB label are preceded by a colon (:) and are on a line all by themselves.

```
MESSAGE:
"ENDIF FOR IF <exp> THEN COMMAND NOT FOUND"
CAUSE:
```

Proc File Basics

This error occurs when the proc could not find a matching ENDIF command for an IF - THEN command. Make sure that there are enough ENDIF's for all IF - THEN's.

```
MESSAGE :  
"NO MENUEND COMMAND SPECIFIED"  
CAUSE :
```

This error occurs when the proc could not find a matching MENUEND command for a MENU command. Make sure that there is a MENUEND command for all MENU commands.

```
MESSAGE :  
"PROCEDURE SYNTAX ERROR---"  
CAUSE :
```

This message is encountered when there is some discrepancy with some parameter of a proc command, for example, an invalid time is entered, or an improper command format, etc. Fix the line in the proc and re-run it.

Section 8: A Proc Demonstration

The following proc is a demonstration of many of the features of Interactive Procedure Language. It can be typed in as shown and run using the instructions described in the first part of this manual.

```

*
* PROC TO DEMO THE NEW PROC INTERPRETER
*
* The proc starts here ----
*
:START_PROC
  CLEAR
  ASSIGN &9
  SETLO
  SHOWLN -----
  SHOWLN PROCEDURE EXECUTIVE DEMONSTRATION PROC - VERSION 1.0
  SHOWLN -----
  SETHI
  * Display the initial system time and date
  POSITIONXY    1    4
  SHOW &D
  POSITIONXY    65   4
  SHOW &T
  * Lets see what kind of computer we have here
  IF &C = ASST THEN
  POSITIONXY    21   4
  SHOW THIS IS AN ASSISTANT COMPUTER
  ENDIF
  IF &C = OPD THEN
  POSITIONXY    21   4
  SHOW THIS IS AN OP-DEALER COMPUTER
  ENDIF
  *
  * The next confusing looking lines show the menu and
  * highlights the letters in front of each selection
  * (a cleaner way to do this might be to use the MENU
  * command)
  SETLO
  POSITIONXY    20   10
  SETHI
  SHOW A.
  SETLO
  SHOW RUN A PROGRAM
  POSITIONXY    20   11
  SETHI
  SHOW B.
  SETLO
  SHOW FIND A FILE
  POSITIONXY    20   12
  SETHI
  SHOW C.
  SETLO
  SHOW PRINT A DOCUMENT
  POSITIONXY    20   13
  SETHI
  SHOW X.

```

Proc File Basics

```
SETLO
SHOW EXIT PROC TEST
POSITIONXY 20 16
SETHI
SHOW ENTER LETTER OF COMMAND TO TEST === [ ]
SETLO
:GET_INPUT
POSITIONXY 57 16
PAUSE 20
KEYPRESSED &9
IF &K = NO GOTO :SHOW_DATE
* IF WE GET PAST HERE, SOMEONE PRESSED A KEY
IF &9 = A GOSUB :RUN_PROGRAM
IF &9 = B GOSUB :FIND_FILE
IF &9 = C GOSUB :PRINT_FILE
IF &9 = X THEN
    POSITIONXY 20 23
    SHOW END OF PROC --- PRESS ANY KEY TO EXIT
    WAIT
    EXIT
ENDIF
GOTO :START_PROC
* Show the system date and time on the top part of the screen
:SHOW_DATE
POSITIONXY 1 4
SHOW &D
POSITIONXY 65 4
SHOW &T
GOTO :GET_INPUT
*
* RUN A PROGRAM
*
:RUN_PROGRAM
CLEAR
POSITIONXYS 10 10
SHOW ENTER THE PROGRAM TO RUN - ENTER "QUIT" EXIT
SETHI
POSITIONXYC 10 11
SHOW PROGRAM NAME <<_____>> ON UNIT <<__>>
SETLO
POSITIONXY 25 11
GET% 10 &1
IF &1 = QUIT RETURN
POSITIONXY 48 11
GET% 2 &2
RUN &1 &2
* CHECK FOR NON EXISTENT PROGRAM
IF &E = YES THEN
    POSITIONXYC 10 12
    SHOW SORRY - THAT PROGRAM WAS NOT FOUND
    WAIT
    GOTO :RUN_PROGRAM
ENDIF
GOTO :RUN_PROGRAM
*
* FIND A FILE FOR THE USER
*
```

```

:FIND_FILE
  CLEAR
  POSITIONXYS      10   10
  SHOW ENTER FILE TO SEARCH FOR - ENTER "QUIT" STOP FINDING F
  POSITIONXYC      10   11
  SHOW <<_____>>
  POSITIONXY       12   11
  GET% 10 &1
  IF  &1 = QUIT RETURN
  FINDFILE  &1
  * CHECK FOR BLANK RETURNED UNIT #
  IF  &U = &B THEN
    POSITIONXYC 10   12
    SHOW THAT FILE WAS NOT FOUND
    WAIT
    GOTO :FIND_FILE
  ENDIF
  IF  &U = 99 THEN
    POSITIONXYC 10   12
    SHOW THAT FILE EXISTS IN MORE THAN ONE PLACE
    WAIT
    GOTO :FIND_FILE
  ENDIF
  POSITIONXYC 10   12
  SHOW THAT FILE WAS FOUND ON UNIT &U -- PRESS ANY KEY ---
  WAIT
GOTO :FIND_FILE
*
* PRINT A DOCUMENT
*
:PRINT_FILE
  CLEAR
  POSITIONXYS      10   10
  SHOW ENTER THE DOCUMENT TO PRINT - ENTER "QUIT" EXIT
  SETHI
  POSITIONXYC      10   11
  SHOW DOCUMENT NAME <<_____>> ON UNIT <<__>> ON PRINTER <<__>>
  SETLO
  POSITIONXY       26   11
  GET% 10 &1
  IF  &1 = QUIT RETURN
  POSITIONXY       49   11
  GET% 2 &2
  POSITIONXY       67   11
  GET% 2 &3
  PRINT &1 &2 &3 1
  * CHECK FOR NON EXISTENT DOCUMENT
  IF  &E = YES THEN
    POSITIONXYC 10   12
    SHOW SORRY - THAT DOCUMENT WAS NOT FOUND
    WAIT
    GOTO :PRINT_FILE
  ENDIF
GOTO :PRINT_FILE
*
* E N D   O F   P R O C
*

```

