P R O M A L

(**PRO**grammer's **M**icro **A**pplication **L**anguage)

LIBRARY MANUAL

For Apple II and Commodore 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609
(919) 878-3600

Rev. C - Sep. 1986

PROMAL LIBRARY

The PROMAL system contains a LIBRARY of predefined procedures and functions which are automatically loaded into memory when PROMAL boots up. This library provides input, output and utility routines which greatly ease the programmer's job. To use any or all of the LIBRARY routines, you simply need to add the statement:

```
INCLUDE LIBRARY
```

near the start of your program. This statement tells the COMPILER to read the definitions for the standard library. This is just a list of external procedures and subroutines. You can display the list with a **TYPE L** command from the EXECUTIVE.. The INCLUDE LIBRARY statement does NOT make your program any larger. The library routines are always present in memory; the INCLUDE LIBRARY statement merely tells the compiler what the names are and where they are.

Once you have defined the routines in the LIBRARY with the INCLUDE statement, you may freely use them in your program. You call LIBRARY routines in the same manner as other PROMAL routines. Unlike normal PROMAL routines, however, most LIBRARY routines can be called with a **variable number of arguments**, some of which are optional. For example, the LIBRARY function PUT can have one or more arguments. In most cases, the optional arguments have a default value, which will suffice for most cases. For special cases, you can specify additional arguments which modify the way the routine works.

Table 1: LIBRARY Summary

Name	Avail.*		Description
ABORT	**AC**	L	Abort program execution, optionally displaying message.
ABS	**AC**	L	Absolute value of REAL value.
ALPHA	**AC**	L	Test if character is alphabetic.
BLKMOV	**AC**	L	Block move.
CHKSUM	**AC**	L	Compute 16 bit checksum of memory region.
CLOSE	**AC**	L	Close an open file or device.
CMPSTR	**AC**	L	Compare strings.
CURCOL	**AC**	L	Determine current cursor column.
CURLINE	**AC**	L	Determine current cursor line.
CURSET	**AC**	L	Position cursor on the display.
DIR	**AC**	L	Display file names matching a pattern
DIROPEN	**A**	P	Open disk directory for reading.
EDLINE	**AC**	L	Edit a line on the screen.
EXIT	**AC**	L	Exit from program, optionally with message.
FILL	**AC**	L	Fill a memory block with a constant.
FKEYGET	**AC**	L	Get a current function key definition.
FKEYSET	**AC**	L	Define a function key expansion string.
GETARGS	**AC**	L	Split command line into arguments.
GETBLKF	**AC**	L	Block-read from file.
GETC	**AC**	L	Input one character from keyboard with echo to screen.

GETCF	**AC**	L	Input one char. from a file or device.
GETKEY	**AC**	L	Input character from keyboard, no echo to screen.
GETL	**AC**	L	Input one line from the keyboard.
GETLF	**AC**	L	Input one line from a file or device.
GETPOSF	**A**	L	Obtain current file position.
GETTST	**AC**	P	Test if T device is ready (serial port)
GETVER	**AC**	P	Return PROMAL version and machine code.
INLINE	**AC**	L	Input a line with screen editing.
INLIST	AC	P	Search linked list.
INSET	**AC**	L	Test if a character is in a string.
INTSTR	**AC**	L	Convert signed value to a string.
JSR	**AC**	P	Call machine language subroutine.
LENSTR	**AC**	L	Return length of string.
LOAD	**AC**	P	Load/unload/execute program or overlay.
LOOKSTR	**AC**	L	Search a list of strings.
MAX	**AC**	L	Return the largest of two or more arguments.
MIN	**AC**	L	Return the smallest of two or more arguments.
MLGET	**AC**	P	Load machine language program or memory image.
MOVSTR	**AC**	L	String copy or substring or concatenate strings.
NUMERIC	**AC**	L	Test if character is numeric.
ONLINE	**A**	P	Get ProDOS volume name for specified disk drive.
OPEN	**AC**	L	Open a file or device for input/output.
OUTPUT	**AC**	L	Formatted output with many options.
OUTPUTF	**AC**	L	Formatted output to a file.
PROQUIT	**AC**	P	Exit from PROMAL system.
PUT	**AC**	L	Output text to the display.
PUTBLKF	**AC**	L	Block-write to file.
PUTF	**AC**	L	Output text to a file or device.
RANDOM	**AC**	L	Obtain a pseudo-random number.
REALSTR	**AC**	L	Convert a REAL value to a string.
REDIRECT	**AC**	P	Redirect standard input/output to file/device.
RENAME	**AC**	L	Rename a file.
SETPOSF	**A**	L	Set desired position in a file (random access).
SETPREFIX	**A**	L	Set the path name for directory searches.
STRREAL	**AC**	L	Convert a string to a REAL numeric value.
STRVAL	**AC**	L	Convert a string to a numeric value.
SUBSTR	**AC**	L	Search for one string in another.
TESTKEY	**AC**	L	Test if a key is pressed on keyboard.
TOUPPER	**AC**	L	Fold lowercase letter to upper case
WORDSTR	**AC**	L	Convert an unsigned value to a string.
ZAPFILE	**AC**	L	Delete a file.

*** Note:** Avail. meaning: First column indicates machine availability, A=Apple II, C=Commodore 64. Second column indicates the required INCLUDE file, L=LIBRARY, P=PROSYS.

A few unusual or system-dependent routines are defined in a separate file called PROSYS.S. These routines are also always resident in memory, but in order to use them you need to add the statement:

```
INCLUDE PROSYS
```

near the beginning of your program. This file also defines some less-frequently-needed system variables.

The LIBRARY routines are summarized above. The "Avail." column indicates which computers are supported (Apple or Commodore) and which INCLUDE file is needed in order to use the routine (LIBRARY or PROSYS).

HOW TO USE THE LIBRARY ROUTINE DESCRIPTIONS

The following section provides a detailed description of each routine in the LIBRARY, in alphabetical order. The USAGE line gives the syntax for calling the routine. Words shown in CAPITAL LETTERS are required to be entered as shown (although they do not have to be typed using upper case letters). Words shown in lower case with the first letter capitalized are user-supplied arguments. These arguments can be variables or constants or more complex expressions. Arguments shown in square brackets, [and], are optional arguments which may be included or left off at the programmer's discretion. The description of the routine will tell what actions are taken if the optional arguments are not supplied. Ellipsis (...) are used to indicate an arbitrary number of repetitions of an optional argument. For functions, the USAGE line will show an assignment statement with the type of result returned indicated by the variable name. For example:

```
USAGE:   Bytevar = GETC [(#Variable)]
```

shows that the function GETC has one optional argument which must be enclosed in parentheses if given, and returns a function result of type BYTE. The # symbol is used in the USAGE line to emphasize that the optional argument must be the address of the variable, not its value.

Some routines may have more than one optional argument, in which case some or all may be specified. It is permissable to refer to the same LIBRARY routine with different numbers of optional arguments specified in the same program.

PROC ABORT ABORT PROGRAM

USAGE: **ABORT** [Arglist]

ABORT is a procedure which does not return to the calling program but instead exits to the EXECUTIVE. Optionally, it may contain any arguments that are legal for procedure OUTPUT, which will be output to the display.

EXAMPLE 1:

```
INCLUDE LIBRARY
DATA WORD OLDFILENAME = "MYFILE.T"
...
BEGIN
...
ABORT "CAN'T FIND FILE #S", OLDFILENAME
```

will display an error message on the display and abort to the PROMAL EXECUTIVE. See OUTPUT for a description of the arguments which may be used.

FUNC ABS ABSOLUTE VALUE

USAGE: Realvar = **ABS** (Value)

Function **ABS** returns the absolute value of a real number. The function returns type REAL. ABS(X) returns X if X is positive and -X if X is negative.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
REAL X
REAL Y
...
Y = ABS(X-1.)/2.
...
```

NOTE:
1. In PROMAL version 2.0 and earlier, ABS was not included in the the standard LIBRARY, but was in file REALFUNC.S instead. For version 2.1 and later, it is in the LIBRARY for improved convenience, performance, and compatibility with IBM PROMAL.

FUNC ALPHA TEST IF CHARACTER IS ALPHABETIC

USAGE: Bytevar = **ALPHA**(Char)

Function ALPHA returns TRUE if the argument is alphabetic. The argument **Char** is expected to be type BYTE (not a string!). Both upper and lower case letters will return TRUE.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
CHAR=GETC
IF ALPHA(CHAR)
  ...
```

PROC BLKMOV COPY BLOCK OF MEMORY

USAGE: **BLKMOV** #From, #To, Count

BLKMOV is a procedure for copying a block of memory to another location. **#From** is the starting address. **Count** is the number of bytes to copy. **#To** is the destination starting address. The block being copied can overlap the destination without a problem.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD VALS [200]
BYTE BUF[40]
WORD I
...
BEGIN
...
BLKMOV $0400, BUF, 40
```

copies 40 bytes (decimal) starting at $0400 to BUF.

EXAMPLE 2:

```
BLKMOV #VALS[I], BUF, $8
```

moves 8 bytes (4 words) starting at the Ith word of VALS to BUF. Note the **#** operator before VALS which is required for proper operation.

FUNC CHKSUM COMPUTE CHECKSUM OF BLOCK OF MEMORY

USAGE: Wordvar = **CHKSUM**(#Start, Size)

Function CHKSUM computes the 16 bit checksum of a block of bytes in memory starting at address **Start**. **Size** is the number of bytes to checksum. The returned value is the sum of all the bytes, modulo 65536.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE ARRAY[1000]
WORD ARYCHK
...
ARYCHK = CHKSUM(ARRAY,1000)
...
IF CHKSUM(ARRAY,1000) <> ARYCHK
  PUT NL,"ARRAY has been modified!"
  ...
```

PROC CLOSE CLOSE FILE OR DEVICE

USAGE: **CLOSE** Handle

CLOSE is a procedure which closes a specified word file **Handle**. The argument **Handle** must be the handle for a previously opened file.

EXAMPLE 1:

```
INCLUDE LIBRARY
WORD INPUTFILE      ;File handle
BEGIN
...
INPUTFILE=OPEN(INPUTFILE)
...
CLOSE INPUTFILE     ;Done with file
...
```

It is not normally necessary to close files in a program since all open files will be closed automatically by the EXECUTIVE when exiting from a program. If you are planning to generate stand-alone application programs which will be run without the EXECUTIVE (as described in the PROMAL DEVELOPER'S GUIDE), then you should be careful to close all files, since they will not be automatically closed. Also if you work with several files in a program, it is a good idea to close a file as soon as it is no longer needed, since a limited number of files may be open at once. A power failure or other system failure may leave a file written to disk incomplete unless it has been closed. Be careful not to close a file which you have already closed previously.

FUNC CMPSTR COMPARE STRINGS

USAGE: Bvar = **CMPSTR**(String1, Op, String2 [,Fold [,Limit]])

Function CMPSTR compares two strings. **String1** and **String2** are the addresses of the two strings to be compared, and **Op** is a string (not a character!) specifying which compare operation is desired, chosen from:

"<" "<=" "<>" "=" ">=" ">"

Fold is an optional Boolean (BYTE) argument defaulting to FALSE which, if TRUE, will cause lower case letters to be considered as equal to their upper case equivalents. **Limit** is an optional argument specifying the maximum number of characters to compare in the string, defaulting to 255. The collating sequence for the comparison is the ASCII character set. Two strings are equal if and only if they are the same length and have the same content (or equivalent if Fold is TRUE). If two strings of different lengths match up to the end of the shorter string, the longer string is considered greater.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE LINE[81]
...
GETL LINE
IF CMPSTR (LINE, ">", "M")
```

tests if the string LINE is greater than "M".

EXAMPLE 2:

```
DATA WORD KEYWORD="DRAW","MOVE","ERASE","QUIT",0
...

I=0
REPEAT
  IF CMPSTR (LINE, "=", KEYWORD[I], TRUE, LENSTR(KEYWORD[I])
    ...
  I=I+1
UNTIL KEYWORD[I]=0
```

tests if the string LINE matches the Ith keyword of a table, up to the length of the keyword. (Note: See function LOOKSTR for a better way to do this).

FUNC CURCOL RETURN CURRENT COLUMN OF CURSOR

USAGE: Bytevar = **CURCOL**

Function CURCOL returns the current column number of the text cursor on the screen. The leftmost column is column 0, not column number 1.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
?A
CON MAXCOL = 79  ; Screen width-1, Apple
?
?C
CON MAXCOL = 39  ; Screen width-1, Commodore
?
...
BEGIN
...
IF MAXCOL-CURCOL < LENSTR (STRING) ; won't fit entirely on current line?
  PUT NL
PUT STRING
...
```

FUNC CURLINE RETURN CURRENT LINE NUMBER OF CURSOR

USAGE: Bytevar = **CURLINE**

Function CURLINE returns the current line number of the text cursor on the screen. The topmost line is line 0, not line number 1.

EXAMPLE:

```
INCLUDE LIBRARY
BYTE SAVELOC
...
BEGIN
...
SAVELOC=CURLINE     ;Save line we're on
CURSET 0,0          ;Move to home position
PUT "Error, please try again."
CURSET 0,SAVELOC    ;Back to where we were, col 1.
```

PROC CURSET SET CURSOR POSITION

USAGE: **CURSET** Column, Line

Procedure CURSET sets the screen cursor to a specified column and line. **Column** is the desired column, and **Line** is the desired line number. The home position on the screen (the upper left hand corner of the screen) is location (0,0) not (1,1).

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE I
...
BEGIN
...
CURSET 0,I
```

moves the cursor to the first column of text row I on the screen.

FUNC DIR EXAMINE DISK DIRECTORY

USAGE: Intvar = **DIR**(Pattern [,Mode])

Function DIR displays the names of any files in a disk directory. For the Apple II, **Pattern** is the desired directory name. No filenames or wildcards are recognized. For the Commodore 64, **Pattern** is a filename string which may include wildcards * and ?. The **Pattern** may optionally have a drive number prefix and a file extension (which can also be a wildcard). The * wildcard matches ANY string and the ? wildcard matches any single character. The function returns an INTeger value indicating the number of files which matched the **Pattern** (including subdirectories for the Apple) if positive or 0, or minus an error code if negative. The absolute value of the error code has the same meaning as for IOERROR for function OPEN. **Mode** is an optional argument, defaulting to 1. If **Mode** is 1 or unspecified, a normal display of file names is made. If **Mode**=0, then the directory will be tested for matching entries and **Intvar** returned, but nothing will be displayed. Alternatively, **Mode** can be an open file handle. In this case, the output is directed to this file or device instead of the screen.

For the Apple II, the format of the output display will be the same as for the FILES command in the EXECUTIVE if the output is to the screen, or will be one filename per line for any other file or device. For the Commodore 64, the format of the output display will be the same as for Commodore BASIC. The pattern matching is performed by the Commodore ROMs resident in the disk drive, and therefore operates as described in the Commodore disk manual. In particular, you should note that a pattern of "*.S" will **not** match all the files ending in ".S", but will instead match ALL the files on the disk. This is because Commodore has chosen to implement the "*" wildcard to mean, "match anything at all" (including ".").

EXAMPLE 1 (for **COMMODORE 64**):

```
INCLUDE LIBRARY
...
BEGIN
...
IF DIR("OLDFILE.D")=1      ; file exists?
  PUT NL,"Want to use existing file?" ...
```

EXAMPLE 2 (for **Apple II**):

```
INCLUDE LIBRARY
...
DATA WORD SUBDIR="ACCOUNTS/" ; Sub-directory in current prefix.
WORD NUMACCTS
...
NUMACCTS=DIR(SUBDIR)  ; Display file name in our sub-directory
...
```

NOTE:
1. For the Apple, any file name part will be ignored. For example, "2:ACCOUNTS/MYFILE.T" is equivalent to "2:ACCOUNTS/".

FUNC DIROPEN **APPLE II ONLY** OPEN DIRECTORY FOR READING

USAGE: Handle = **DIROPEN**(Dirname [, Mode])

Function DIROPEN is used to open a disk directory for reading on the Apple II. **Dirname** is a string specifying the directory name. **Mode** is the optional access mode character, which must be 'R' (read access) if specified. Opening a directory for writing is not permitted by ProDOS. DIROPEN returns a file handle (type WORD) as in a normal OPEN function, if successful. Once opened, the directory can be read like an ordinary file. Please consult the ProDOS reference manual for information on directory organization.

EXAMPLE 1:

```
...
INCLUDE LIBRARY
INCLUDE PROSYS
DATA WORD PATH = "2:"
WORD DIRHANDLE
...
BEGIN
...
DIRHANDLE = DIROPEN(PATH)
IF DIRHANDLE = 0
  PUT NL,"Can't open directory for drive 2"
...
```


NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program in order to use DIROPEN.
2. Although DIROPEN is only available on the Apple, you may open a Commodore 64 directory using the OPEN function (see OPEN).
3. The file PRODOSCALLS.S contains examples of a way to get or set file attributes on the Apple without reading the directory.

FUNC EDLINE EDIT LINE ON SCREEN

USAGE: Bytevar = **EDLINE**(String [,Limit [,Mode [,#Col]]])

Function EDLINE is used to allow on-screen editing of a single line of text in the same way as is supported by the PROMAL EXECUTIVE. **String** is the address of the string to be displayed and edited in place. **String** should be the address of a buffer **large enough to hold at least Limit+1 characters.** **Limit** is an optional parameter defaulting to 80 which is the maximum number of characters acceptable in the line. **Mode** is an optional argument defaulting to $00 which controls several options based on individual bits in **Mode**, as follows:

Bit 0 = 1 (Mode=$01) means display the line in reverse video (highlighted).
0 means display the line in normal video.
Bit 1 = 1 (Mode=$02) means return "raw" function key codes from the keyboard.
0 means expand the function keys to their current definitions (see FKEYSET).
Bit 2 = 1 (Mode=$04) means return "strange" control keys (explained below).
0 means ignore "strange" control keys.
Bit 3 = 1 (Mode=$08) means initially display cursor at the column specified in the BYTE variable **Col**, if specified, otherwise at the first character.
0 means initially display cursor after last character.

The last optional argument, **#Col**, is ignored unless bit 3 of **Mode** is 1. In this case, **#Col** is the **address** of a variable of type BYTE which contains the desired starting column for the cursor. If the specified column is greater than the length of the line, the cursor will be positioned immediately after the last character. On exit from EDLINE, the variable Col is updated to the position of the cursor at the time of exit from EDLINE.

Mode bits may be combined. For example Mode=$09 enables reverse video and positions the cursor at the start of the field instead of the end (assuming no **#Col** argument is specified).

Function EDLINE returns a byte which is the terminator entered. For normal **Mode**, this will be a carriage return ($0D). However, if bits 1 and/or 2 of **Mode** are 1, it could be a function key, cursor up/down key, control key, etc. "Strange" control keys are defined as those control keys which are not allowed for line editing, or keys returning a value greater than $7F other than function keys. See **Appendix B** for key code values. For the Apple, function keys are defined as either Apple key in conjunction with a number key 1 though 8.

When EDLINE is called, it will display the **String** passed (which can be null), starting at the current cursor position. It will then output enough blanks so that a total of **Limit** characters are displayed. This is particularly useful when bit 0 of **Mode** is set to 1 to select reverse video, since EDLINE will display a reverse video "box" indicating the allowable "field size" on the screen. EDLINE will then position the cursor after the last character of the string (assuming bit 3 of mode is not set) and wait for keyboard input. All line editing keys allowed by the PROMAL EXECUTIVE can be used in the same manner with EDLINE, including CTRL-B to recall a prior line entry. See **Table 1** of the PROMAL USER'S GUIDE for a complete list of supported editing keys. The **String** will be edited "in place".

Note that during input, the cursor is held "captive" in the limits of the line, making it suitable for various kinds of data entry. By setting **Mode** appropriately, EDLINE can become the basis of an editor or field-oriented data entry system. By setting bits 2 and 3 (Mode=$08+$04) and specifying **#Col**, you can detect, for example, when a "cursor up" key is entered (by the value returned by the function), and what column the cursor was in at the time (by the value returned in Col). You could then call EDLINE again to edit a string which is on the line above the present line on the screen, with the same arguments, and the cursor would appear initially in the same column as on the previous line.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE DUMMY
BYTE BUFFER[81]
...
BEGIN
...
MOVSTR "ERASE ",BUFFER
DUMMY=EDLINE(BUFFER) ; let user complete or change the command
...
```

This program fragment will display the word "ERASE" on the screen, followed by a blank and the cursor. The user could then complete the command as desired.

EXAMPLE 2:

```
INCLUDE LIBRARY

BYTE LINE[41]
BYTE COL
BYTE KEY
...
MOVSTR "This is a line to be edited.", LINE
COL=3
CURSET 8,0
KEY = EDLINE(LINE, 40, $0F, #COL)
...
```


This program fragment will display the specified string starting at the 8th column of the first line on the screen, in a reverse video "box" 40 characters long (which will wrap around to the next line on the Commodore 64), and will position the cursor on the "s" in "This". The line can then be edited by the user in the usual manner. When a "strange" key (such as cursor up or down) is entered, EDLINE returns the edited string in LINE, sets KEY to the key code for the strange key, and updates Col to the cursor position at the time the key was pressed.

NOTE:
1. It is possible, with care, to change almost all of the choices for editing keys for EDLINE, as well as the cursor blink rate. You can also disable CTRL-C (for the Apple) or other editing keys if you wish. See **Appendix G.**

PROC EXIT EXIT FROM PROGRAM

USAGE: **EXIT** [Arglist]

EXIT is a procedure which does not return to the calling program but instead exits to the EXECUTIVE (or to the parent program if this program was LOADed by another). Optionally, the call may contain any arguments that are legal for procedure OUTPUT, which will be output to the display.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
EXIT "Program Complete."
```

This will display a message on the display and exit to the PROMAL EXECUTIVE. See procedure OUTPUT for a description of what arguments may be used.

PROC FILL FILL A BLOCK OF MEMORY WITH A CONSTANT

USAGE: **FILL** #From, Count [,Byteval]

FILL is a procedure which fills a block of memory with a specified value of type BYTE. **#From** is the desired starting address. **Count** is the number of bytes to fill. The optional argument **Byteval** is the value to be placed in each byte, defaulting to $00. FILL operates much faster than a programmed loop.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
CON BUFSIZE = 500
BYTE BUF[BUFSIZE]
...
FILL BUF, BUFSIZE
```

zeroes the array BUF of size BUFSIZE bytes.

EXAMPLE 2:

```
BYTE MYSTRING[20]
...
FILL MYSTRING, LENSTR(MYSTRING),
```

blank fills the string MYSTRING up to its present length.

When using FILL to zero an array of type WORD or INT, remember that the size needed for the second argument should be twice the array dimension (six times for REAL).

PROC FKEYGET GET A CURRENT FUNCTION KEY DEFINITION STRING

USAGE: **FKEYGET** Keynumber, #String

Procedure FKEYGET sets a string to the currently-defined function key substitution string. **Keynumber** is the desired function key number, from 1 to 8. **#String** is the address of a buffer at least 32 characters long to receive the desired string.

EXAMPLE 1:

```
INCLUDE LIBRARY

BYTE KEYDEF[32]
WORD I
...
BEGIN
...
PUT NL,"The current function key definitions are:"
FOR I=1 TO 8
  FKEYGET I,KEYDEF
  OUTPUT "#C#I = #S",I,KEYDEF
...
```

NOTE:
1. To define function key strings, see FKEYSET.
2. For programs created with the optional GENMASTER utility of the optional Developer's system, function key definitions are initially null strings until defined by calls to FKEYSET.

PROC FKEYSET DEFINE A FUNCTION KEY EXPANSION STRING

USAGE: **FKEYSET** Keynumber, String

Procedure FKEYSET is used to define a function key substitution string of up to 31 characters. **Keynumber** is the desired function key number, 1 to 8. **String** is the desired function key substitution string.

Once the function key substitution string is defined, pressing the function key in the PROMAL EXECUTIVE, or during data entry to a GETL, EDLINE, or INLINE call, will cause the defined string to replace the current line. Up to 31 characters may be defined. Only normal, displayable characters ($20 through $7E) should be used.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
FKEYSET 2,"COMPILE 2:MYPROG"
```

defines function key F2 to be "COMPILE 2:MYPROG".

NOTE:
1. On the Apple II, function keys are activated by holding down either Apple key and pressing a number key 1 through 8.
2. You can ignore function keys in EDLINE/INLINE/GETL by using FKEYSET to set the key definition to a null string (e.g., FKEYSET 1,"").
3. You can cause the function keys to return their original key code in EDLINE/INLINE/GETL by using FKEYSET to define a string consisting of the key code (see **Appendix B**) followed by a zero byte (e.g., FKEYSET 1,"\85" for the Commodore 64 F1 key).
4. Function key settings defined in a program remain in effect when control is returned to the EXECUTIVE.

FUNC GETARGS SPLIT A COMMAND LINE INTO ARGUMENTS

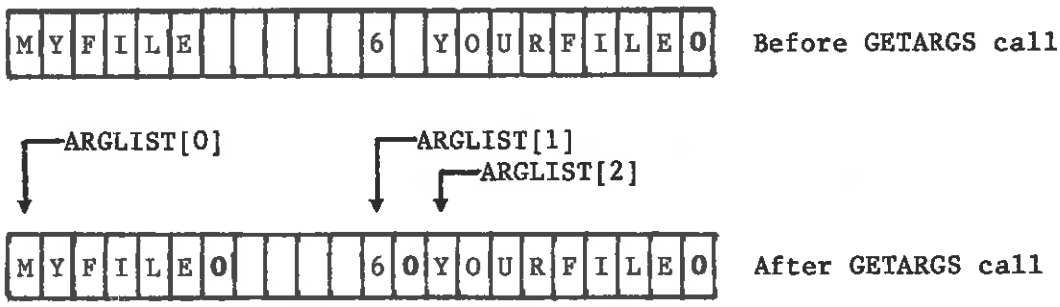
USAGE: Bytevar = **GETARGS**(Argline, #Ptrlist [,Limit [,Sep]])

Function GETARGS is used to parse a line into a list of strings, one string for each argument. **Argline** is the string argument which is to be separated into arguments. **#Ptrlist** is the address of an ARRAY of WORD variables. The function will return a list of pointers to the arguments in this array. **Limit** is an optional argument specifying the maximum number of arguments which may be returned. **Sep** is an optional BYTE argument defaulting to ' ', which specifies what character is to be considered the separator of arguments. The GETARGS function will modify the **Argline** string in place, by installing a 0 byte at the end of each argument, replacing the first separator (usually a blank) after each argument. Each entry in the pointer list will be filled with a pointer to the first **non-blank** character of the argument. The returned value of the function is a BYTE variable indicating the number of arguments returned in the pointer list.

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE ARGLINE[81]
WORD ARGLIST[8]
...
BEGIN
...
MOVSTR "MYFILE     6 YOURFILE", ARGLINE
...
N = GETARGS(ARGLINE,ARGLIST)
```

will return N=3, and set ARGLIST[0]="MYFILE", ARGLIST[1]="6", and ARGLIST[2]="YOURFILE". In an actual application, ARGLINE would typically be read from the keyboard instead. The effects of GETARGS on the ARGLINE array in memory are illustrated below (0 indicates $00 terminator):



NOTE:
1. Since the Argline string is modified in place, if you alter any of the strings returned in ARGLIST, you should be careful not to make them larger or they will affect the content of neighboring strings.

FUNC GETBLKF READ A BLOCK FROM A FILE INTO MEMORY

USAGE: Wordvar = **GETBLKF**(Handle, #Start, Maxsize)

Function GETBLKF does a block read from a file or device. **Handle** is the file handle of the previously opened file (see OPEN for more information on file handles). **#Start** is the desired address where the data should be installed in memory. **Maxsize** is the maximum number of bytes to read. **Wordvar** is returned as the number of bytes actually read. If **Wordvar** is less than **Maxsize**, then end-of-file was encountered before **Maxsize** bytes could be read. GETBLKF does not recognize any record boundaries or delimiters except end-of-file. It is the complementary function to PUTBLKF.
It is the fastest way to read data from disk.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD SCREENFILE     ;File handle
WORD READSIZE       ;# bytes actually read
...
BEGIN
...
SCREENFILE=OPEN("SCREENDATA.D")  ;Open for reading
IF SCREENFILE=0     ;Open error?
  ABORT "#CCant read SCREENDATA.D file!"
READSIZE=GETBLKF (SCREENFILE,$0400,1000)
...
```

reads 1000 bytes (decimal) from the file "SCREENDATA.D" into memory starting at location $0400.

NOTE:
1. The predefined (in the LIBRARY) variable DIOERROR can be checked after a GETBLKF operation to test for possible errors, if desired. If DIOERROR=0, the read was completed normally. If DIOERROR = 2, a disk read error occurred (in which case GETBLKF will return as much as could be successfully read before the error).
2. When using GETBLKF to read data into memory starting at a **particular** element of an array, be sure to specify the **#** operator to indicate that you want the address of the array element, not the value. For example,

```
READSZ = GETBLKF(HANDLE, #BUF[I,0])
```

3. **IMPORTANT: For Commodore 64**, GETBLKF is the only Library routine which uses DYNODISK, if it is enabled. You must **not** mix GETBLKF calls with other, non-DYNODISK read calls (such as GETLF or GETCF) on the same file while DYNODISK is enabled. Also, do not mix GETBLKF calls with DYNODISK off and DYNODOISK on in the same file. To disable DYNODISK from within a program, set C64DYNO to 0 (defined in file PROSYS.S).

FUNC GETC RETURN ONE CHARACTER FROM KEYBOARD

USAGE: Bytevar = **GETC**[(#Variable)]

GETC is a function (not a procedure!) which gets one character from the keyboard and displays it on the screen. It has one optional argument which is the **address** of a variable to receive the character entered. It returns an argument of type BYTE, which is the character read. The same character will be installed in the variable whose address is the argument, if present. The optional argument allows a convenient way to save the character and test it in the same statement. GETC blinks the cursor while waiting for a key to be pressed, and echoes the key to the screen.

CAUTION: If you use the optional second argument, be sure to specify the # operator in front of the variable to receive the character. Otherwise, the character will be installed somewhere in the first page of memory, corresponding to whatever value happens to be in that variable at the time, possibly corrupting the PROMAL system.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE NAME[41]
WORD I
...
BEGIN
...
I=0
WHILE ALPHA(GETC(#NAME[I]))
   I=I+1
NAME[I]=0
```

This fills the buffer NAME with characters from the keyboard until a non-alphabetic character is entered, and terminates it with a $00 byte to make it a string. Alternatively, the form without an argument could be used:

```
I=0
REPEAT
  BUF[I]=GETC
  I=I+1
UNTIL NOT ALPHA(BUF[I-1])
BUF[I]=0
```

NOTE:
1. GETC processes the Alpha lock key (CTRL-A) internally.
2. GETC treats CTRL-Z as end-of-file from the keyboard and therefore returns $00 instead of $1A for CTRL-Z.
3. If you wish to get a key without keyboard echo, see GETKEY.
4. If you wish to test if a key is pressed without waiting for one, see TESTKEY.
5. It is possible to change the cursor blink rate. See **Appendix G.**

FUNC GETCF GET A BYTE FROM A FILE OR DEVICE

USAGE: Flagbyte = **GETCF**(Handle, #Variable)

GETCF is similar to GETC but accepts input from a file or device. The first argument is a WORD variable which is the file **Handle** (see OPEN for information on file handles). The second argument specifies the **address** of the variable to receive the character. GETCF returns FALSE if end-of-file is encountered and TRUE otherwise. **Be sure to remember to specify the # operator on the second argument.**

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE CHAR
WORD INFILE
WORD COUNT
...
BEGIN
...
COUNT=0
INFILE=OPEN(CARG[1])
WHILE GETCF(INFILE,#CHAR)
  COUNT=COUNT + (CHAR=',') ; bump count if is ','
OUTPUT "#C#S contains #W commas.",CARG[1],COUNT
```

This will read the file specified on the command line and display a count of all commas in the file.

NOTE:
1. GETCF is not limited to reading text files. It will correctly return all 256 possible values which can be read from a file, including $00.
2. If the handle is STDIN (the keyboard), then characters are processed as described for GETC above, and GETCF returns TRUE when CTRL-Z is entered.
3. On the Apple after GETCF returns, DIOERR will be 0 normally and 2 if a disk read error occurred, if you wish to check it.

FUNC GETKEY RETURN ONE CHARACTER FROM KEYBOARD WITHOUT ECHO

USAGE: Bytevar = **GETKEY**[(#Variable)]

GETKEY is a function (not a procedure!) which gets one character from the keyboard without displaying it on the screen. It has one optional argument which is the **address** of a variable to receive the value input. It returns an argument of type BYTE, which is the character read. The same character will be installed in the variable whose address is the argument, if present. The optional argument allows a convenient way to save the character and test it in the same statement. **Appendix B** gives the key codes returned by GETKEY.

CAUTION: If you use the optional second argument, be sure to specify the # operator in front of the variable to receive the character. Otherwise, the character will be installed somewhere in the first 256 bytes of memory, corresponding to whatever value happens to be in that variable at the time, possibly corrupting PROMNAL or the operating system.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
PUT NL,"Press any key when ready, or * to exit."
IF GETKEY = '*'
  ABORT "#cProgram terminated."
...
```

NOTE:
1. GETKEY processes Alpha-lock (CTRL-A) internally.
2. GETKEY returns CTRL-Z as $1A, without special treatment.
3. You may alter the cursor blink rate. See **Appendix G.**

PROC GETL GET LINE OF TEXT FROM KEYBOARD

USAGE: **GETL** #Buffer [,Limit]

Procedure GETL inputs a line from the keyboard, allowing all editing (backspace, etc.) supported by the PROMAL EXECUTIVE prior to the carriage return. GETL has one required argument which is the address of the buffer to receive the line. A second optional argument can be used to specify the maximum number of characters to be read. The default limit is 80 characters. The line will be returned as a string, with a $00 byte replacing the carriage return at the end of line. The carriage return is not returned. Therefore the buffer for the default GETL should be 81 bytes long to allow for the full input.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
BYTE LINE[81]  ;Input line buffer
...
BEGIN
GETL LINE
```

This inputs a line from the keyboard into the LINE buffer.

EXAMPLE 2:

```
BYTE PAGE [41,25]  ; Array of 25 lines of up to 40 chars each
WORD I
...
GETL #PAGE[0,I], 40
...
```

This reads a line form the keyboard into the Ith line of the PAGE array, up to 40 characters long.

NOTE:
1. **Table** 1 in the PROMAL LANGUAGE MANUAL lists the line editing keys.
2. Due to buffer size limits, the maximum line size allowable for the Commodore 64 is 80 characters, and 127 characters for the Apple II.
3. It is possible to alter the cursor blink rate and the editing keys used by GETL. See **Appenix G.**
4. GETL always clears a space on the screen large enough to enter Limit characters by outputting blanks from the present cursor position, before accepting input (at the original cursor position). This may cause the screen to scroll if the initial cursor position was within Limit characters of the end of the screen.

FUNC GETLF GET LINE OF TEXT FROM FILE OR DEVICE

USAGE: Flagbyte = **GETLF**(Handle, #Buffer [, Limit])

Function GETLF (not a procedure!) inputs a line from a file or device specified by the file **Handle**, which is the first argument. See OPEN for more information on file handles. The second argument is the address of the buffer to receive the line. An optional third argument can be used to specify the maximum number of characters to be returned. If the line contains more than **Limit** characters, the entire line is read up to and including the carriage return, but only the first Limit characters are copied into the buffer. The line will be terminated by a $00 byte and will not include the carriage return. The returned value of the function is TRUE normally and FALSE (0) if end-of-file was encountered before any bytes could be read from the file or device.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD INPUTFILE
BYTE LINE[41] ; Input buffer
...
BEGIN
...
INPUTFILE=OPEN("MYFILE.T")
IF INPUTFILE=0 ; open error?
  ABORT "#CCan't open MYFILE.T - Program Aborted"
WHILE GETLF(INPUTFILE,LINE,40) ;only 40 chars max please
  PUT NL,LINE
...
```

This will display the first 40 characters of every line of file MYFILE.T.

NOTE:
1. Due to buffer size limitations, a maximum of 127 characters can be read for a line. On the Commodore 64, if the Handle is STDIN (the keyboard), this is reduced to a maximum of 80 characters. To read lines larger than 127 characters from a file, you could use GETCF instead, installing characters in your own buffer one at a time, checking for a carriage return.
2. If the Handle is STDIN (the keyboard), the alpha-lock character (CTRL-A) will be processed internally, and CTRL-Z will be treated as end-of-file if it is the first character of the line.
3. When using GETLF to input starting at a particular element of an array, be sure to specify the # operator to indicate the address of the element. Like all PROMAL routines processing strings, the GETLF procedure expects the **address** of the desired destination for the string.

FUNC GETPOSF NOT AVAILABLE ON COMMODORE 64 RETURN PRESENT FILE POSITION

USAGE: Wordvar = **GETPOSF**(Handle [,#Segvar])

Function GETPOSF returns the relative position of the next byte to be read/written in a file. **Handle** is the file handle of a previously OPENed file. **Wordvar** is returned as the relative offset from the beginning of the file in bytes, from 0 to 65535. **#Segvar** is an optional address of a word variable to receive the high order 16 bits of the relative offset. It is necessary to specify #Segvar only if the file is more than 64K bytes long and you wish to know the full offset into the file. GETPOSF should not be used for devices.

A common use of GETPOSF is to save the current file position for a file which has been partially read but must be closed temporarily for some reason (such as changing disks during a single-drive copy operation), and then restoring the file to the same position so that you can continue reading.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD CURPOSN
WORD FILE
...
FILE=OPEN("MYFILE.D",R)
...
CURPOSN=GETPOSF(FILE)
CLOSE FILE
...
FILE=OPEN("MYFILE.D",R)
SETPOSF FILE,CURPOSN
...
```

NOTE:
1. This function is not supported on the Commodore 64 because the Commodore hardware and ROMs to not support it.

FUNC GETTST GET T DEVICE STATUS

USAGE: Bytevar = **GETTST** (IOflag)

Function **GETTST** tests if the T device (serial port) is ready to send or receive a character. **IOFlag** is 0 to test the input status and 1 to test the output status. The function returns TRUE if the serial port is ready and FALSE if not. When testing the input status, **GETTST** returns TRUE if a character has been received. When testing the output status, the function returns TRUE if the transmitter is empty (the last character, if any, has been sent).

Appendix F contains additional information on GETTST and related topics.

EXAMPLE 1:

```
INCLUDE PROSYS  ;Where GETTST is defined
...
WORD COM        ; File handle for serial port
BYTE CHAR       ; Received character
...
COM = OPEN("T", 'B') ; Open serial port for input/output
IF COM=0
  ABORT "#CUnable to open serial port"
TDEVRAW=$80     ;Enable "raw" serial input mode (see Appendix F)
...
REPEAT
  IF GETTST(0)   ;Character received from serial port?
    CHAR=GETCF(COM) ;Get it
    PUT CHAR      ;Display it
UNTIL TESTKEY     ;Do it until any key is pressed
CLOSE COM         ;Close the serial port
...
```

NOTE:
1. You will need to have INCLUDE PROSYS near the start of your program in order to use GETTST.

FUNC GETVER OBTAIN PROMAL VERSION CODE

USAGE: Wordvar = **GETVER**

Function GETVER returns a WORD value indicating the version of PROMAL which is running. There are no arguments. The low byte of the returned code is the version number as two hex digits (for example, $21 for version 2.1). The high order byte indicates the target machine for the PROMAL runtime package, as follows: $01 = Commodore 64, $02 = Apple II, $03 = IBM PC small memory model, $04 = IBM PC large code memory model. Additional codes may be defined as PROMAL becomes available on other target machines.

EXAMPLE 1:

```
INCLUDE LIBRARY
INCLUDE PROSYS       ; Where GETVER is defined
...
IF GETVER:> <> $02   ; Make sure we're on an Apple
  ABORT "#cThis program runs only on Apple II"
...
```

NOTE:
1. You will need to INCLUDE PROSYS near the start of your program in order to use GETVER.

FUNC INLINE INPUT LINE OF TEXT FROM SCREEN

USAGE: Bytevar = **INLINE**(String [,Limit [,Mode]])

Function INLINE is the same as EDLINE, except that the **String** to be edited in place is automatically set to null at the start of the routine. The **String** argument should be the address of a buffer **large enough to hold Limit+1 characters.** Please see EDLINE for a full description.

FUNC INLIST SEARCH LINKED LIST

USAGE: Wordvar = **INLIST** (String, Listend [,Fold [,Limit [,Safety]]])

Function INLIST is a special purpose routine for advanced programmers. It searches a linked list of a specific form for an entry matching a string. If the string is not found, 0 is returned. Otherwise, the address of the matching string is returned. **String** is the string desired. **Listend** is a pointer to the end of the list, as shown below (i.e., the link to the first name to try is the word at Listend-2. The optional argument **Fold** is a flag, defaulting to FALSE, which if set to TRUE indicates that lower case alphabetic characters should be considered as matching their uppercase equivalents. **Limit** is an optional argument defaulting to 255, indicating the maximum number of characters required to match in **String.** **Safety** is an optional argument defaulting to 8192 ($2000) indicating the maximum number of entries to test before giving up. **Safety** prevents the function from "hanging up" forever if the linked list is corrupted. The assumed format of the linked list is as follows:


```
                                                    <---LISTEND
    ---------------------------------------------
    Address of last entry string (2 bytes)
    ---------------------------------------------
    String for last entry (N bytes)
    ---------------------------------------------
    Address of next-to-last entry string (2 bytes)
    ---------------------------------------------
        ...//...
    ---------------------------------------------
    Address of first entry string (2 bytes)
    ---------------------------------------------
    First Entry string (M bytes)
    ---------------------------------------------
    $0000 (2 bytes, beginning-of-list sentinel)
    ---------------------------------------------
```

EXAMPLE 1:

```
    ;   This example shows how to build a simple Symbol Table for a
    ;   compiler, assembler, etc. using a linked list, where each entry
    ;   is a variable-length name and its associated definition (value).

    INCLUDE PROSYS      ; where INLIST is defined

    WORD SYMTAB [1000] ;space for linked list
    WORD LISTEND       ;ptr to next unused entry

    PROC PUTST ;NAME, VALUE
      ; Install NAME, VALUE into symbol table linked list.
    ARG WORD NAME  ; string to install
    ARG WORD VALUE ; associated definition of name
    WORD STPTR
    BEGIN
    MOVSTR NAME,LISTEND ; install name
    STPTR=LENSTR(NAME)+LISTEND+1 ; after name string
    M[STPTR]=VALUE:<    ; install low byte of value
    M[STPTR+1]=VALUE:>  ; ...hi byte
    M[STPTR+2]=LISTEND:<
    M[STPTR+3]=LISTEND:>
    LISTEND=STPTR+4     ; next available location
    END

    FUNC WORD GETST ; NAME
        ; Returns value stored in symbol table for NAME
    ARG WORD NAME   ; name to look up in symbol table
    WORD ENTRY
    BEGIN
    ENTRY=INLIST(NAME,LISTEND)  ; search list for name
    RETURN (ENTRY+LENSTR(ENTRY)+1)@+  ; = value of NAME
    END
    ...
    ;   Initialize start of list for symbol table...
    SYMTAB[0]=0      ; end of list sentinel
    LISTEND=SYMTAB+2 ; starting address
    ...
```


NOTE:
1. A "real" symbol table manager would need to check for errors such as no more room in the buffer, symbol not found, etc.
2. You will need to have INCLUDE PROSYS near the front of your program to use INLIST.

FUNC INSET TEST IF A CHARACTER IS IN A STRING OR SET

USAGE: Bytevar = **INSET**(Char, String [,Meta])

Function INSET returns the position of a specified character in a string. **Char** is the desired character, **String** is the string to search. **Meta** is an optional argument character, which is usually '-' if specified. If **Meta** is specified, then the **Meta** character can be used to denote a range of characters. **Bytevar** is returned as 0 if the character is not found in the string, or as the index to the matching character **plus one** if the character is found in the string.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE CHAR
BYTE I
...
BEGIN
...
CHAR='A'
...
I = INSET(CHAR, "ABC")
```

This returns I=1 because the A matches first character of the string.
When **Meta** is not specified, INSET is often used to find a particular delimiter in a string:

EXAMPLE 2:

```
WORD LINE
...
LINE="100, SPRING INVENTORY"
...
PUT LINE+INSET(',',LINE)
```

This will display:

```
SPRING INVENTORY
```

because the INSET function returns the number of characters to skip over to get beyond the comma. A different use for function INSET is to test for membership of a byte in a set of bytes:

EXAMPLE 3:

```
BYTE LINE[80]
...
IF INSET(LINE[I], "A-Za-z0-9.",'-')
```

tests to see if the character LINE[I] is alphabetic, numeric, or a period. The **Meta** argument is specified as '-', so "A-Z" will be matched by any character between A and Z inclusive. If LINE[I] was any character between 'B' and 'Y' inclusive, INSET would return 2 (the position of the '-' plus one).

PROC INTSTR CONVERT SIGNED INTEGER VALUE TO STRING

USAGE: **INTSTR** Value, #Var [,Radix [,Minfield [,Padding]]]

Procedure INTSTR takes a signed value and generates the ASCII string representing the value. **Value** is the desired value to encode and **#Var** is the address of the buffer to receive the ASCII characters. **Radix** is the optional base to be used, defaulting to 10. **Minfield** is the minimum field width to generate, defaulting to 0. **Padding** is an optional character (not string!) argument which is the padding character desired to fill out the buffer to the minimum field width, defaulting to blank.

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE BUF[8]
INT MYNUM
...
BEGIN
...
MYNUM=568-11
INTSTR MYNUM, BUF
...
PUT NL,BUF
```

This will display:

```
557
```

EXAMPLE 2:

```
INTSTR $FFFE, BUF, 10, 4
```

will install the string " -2" into BUF.

NOTE:
1. If a minimum field width is specified, the number will always be right-justified in the field. If more characters are required to output the number than are specified for the minimum field width, they will be encoded without any error indication.
2. To convert an unsigned (BYTE or WORD) variable, use procedure WORDSTR instead. To convert a REAL value, use procedure REALSTR instead.

PROC JSR CALL MACHINE LANGUAGE SUBROUTINE

USAGE: **JSR** [Address [,Areg [,Xreg [,Yreg [,Flags]]]]]

Procedure JSR calls a machine language subroutine at a specified address, optionally loading the 6502 (or 6510 or 65C02) processor's hardware registers with specified values before the call. **Address** is the address of the desired routine. **Areg**, **Xreg**, **Yreg**, and **Flags** are optional arguments which specify the desired values to be installed in the A, X, Y, and flags (processor status word) registers, respectively. All register arguments should be type BYTE. Naturally the address must be type WORD. It is possible to sample the values returned in the registers from the machine language program.

Please see **Appendix I** for a detailed explanation and examples of JSR.

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program in order to use JSR.

FUNC LENSTR RETURN LENGTH OF STRING

USAGE: Bytevar = **LENSTR** (String)

LENSTR is a function which returns a BYTE result indicating the length of the **String** which is the argument.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE NAME[20]
BYTE SIZE
...
BEGIN
...
MOVSTR "Hello", NAME
...
SIZE=LENSTR(NAME)
```

This sets SIZE=5. The size does not include the $00 byte terminator.

NOTE:
1. You may find frequent need of a statement similar to:

```
IF LENSTR(LINE) > 0
  ...
```

where LINE is an array of bytes holding some string. This can be more economically written as:


```
IF LINE@<
  ...
```

which is equivalent, since if a string is non-null, the first character can't be the string terminator.

PROC LOAD LOAD, UNLOAD, OR EXECUTE PROGRAM OR OVERLAY

USAGE: **LOAD** Progname [,Bitflags]

The LOAD procedure loads, unloads, and executes programs and overlays on the Apple II and Commodore 64. **Progname** is the desired program or file name. **Bitflags** is an optional BYTE argument consisting of several 1-bit flags used to control the action taken by the LOADer. Please see the Chapter 8 of the PROMAL LANGUAGE MANUAL for details and examples.

NOTE:
1. You will need to INCLUDE PROSYS near the start of your program in order to use LOAD.

FUNC LOOKSTR SEARCH A LIST OF STRINGS

USAGE: Intvar = **LOOKSTR** (String, Plist [,Nstr [,Fold [, Limit]]])

Function LOOKSTR searches an array of strings, trying to match a given string. **String** is the desired string to try to match, **Plist** is the starting address of a list of pointers to strings, terminated by a $0000 word. **Nstr** is an optional argument specifying the maximum number of strings to search. **Fold** is an optional argument, which if set TRUE, will cause lower case alphabetic characters to be considered equal to their upper case equivalents. **Limit** is an optional argument specifying the maximum number of characters to compare within each string. **Intvar** is returned as -1 if the string did not match, or as the array index to the string that did match.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
INT I
BYTE COMD [20]
DATA WORD KEYWORDS []="MOVE","DRAW","ERASE","DASH","REDRAW","EXIT",0
...
BEGIN
...
MOVSTR "ERASE",COMD
...
I=LOOKSTR(COMD,KEYWORDS)
```

This will return I=2, because the string in COMD matches the third entry in the list.

FUNC MAX RETURN THE LARGEST OF TWO OR MORE VALUES

USAGE: Wordvar = **MAX** (Val1,Val2[,...])

Function MAX returns the largest of two or more arguments **of type WORD (unsigned).** It is normally used to find the larger of two or more addresses. Do not use it with type REAL arguments.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD I
WORD J
WORD K
...
BEGIN
...
J=1000
K=$D000
...
I=MAX(J,K,$C000)
```

This will return I=$D000.

NOTE:
1. Each value to be tested must be explicitly included in the function call. You cannot find the largest value in an array by merely calling MAX with the array name as an argument. The following example shows how a loop can perform this function.

EXAMPLE 2:

```
WORD LARGEST
WORD MYARRAY[100]
WORD I
...
BEGIN
...
LARGEST = 0      ; Dummy to initialize
FOR I = 0 TO 99 ; Find largest value in array
  LARGEST = MAX (MYARRAY[I], LARGEST)
...
```

FUNC MIN RETURN THE SMALLEST OF TWO OR MORE VALUES

USAGE: Wordvar = **MIN** (Val1,Val2[,...])

Function MIN returns the smallest of two or more arguments **of type WORD (unsigned)**. Do not use it with type REAL arguments. It is normally used to find the lesser of two or more addresses.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD I
DATA WORD BOUND []= 100,200,300,400,500,600
...
BEGIN
...
I=351
...
I=MIN(I,BOUND[3])
```

This will return I=351, because 351 is smaller than 400.

FUNC MLGET LOAD MACHINE LANGUAGE PROGRAM

USAGE: Wordvar = **MLGET** (Filename [,Loadaddress])

Function MLGET loads a machine language program. On Commodore 64 systems it is expected to be in standard Commodore format for a machine language PRG file. For Apple II systems it is expected to be a standard Apple BSAVE type file. **Filename** is a string containing the desired file name. **Loadaddress** is an optional load address. If not specified or 0, the load address will be the address at which the program was saved. The function returns a word result which will be $0000 if an error occurred, or the address of the last byte loaded if successful.

You may load as many programs as needed by making multiple calls. No checks are made to see if the loaded program conflicts with other memory usage, and the memory allocation pointers (LOFREE, HIFREE, etc.) are not adjusted. It is your responsibility to insure that an appropriate location is used.

Please see **Appendix I** for more information.

EXAMPLE 1:


```
INCLUDE PROSYS
WORD ENDPROG                              ; Last address
DATA WORD MLPROGNAME = "MLROUTINES"  ; File name
BYTE DUMMY
...
REPEAT
  ENDPROG = MLGET(MLPROGNAME)         ; Load Machine Lang. support routines

  IF ENDPROG = 0
    PUT NL,"Cant load file ",MLPROGNAME
    PUT NL,"Please insert Master diskette and close drive door."
    PUT NL,"Press any key when ready."
    DUMMY = GETC
UNTIL ENDPROG <> 0
...
```

NOTE:
1. For Commodore 64, file names for MLGET must match the desired name **exactly**, including upper and lower case and character set selection.
2. For Apple II, remember that many Apple programs load at $2000 before relocating themselves to their final destination. In this case you may need a BUFFERS HIRES command before loading to help protect PROMAL from being overwritten.
3. You will need to INCLUDE PROSYS near the beginning of your program in order to user MLGET.
4. For Apple II, the default load address is found in the AUX_TYPE field of the directory entry. See the ProDOS Technical Reference Manual for details.

PROC MOVSTR COPY OR JOIN STRINGS OR EXTRACT SUBSTRING

USAGE: **MOVSTR** FromString, ToString [,Limit]

MOVSTR is a procedure which is used to copy strings, to concatenate strings, or to extract substrings (i.e., replaces the LEFT$, MID$, and RIGHT$ functions found in BASIC). **FromString** is the address of the string to copy. **ToString** is the address of the destination. **Limit** is an optional argument specifying the maximum number of characters to copy.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE LINE[81]
BYTE SAVELINE[81]
BYTE KEYWORD[5]
...
BEGIN
...
MOVSTR LINE, SAVELINE
```

This copies the string LINE to the buffer SAVELINE.

EXAMPLE 2:

```
    MOVSTR " today.", LINE+LENSTR(LINE)
```

This concatenates the string literal " today." to the end of the string LINE.

EXAMPLE 3:

```
    MOVSTR LINE, KEYWORD, 4
```

This extracts the first 4 characters of the string LINE and installs them in the string KEYWORD. The Limit argument does not include the 0 byte string terminator. The destination string may overlap the source string without problems.

EXAMPLE 4:

```
    MOVSTR LINE,LINE+1
    LINE[0]='A'
```

This inserts the character 'A' at the beginning of the string LINE.

NOTE:
1. MOVSTR always installs a 0 byte terminator at the end of the copied string. Therefore you should always allow room for it.
2. When specifying a particular element of an array for the source or destination , be sure to include the # operator to indicate the address of the element instead of the value (e.g., #BUF[I] is correct).

FUNC NUMERIC TEST IF A CHARACTER IS A DIGIT

USAGE: Bytevar = **NUMERIC** (Char)

Function NUMERIC returns TRUE if the argument is an ASCII numeric digit and FALSE otherwise. The argument is expected to be type BYTE (not a string!).

EXAMPLE 1:

```
    INCLUDE LIBRARY
    ...
    WORD VAL
    BYTE CHAR
    BEGIN
    ...
    VAL=0
    WHILE NUMERIC(GETC(#CHAR))
      VAL=10*VAL+(CHAR-'0')
    ...
```

This accepts a series of keystrokes until a non-digit is entered, and sets VAL to the numeric decimal value entered.

FUNC ONLINE **AVAILABLE ONLY ON APPLE II** GET VOLUME NAME OF DISK

USAGE: Bytevar = **ONLINE** (Slot, Drive, #Buf)
or
Bytevar = **ONLINE** (0, Unit, #Buf)

Function ONLINE tests if an Apple disk drive (including /RAM disk) is ready, and if so, installs the ProDOS volume name in a specified buffer. In the first form, **Slot** is the Apple slot number (1 to 7), **Drive** is the drive number (1 or 2), and **#Buf** is the address of a buffer of at least 18 bytes which will receive the volume name. The function returns TRUE if the drive is ready and FALSE otherwise (in which case IOERROR holds a code indicating the reason as described in the OPEN function, which will normally be 2 for illegal unit, 3 for not ready, or $28 for non-exisitent). If the first argument is 0, then the second argument is interpreted as a ProDOS unit number (sometimes called a device ID), which is a byte with the following format: Bit 7 is the drive number bit (0=drive 1, 1= drive 2); and bits 4-6 are the slot number (0-7); bits 0-3 are 0. The volume name is returned in the specified buffer as a PROMAL string. The name will have a leading and trailing '/', for example "/USER.DISK/".

EXAMPLE 1:

```
INCLUDE LIBRARY
INCLUDE PROSYS
...
BYTE VOLNAM [18]  ; Buffer for diskette volume name
WORD HANDLE
...
IF ONLINE(6,2,BUF) ; have second floppy disk?
  FILE = OPEN ("2:SCRATCH.T",'W')  ;Open file on drive 2
ELSE
  FILE = OPEN ("1:SCRATCH.T",'W')
IF FILE=0
  ABORT "#cCan't open SCRATCH.T for writing"
...
```

NOTE:
1. The /RAM device is normally configured for slot 3 drive 2 and may be tested in with ONLINE.
2. You will need to INCLUDE PROSYS near the front of your program to use ONLINE.

FUNC OPEN OPEN FILE OR DEVICE

USAGE: Wordvariable = **OPEN** (Filename [,Mode [,Nocheck [,Type]]])

OPEN is a function (not a procedure!) which opens a specified file or device for input or output. The first argument is a string which is the desired file or device name. The second argument is optional and is a character (not a string!) specifying the desired access **Mode**, chosen from the following:

'R'	Read access
'W'	Write access
'A'	Append (write, beginning at end of file) access
'B'	Both read and write (**Not available on Commodore 64** except as noted below for use with the command channel or T device).

The default access mode is 'R'. The remaining optional arguments **Nocheck** and **Type** are normally omitted, and are used for opening special system-dependent file types. These system-dependent options are discussed below.

The function OPEN returns a non-zero **file handle** of type WORD if the open was successful, and 0 if it was not. This file handle (also sometimes called a file descriptor) should be saved in a WORD variable. After opening the file, you can refer to the file for I/O operations by simply using this handle. The handle is required as the first argument for other library routines which operate on files.

If OPEN returns 0, indicating that the file could not be opened, then the pre-defined variable IOERROR indicates the reason, as follows:

IOERROR	Meaning (If function OPEN returns 0)
0	No error. Normal.
1	Illegal access mode character.
2	Illegal file or device name.
3	Device not ready (or volume not found on Apple II).
4	File not found (for R mode access).
5	File already exists (for W mode access).
6	Can't open another file (e.g., no more disk buffers) .
7	Write protected (for A or W access).
8 or more	Other (system dependent, see your computer manual).

You should always test for an unsuccessful open.

EXAMPLE 1:


```
INCLUDE LIBRARY
...
WORD INPUTFILE ;input file handle
BYTE LINE[81]  ;input line buffer
BEGIN
INPUTFILE=OPEN("MYDATA.D")
IF INPUTFILE=0
   ABORT "Can't open input file!"
WHILE GETLF (INPUTFILE,LINE)
...
```

The example above could be expanded for better error processing as follows:

EXAMPLE 2:

```
INPUTFILE=OPEN("MYDATA.D",'R')
IF INPUTFILE=0    ; open error?
   CHOOSE IOERROR
   3
     PUT NL,"Disk not ready."
   4
     PUT NL,"MYDATA.D file not found."
   ELSE
     OUTPUT "#CDisk error #I",IOERROR
   ABORT "#CProgram aborted."
...
```

NOTE:
1. The **Commodore 64** firmware limits the maximum number of open files to three, of which only one may be open for writing or appending. If using relative files (see **Appendix M**), at most one sequential file may be open, and DYNODISK should be off. DYNODISK uses up one buffer inside the 1541 drive.
2. For the **Apple II**, the maximum number of open files is governed by the number of available buffers (see the BUFFERS command in the PROMAL USER'S GUIDE).
3. For the Apple and Commodore, the file handle points to a data structure maintained by PROMAL. The first part of this data structure is the file name, as a PROMAL string. Therefore, if you wish to display the name of a successfully opened file, you can simply output the file handle, for example:

```
PUT NL,"Now reading file ", INPUTFILE
```

4. Devices such as the printer, modem, workspace, etc. are opened in the same manner as files. For example:

EXAMPLE 3:

```
WORD PRTR
...
PRTR = OPEN ("P",'W')
IF PRTR = 0
  PUT NL,"Printer is not ready."
ELSE
  PUTF PRTR, NL,"This will be printed.",NL
...
```


This opens the printer and outputs a line to it. Be sure to always send a final NL to the printer; most printers do not actually print until they receive a carriage return.
5. For special considerations for opening and using the T device (modem), see **Appendix F.** The INTERFACING chapter of the PROMAL LANGUAGE MANUAL contains additional information on opening files and devices, including the printer.

Opening Special System-Dependent Files

The optional argument **Nocheck** is a flag, which, if TRUE, allows files to be opened which do not conform to the standard PROMAL naming conventions and file types. When **Nocheck** is TRUE, you can open any file allowed by the underlying operating system, and no default file extension will be added to the name. This allows your PROMAL programs to read BASIC program files, machine language files, word processor files, etc. If you specify **Nocheck** as TRUE, you may also optionally specify the argument **Type**, which is an argument of type BYTE specifying the type of file desired. This argument is system-dependent.

For the Commodore 64, it can be any of the following:

```
'P'  for PRG type files (BASIC and machine language files)
'S'  for SEQ type files (Sequential files)
'U'  for USR type files (User files).
```

The default type is 'S'. For example:

```
C64HANDLE = OPEN("BASIC PROG",'R',TRUE,'P')
```

opens a file named "BASIC PROG" of type PRG for reading.

```
C64HANDLE = OPEN("WordProcData",'W',TRUE,'U')
```

opens a file of type USR for writing.

You can also open to read a directory, open a direct access channel, or the command/error channel. The Type argument should not be specified in this case. For example:

```
C64DIR = OPEN("1:$",'R',TRUE)
```

opens the directory on drive 1. Do not attempt to open a directory for writing. After opening a directory, the contents read will be the sector contents of the directory (minus the track and sector links to the next sector), starting with the BAM. Consult Anatomy of the 1541 Disk, by Abacus Software, for further information on the format of the directory. It is recommended that GETBLKF be used to read the data.

EXAMPLE 4 (COMMODORE 64):

```
WORD C64DA ; Handle for DA file
BYTE CHAN  ; C64 channel # for DA file
...
C64DA = OPEN("#",'B',TRUE)
CHAN = (C64DA+LENSTR(C64DA)+2)@<
```


This opens a direct access file. CHAN is needed so that it can be embedded in the commands to read and write blocks. GETBLKF and PUTBLKF are the best commands to use to read and write the data. Note that the file is opened in 'B' mode, so both reading and writing are permitted. Consult the Commodore 64 Programmer's Reference Guide or the Abacus book for more information.

EXAMPLE 5 (COMMODORE 64):

```
EXT BYTE C64DYNO AT $0DE3 ; From PROSYS.S file
WORD C64CMD  ; File handle for command/error channel
BYTE BUF[81] ; Holds reply from error channel
DATA WORD FMTCMD = "N0:TrashDisk,r8"
...
C64DYNO=0  ;Disable DYNODISK
C64CMD = OPEN("%", 'B',TRUE)
PUTBLKF C64CMD, FMTCMD, LENSTR(FMTCMD)
IF GETLF(C64CMD,BUF)
  PUT NL,BUF,NL
```

This opens the Commodore 64 command/error channel, immediately issues a command to format the disk, and displays the error message from the error channel. Internally, PROMAL will use channel 15 for drive 0: (device 8) and 14 for drive 1: (device 9). Error messages are best read using GETLF. Commands **must** be sent using PUTBLKF (not PUTF or OUTPUTF). Before sending commands to the disk command channel, you should disable DYNODISK, because the commands you send may cause the disk to destroy the special DYNO code in the disk drive. Internally, PROMAL always leaves the command/error channel(s) open all the time; closing and opening an error channel makes the appropriate "connection" through the file handle.

PROMAL assigns Commodore channel 3 to the Printer and 2 to the T device (serial port). The secondary address for the printer can be selected by setting the variable C64PSA before the open (see **Appendix G**). Channel 1 is reserved for the DIR function. Files are assigned channels of 4 and up, with secondary addresses the same as the channel. Therefore if you wish to use a channel for some special purpose in a machine language program, you should choose a channel like 9 or 10 to avoid a possible conflict. Do not close the Commodore command/error channel. Do not attempt any direct serial bus activity from a machine language program with DYNODISK enabled.

Since DYNODISK uses one extra buffer inside the 1541/1571 drive, under some circumstances you may be able to open fewer files with DYNODISK enabled. Under no circumstances should a file be opened with DYNODISK enabled while any other device other than a single disk drive is connected to the serial bus. Failure to observe this precaution will probably result in a "hung" system.

For the **Apple II**, you do not have to specify the **Type** of file in Read mode; any type of file can be opened when **Nocheck** is TRUE. For write mode, the file type should be specified. The values for common ProDOS file types are:

BAD	$01	PCD	$02	PTX	$03	PDA	$04
TXT	$05	BIN	$06	FNT	$07	FOT	$08
BA3	$09	DA3	$0A	WPF	$0B	SOS	$0C
RPD	$10	RPI	$11	DLR	$0F	CMD	$F0
PRML	$F8 (U8)	INT	$FA	IVR	$FB	BAS	$FC
VAR	$FD	REL	$FE	SYS	$FF		

For further information, consult the ProDOS Technical Reference Manual. For example,

```
AIIHANDLE = OPEN("BASPRG",'R',TRUE)
```

opens the file BASPRG for reading. BASPRG could be a Basic program (or any other type of file).

```
AIIHANDLE = OPEN ("LETTER",'W',TRUE,$05)
```

This opens file LETTER of type TXT for write access.

For the Apple II, attempting to open a locked file for 'W' access will be treated as a write-protect error. However, opening a locked file or write protected disk for append mode cannot be detected as an error until an actual attempt is made to write the file. Therefore you should always check DIOERR after the first write operation in append mode. The file PRODOSCALLS.S contains examples which show how to lock or unlock a file.

PROC OUTPUT FORMATTED OUTPUT TO SCREEN

USAGE: OUTPUT Formatstring [, item...]

Procedure OUTPUT displays formatted output on the screen. **Formatstring** is a string which governs how the output will be displayed, and how any optional arguments which follow the format string will be interpreted. The special character **#** is used as a field descriptor inside the format string. Field descriptors indicate what kind of output is desired, chosen from the list below:

#nI	Output signed decimal integer, right justified in a field n characters wide, with leading blank fill. Display "-" after leading blanks if negative (no "+" if plus). If n is omitted, use minimum field width needed to display value.
#nW	Output unsigned decimal word, right justified in a field n characters wide, with leading blank fill. If n omitted, use minimum field width needed to display value.
#nH	Output unsigned hex word, right justified in a field n characters wide, with leading 0 fill. If n is omitted, use minimum field width needed to display value (no leading zeroes).
#nB	Output n blanks (1 if n omitted).
#nS	Output single character or string, left justified in a field of n characters with trailing blank padding.
#nC	Output **one** ASCII character whose value is n decimal. If n is omitted, then output the newline character (ASCII CR, $0D). #0C is not allowed.
#nE	Output scientific notation REAL using a field of n characters (n defaults to 12 if omitted); n must be between 7 and 16.
#n.dR	Output a REAL number using a field n characters wide, with d decimal places shown; n must be between 3 and 12, and d must be less than (n-1).

For each field descriptor in the string there **must** be a corresponding argument following the string (except for #nB and #nC). The value **n** is optional, and defaults to 0 except as noted above. The maximum value for n is 253. Up to a total of 254 characters may be output by the entire procedure call. Hex output will show leading zeros; other numeric output will not. To output the character "#" literally in the format string, use ##.

For #nI, #nW, #nH, and #nS field descriptors, if the value to be output will not fit in the specified field width, extra characters will be output sufficient to display the entire value. For instance, trying to display the value 20000 using a #3W field descriptor will display all five digits, not just 3. However, if fewer digits are needed, blank "padding" will be output to make up the difference. For #n.dR output, remember that you must specify a field width wide enough for the sign and the ".", even if you **know** the answer will be positive (a blank will be displayed). If you try to output a value using #n.dR which is too large to be displayed, PROMAL will first try to display the number using #nE format instead (with the same n as you specified). Failing that, it will print asterisks instead of a value. It is usually a good idea to pick a larger value for n than you really think you will need when using #n.dR format output.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD N
...
BEGIN
N=723
...
OUTPUT "The answer is #W days.", N
```

This will output to the display:

```
The answer is 723 days.
```

EXAMPLE 2:

```
BYTE LINE[81]
LINENO=20
MOVSTR "BEGIN",LINE
...
OUTPUT "#C#4H#5B#S",LINENO,LINE
```

will display (after a carriage return):

```
0014     BEGIN
```

EXAMPLE 3:

```
INCLUDE LIBRARY
...
REAL X
DATA REAL PI = 3.1415926535
...
X = PI * 100000. / 3.
OUTPUT "PI=#10.4R, X=#13E", PI,X
```

will display:

```
PI=    3.1416, X= 1.047198E+06
```

NOTE:

1. The **format string is always required**, and **the number of arguments after the format string must agree with the number of field descriptors given in the format string** (excluding #nB and #nC). You may **not** simply OUTPUT variable names without a format string to display their value!
2. You may output single characters (type BYTE) as well as strings (type WORD) using the #nS field descriptor.
3. The output forms for REAL output will display with rounding based on digits beyond the displayed field. However some decimal fractions such as .005 are not exactly representable in binary format (so, for example, .005 is really .00499999999...). Therefore a value of exactly .005 may be displayed as .00 instead of .01 with a #n.2R field specification.
4. Some useful forms of the #nC field descriptor are:

#C or #13C	Start a new line (carriage return)
#12C	Clear the screen and home the cursor
#15C (Apple) or #18C (Commodore)	Start reverse video
#14C (Apple) or #146C (Commodore)	End reverse video

More information on formatted output is given in the INTERFACING chapter of the PROMAL LANGUAGE MANUAL. The BUDGET.S demo program on the PROMAL disk illustrates how to use formatted output for preparing tabular output data.

PROC OUTPUTF FORMATTED OUTPUT TO FILE OR DEVICE

USAGE: **OUTPUTF** Handle, Formatstring [, item...]

Procedure OUTPUTF operates in the same manner as procedure OUTPUT above, except that the first argument must be a file **Handle** of a previously opened file or device, which is to receive the output.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD I
BEGIN
...
I=100
...
PRTR = OPEN ("P",'W')
...
OUTPUTF PRTR,"#C#10B#I days.#C", I
```

This will output to the printer:

```
        100 days.
```

A carriage return will be written after the line, because of the #C field specified at the end of the **Formatstring**. Note that it is important to remember to send a final CR to the printer, because most printers accumulate characters in a buffer until a carriage return is received. If no final CR is received, the last line will never be printed.

EXAMPLE 2:

```
WORD OUTFILE   ; Output file handle
REAL NETWORTH  ; Total net worth in $
...
OUTFILE=OPEN(CARG[1], 'W') ; Open specified output file
IF OUTFILE=0
  ABORT "#cUnable to open output file #S",CARG[1]
...
OUTPUTF OUTFILE, "#cYour current net worth = $#8.2R", NETWORTH
...
```


NOTE:
1. More information on output to files and devices is given in the INTERFACING chapter of the PROMAL LANGUAGE MANUAL. More examples of output formatting are given for PROC OUTPUT, above.
2. On the Apple II, you may test DIOERR if you wish after writing to a file. DIOERR is set to 0 normally, 1 if the disk is full, and 3 for a disk write error.

PROC PROQUIT EXIT FROM PROMAL SYSTEM

USAGE: **PROQUIT**

Procedure PROQUIT causes an immediate exit from the PROMAL environment. For the Commodore-64, the computer is reset, re-starting BASIC. For the Apple II, the ProDOS "Quit" call is executed as described in Apple's ProDOS Technical Note #7, which will result in a prompt for a new path name and complete prefix for the next system program to be executed. PROMAL does not close any files prior to exiting. However, for the Apple II, PROMAL will restore the /RAM disk using the Apple-presecribed method if it was disabled on startup.

EXAMPLE 1:

```
INCLUDE PROSYS
...
PROQUIT  ; Permanently exit PROMAL
```

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program in order to use PROQUIT.
2. Once you exit PROMAL, it must be re-booted to resume. There is no "warm" entry point.

PROC PUT OUTPUT CHARACTERS OR STRINGS TO THE SCREEN

USAGE: **PUT** item [, item...]

PUT is a procedure for outputting text (including control characters) to the display. PUT may have one or more arguments. Each argument may either be a single character or the address of a string.

EXAMPLE:

```
PUT "Hello, world!",13
```

outputs the string "Hello world!" followed by a carriage return (13 decimal). No carriage return is automatically added before or after the PUT is executed; it must be explicitly indicated. This allows lines of output to be generated using as many separate PUTs as needed.

EXAMPLE 1:


```
INCLUDE LIBRARY
...
WORD PHRASE1
...
PHRASE1= "Abe Lincoln"
PUT cr," The answer was ",PHRASE1, " or "
PUT "Harold Robbins."
```

This will output the sentence:

```
The answer was Abe Lincoln or Harold Robbins.
```

on a new, single line.

NOTE:
1. PUT treats any argument between $00 and $FF inclusive as a single character to be output, and all other values as a pointer to a string of characters to be output. Strings must be terminated by a $00 byte.
2. You may **not** use PUT to display the value of numeric values. Use OUTPUT to perform this function.
3. If you wish to output a **string** starting at a particular element of an array of bytes, don't forget the **#** operator (for example, PUT #PAGE[0,I]). Otherwise, only a single character will be printed (for the reason given in note 1 above).
4. PUT 12 will clear the screen. PUT $12, X, $92 will output X in reverse video on the Commodore 64. PUT $0F, X, $0E will output X in reverse video on the Apple II.
5. PUT N can be used to change text colors on the Commodore 64, where N is as follows: $05=WHT, $1C=RED, $1E=GRN, $1F=BLU, $81=ORG, $90=BLK, $95=BRN, $96=LTRED, $97=GRY1, $98=GRY2, $99=LTGRN, $9A=LTBLU, $9B=GRY3, $9C=PUR, $9E=YEL, $9F=CYN.
6. More information about PUT is contained in the INTERFACING chapter of the PROMAL LANGUAGE MANUAL.

PUTBLKF WRITE MEMORY BLOCK TO FILE OR DEVICE

USAGE: **PUTBLKF** Handle, #Start, Size

Procedure PUTBLKF does a block write to a file or device. **Handle** is the file handle of the previously opened file. **#Start** is the address of the first byte to be written. **Size** is the size of the block to be written, in bytes. The output will be an exact match of the contents of the memory block; no conversions take place, and no terminators or delimiters are added. PUTBLKF and GETBLKF may be used to save and restore memory images, such as arrays or buffers or complete screens.

EXAMPLE:


```
INCLUDE LIBRARY
...
WORD OUTFILE
BYTE BUFFER[300]
...
BEGIN
...
OUTFILE=OPEN(CARG[1],'W')  ;open file name given on command line
IF OUTFILE=0
   ABORT "Can't open file!"
PUTBLKF OUTFILE,BUFFER,300 ;save buffer contents to file
...
```

writes the contents of the BUFFER array to the file specified as the first argument on the command line.

EXAMPLE 2:

```
INCLUDE LIBRARY
...
CON REALSZ = 6          ; # bytes for each REAL variable
REAL ELASTICITY [100]   ; Elasticity matrix for stress analysis
WORD TEMPFILE           ; Temporary file
WORD I                  ; Index to ELASTICITY matrix
...
TEMPFILE=OPEN ("TEMPFILE.MEM", 'w')
...
PUTBLKF TEMPFILE, #ELASTICITY[I], REALSZ*(100-I) ; Save end of matrix
...
```

This saves an exact memory image of the REAL values ELASTICITY[I] through ELASTICITY[99] inclusive to file TEMPFILE.MEM. No conversion to ASCII takes place (i.e., TEMPFILE.MEM is not a text file).

NOTE:
1. If you wish, you may test DIOERR after a PUTBLKF to check for disk errors. DIOERR=0 normally; 1=disk full (works for W device too); 3=disk write error.

PROC PUTF OUTPUT CHARACTERS OR STRINGS TO FILE OR DEVICE

USAGE: **PUTF** Handle, item [,item...]

Procedure PUTF is similar to PUT except the first argument must be a file **Handle** for a previously opened file or device.

EXAMPLE:

```
INCLUDE LIBRARY
...
WORD OUTFILE
DATA WORD FILENAME = "1:MYFILE.D"
BYTE LINEOBUF[20]
BYTE LINE[81]
...
BEGIN
...
OUTFILE=OPEN(FILENAME,'W') ;File name specified in DATA stmt.
...
PUTF OUTFILE,LINENOBUF,' ',LINE,NL
```

This outputs the string LINENOBUF, a blank, the string LINE, and a carriage return to the output file MYFILE.D on drive 1.

NOTES:
1. See PUT above for more information about valid arguments.
2. More information on using PUTF to output to files or devices (including the printer) is given in Chapter 6 of the PROMAL LANGUAGE MANUAL.

FUNC RANDOM RETURN A RANDOM VALUE OF TYPE WORD

USAGE: Wordvar = **RANDOM** [(Seed)]

Function RANDOM returns a pseudo random number **of type WORD**, uniformly distributed between 1 and 65535. If the optional non-zero argument Seed, of type WORD is specified, it will be used as the seed to generate this and any succeeding random numbers.

EXAMPLE 1:

```
INCLUDE LIBRARY
WORD DIEROLL
...
BEGIN
...
DIEROLL = RANDOM % 6 + 1
```

This sets DIEROLL to a random number between 1 and 6 inclusive.

NOTE:

1. RANDOM uses a fast, feedback-shift-register method for generating random numbers, suitable for games, etc. It does not generate random numbers of type REAL.

PROC REALSTR CONVERT REAL VALUE TO STRING

USAGE: **REALSTR** Realval, #Buffer, Fieldwidth [,Decplaces]

Procedure REALSTR is used to convert a REAL numeric value to an ASCII string representing its value. **Realval** is the desired value to convert. **#Buffer** is the address of the string to receive the ASCII numeric representation. **Fieldwidth** is the desired number of characters to represent the number. **Decplaces** is an optional argument specifying the desired number of decimal places to be displayed. If **Decplaces** is omitted, the number will be converted using scientific notation. **Fieldwidth** and **Decplaces** should be expressions of type BYTE or WORD.

Fieldwidth must be specified between 3 and 12 if **Decplaces** is specified (for normal output), or between 7 and 16 if **Decplaces** is not specified (for scientific notation output). If **Decplaces** is specified, it must be less than or equal to the field width minus two. This is because the field width must always include room for a sign and the decimal point itself. If the sign of the value to be printed is +, a blank will be output instead. If the sign of the value is negative, a '-' will be output immediately to the left of the leftmost digit (with any necessary blank padding).

If **Decplaces** is specified, but the value is too large to fit in the specified format, REALSTR will first attempt to convert the number in scientific notation in the specified field width. If it is still too large, the number will not be printed, and the field will be filled with asterisks **(*)**.

EXAMPLE 1:

```
INCLUDE LIBRARY
REAL COST
REAL OVERHEAD
REAL PROFIT
REAL GROSS
BYTE BUFFER[10]
BEGIN
GROSS=1299.95
COST=557.44
OVERHEAD = .18*GROSS
PROFIT=GROSS-COST-OVERHEAD
...
REALSTR PROFIT,BUFFER,7,2
...
PUT NL,"Our profit = $",BUFFER
```

This will display:

```
Our profit = $ 508.52
```


EXAMPLE 2:

```
REAL XVAL
...
XVAL=-0.0000005543
...
REALSTR XVAL,BUFFER,12
```

This will install "-5.54300E-07" in BUFFER.

If the format of the output from REALSTR does not exactly meet your needs, it is usually simple to write a procedure to manipulate the converted output into the format you do want. For example, the following program fragment will pad BUFFER with leading asterisks, such as might be used in a program to write checks:

```
WORD PRINTER ;File handle
REAL AMOUNT
BYTE BUF[10]
WORD I
...
PRINTER=OPEN("P",W)
...
AMOUNT=887.50
...
REALSTR AMOUNT,BUF,9,2
I=0
WHILE BUF[I]=' '
  BUF[I]='*'
  I=I+1
PUTF PRINTER,'$',BUF
...
```

This would print:

```
$***887.50
```

PROC REDIRECT REDIRECT INPUT OR OUTPUT

USAGE: **REDIRECT** #STDIN [,Handle]
- or -
REDIRECT #STDOUT [,Handle]

Procedure REDIRECT is used by advanced programmers to redirect one of the two standard I/O paths available in PROMAL: STDIN (standard input), or STDOUT (standard output). Each of these paths is a global variable of type WORD, defined in LIBRARY, and is initialized to point by default to the keyboard device for input or the screen device for output. **Handle** is a file handle of a previously opened file or device. The REDIRECT procedure sets the standard path to point to the open file or device. If the **Handle** argument is not given, the default redirection is made back to the keyboard or screen. If the handle is specified, it must be open and must have the appropriate mode (direction) for the specified STDxxx (e.g., you can't redirect STDOUT to the keyboard). A

violation of either of these requirements generates a runtime error. Only the two global variables above can be redirected. The EXECUTIVE will automatically redirect STDIN and STDOUT back to the default devices at program termination.

EXAMPLE 1:

```
INCLUDE PROSYS    ; Where REDIRECT is defined
...
WORD OUTFILE
...
OUTFILE=OPEN("SCREENFILE.T",'W')
REDIRECT #STDOUT,OUTFILE
...
PUTF STDOUT,"This will go to SCREENFILE.T",cr
```

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program to use REDIRECT.
2. This function is used by the EXECUTIVE to redirect input and output.

FUNC RENAME RENAME A FILE

USAGE: Bytevar = **RENAME** (Oldfile, Newfile)

Function RENAME is used to change the name of an existing file. **Oldfile** is a string specifying the old file name, as described for OPEN. For the Commodore 64, it may optionally include a drive prefix and file extension. For the Apple II, it may optionally include a drive prefix and pathname. **Newfile** is a second string specifying the desired new name, which must be unique. If the drive or prefix is specified for **Newfile**, it is ignored, and the drive number or prefix for **Oldfile** will be used. It can change the file extension, however. The function returns 0 normally or an error code as described for OPEN.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE RENAMERROR
...
BEGIN
...
RENAMERROR=RENAME("TEMP",CARG[1])
IF RENAMERROR
   PUT NL,"Attempt to rename TEMP.C to ",CARG[1]," failed."
...
```

NOTE:
1. For the Commodore 64, a default file extension will be applied unless NOFNCHK (Defined in file PROSYS.S) bit 7 is 1 (set to $80). If NOFNCHK bit 7 is set, non-SEQ files or files with lower case letters can be renamed. Normally, NOFNCHK is 0, which matches only upper case file names and applies a default extension if none is specified.

2. For the Apple, setting bit 7 of NOFNCHK will allow renaming a file or subdirectory with no file extension.

PROC SETPOSF **NOT AVAILABLE ON COMMODORE 64** SET FILE POSITION

USAGE: **SETPOSF** Handle, Position [,Segment]

Procedure SETPOSF sets the relative position of the next byte to be read/-written in a file. **Handle** is the file handle for a previously opened file. **Position** is a WORD value giving the desired file position. If the desired file position is greater than 65535 (64K), then **Segment** should be specified as the high order 8 bits of the complete 24 bit file position. The first byte of the file is byte 0. If the position specified is greater than the current end-of-file, then the file will be positioned to end-of-file instead, without any error indication. Therefore if you wish to use SETPOSF for implementing a random-access file organization, you should initialize the file when it is created by writing dummy records to the file until it has reached the desired maximum size.

A common use of SETPOSF is to determine a file's size. To do this, open the desired file, then use SETPOSF to set the file to a position known to be larger than end-of-file. Then use GETPOSF to read the true end of file position. An example for function GETPOSF, above, illustrates a second common use of SETPOSF.

EXAMPLE 1:

```
INCLUDE LIBRARY
CON RECSIZE=80
BYTE RECORD[RECSIZE+1] ; Current record contents
...
PROC GETRECORD ; File, RecNum
  ; Read record # RecNum from File into Record
ARG WORD FILE ; Open file handle
ARG WORD RECNUM ; Desired record #
BEGIN
SETPOSF FILE,RECNUM*RECSIZE
IF GETBLKF(FILE,#RECORD,RECSIZE) < RECSIZE
  PUT NL,"***Tried to read beyond end of file on file #S",FILE
  OUTPUT "#C***Record requested = #W",RECNUM
  CLOSE FILE
  ABORT "#CFile closed, program terminated."
END
...
```

The above example shows a routine to read a random record into memory from a database file, given the record number and file handle, for a file of up to 64K bytes.

FUNC SETPREFIX **NOT AVAILABLE ON COMMODORE 64** SET PATHNAME

USAGE: Bytevar = **SETPREFIX**(Dirname)

Function SETPREFIX attempts to set the current pathname to the specified volume or directory name string, **Dirname**. If successful, it returns TRUE. If the specified directory is not on line, FALSE is returned and the current path remains unchanged. The string specified by **Dirname** must end with a /. If a leading / is not specified, the Dirname will be appende to the current prefix.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
DATA WORD VOLNAME = "/ACCOUNTS/"  ;Desired volume name
...
IF SETPREFIX(VOLNAME)
  RECEIVEABLES
  ...
ELSE
  PUT NL,"Can't find ", VOLUME," disk"
...
```

FUNC STRREAL CONVERT NUMERIC STRING TO REAL VALUE

USAGE: Bytevariable = **STRREAL** (String, #Variable)

STRREAL is a function which decodes (converts) a string into a numeric value of type REAL. The first argument is the address of the desired string. The second argument is the **address** of the REAL variable to receive the value represented by the string. The string may have any number of leading blanks and optionally a leading minus sign (-). The string may express the number in normal notation or scientific notation (E-format). Conversion proceeds until a character is encountered which cannot legally be part of the number (such as a trailing blank, end-of-line, comma, etc). The function returns a result of type BYTE which is an index to this delimiter. A returned value of 0 indicates no legal digits were encountered, probably indicating an error condition.

EXAMPLE:

```
INCLUDE LIBRARY
BYTE LINE[81]
REAL VELOCITY
BYTE INDEX
...
BEGIN
...
GETL LINE
INDEX=STRREAL(LINE, #VELOCITY)
IF VELOCITY < 0.0
  ...
```


This would read a line from the keyboard and install the value of the number typed into the variable VELOCITY. Some examples of acceptable input are shown below:

```
0     3.14     9070     .077     7856.004     -200000    -5.56333308E-11
```

CAUTION: Be sure to remember to specify the # operator in front of the REAL variable to receive the value.

A general purpose numeric input routine, INPUTR, is described in **Chapter 5** of the PROMAL LANGUAGE MANUAL and is provided on disk file INPUTR.S.

FUNC STRVAL CONVERT NUMERIC STRING TO WORD OR INT VALUE

USAGE: Bytevar = **STRVAL** (String, #Variable [,Radix [,Maxfield]])

STRVAL is a function which decodes (converts) a string into a numeric value. STRVAL may have two to four arguments. The required arguments are **String**, the desired string, and **#Variable**, the **address** of a WORD or INT variable (**not BYTE or REAL!**) to receive the value represented by the string. **Radix** is an optional conversion base defaulting to base 10, and **Maxfield** is an optional maximum field width defaulting to 255 characters. The value to be converted may be signed or unsigned. The string may have any number of leading blanks. Conversion proceeds until **Maxfield** characters are used from the string or until a character is encountered which cannot legally be a digit in a number in the specified or default base. A byte variable is returned as an index to this delimiter.

EXAMPLE 1:

Assume that the following program segment inputs the line, " 123,456" from the keyboard (without the quotes):

```
INCLUDE LIBRARY
...
BYTE LINE[81]
WORD XDIST
WORD YDIST
...
BEGIN
GETL LINE
BINDEX = STRVAL (LINE,#XDIST)
```

This will install the value 123 decimal in XDIST and set BINDEX=4. If desired, additional statements could determine that the delimiter was "," and so decode any additional values (such as the 456):

EXAMPLE 2 (continued from Example 1 above):

```
IF LINE[BINDEX]=',
  BINDEX=STRVAL (LINE+BINDEX+1, #YDIST)
```


EXAMPLE 3:

Assume BUF contained "BDF30A". Then:

```
BINDEX=STRVAL(BUF,#PC,16,3)
```

would set PC to $0BDF and return BINDEX=3, because a maximum field width of three characters was specified. Base 16 decoding was specified. Any radix between 2 and 36 can be used.

NOTES:
1. Be sure to remember to specify the # operator in front of the variable name to receive the numeric value. If you forget it, the value will be installed in memory at whatever address happens to be in that variable at the time!
2. If you wish to input a number of type BYTE, first use STRVAL with a destination of type WORD, and then copy the low byte to the final destination. If you use STRVAL to decode directly to a BYTE variable, the following byte in memory will also be affected.
3. If you wish to input a number of type REAL, use function STRREAL.
4. If the function returns 0 (no digits), the variable is also set to 0.
5. If frequent numeric input is anticipated from the keyboard, you may wish to use the following function (which can be found as file INPUTW.S on a PROMAL disk), which displays a specified prompt and returns a WORD value typed from the keyboard:

```
FUNC WORD INPUTW ; Prompt
  ; Output PROMPT, accept line of numeric input from keyboard,
  ; return the numeric value.
ARG WORD PROMPT    ; Desired prompting message
WORD TEMP          ; Temporary value
BYTE INDEX         ; Number of digits input
OWN BYTE BUF[10]   ; Buffer for keyboard input
BEGIN
REPEAT
  PUT NL,PROMPT    ; Display prompt
  GETL BUF,10      ; Input line
  INDEX=STRVAL(BUF,#TEMP)  ; Convert to numeric value
  IF INDEX=0       ; Invalid entry?
    PUT NL,"Please enter a numeric value."
UNTIL INDEX > 0
RETURN TEMP
END
```

The following example illustrates the use of this function:

```
WORD ILINE
WORD MAXLINE
...
BEGIN
ILINE = INPUTW("What line do you wish to go to? ")
IF ILINE > MAXLINE
  ...
```

FUNC SUBSTR LOCATE SUBSTRING IN STRING

USAGE: Bytevar = **SUBSTR**(Wantstring, Trystring [,Fold [,Max [,Limit]]])

Function SUBSTR searches a string **Trystring** for the presence of another string, **Wantstring**. **Fold** is an optional argument defaulting to FALSE, which, if TRUE, causes lower case letters to be treated as matching upper case letters. **Max** is an optional argument specifying the last character position in **Trystring** at which the match can start, defaulting to LENSTR(Trystring). For instance, if **Max** is 1, then the match must occur starting with the first character of **Trystring**. **Limit** is an optional argument specifying the number of characters in **Wantstring** which must match, defaulting to LENSTR(Wantstring). For example, if **Limit**=2, then SUBSTR will consider a match made if the first two letters of **Wantstring** are found in **Trystring**. The function returns zero if no match is found, or an index to the character **plus one** if it is found.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
DATA WORD ASTRING = "PROMISE ME PROMAL FOR MY BIRTHDAY"
DATA WORD WSTRING = "PROMAL"
BYTE TRY1
BYTE TRY2
BYTE TRY3
...
TRY1=SUBSTR(WSTRING,ASTRING)
TRY2=SUBSTR(WSTRING,ASTRING,TRUE,20,4)
TRY3=SUBSTR(WSTRING,ASTRING,TRUE,10)
```

will set TRY1 to 12, TRY2 to 1, and TRY3 to 0.

FUNC TESTKEY TEST IF A KEY IS PRESSED

USAGE: Bytevar = **TESTKEY** [(#Char)]

Function TESTKEY tests if a key is pressed on the keyboard. If not, it returns 0. If a key is pressed, it is returned as the value of the function and also is installed in the optional character address if specified. The character is not echoed to the display. The key code returned will be ASCII as given in **Appenidx B**. Testkey does not display or blink the cursor.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
REPEAT
  NOTHING
UNTIL TESTKEY
```


This waits for any key depression without echoing it to the screen.

NOTE:
1. **CAUTION**: be sure to remember to specify the # operator in front of the variable name to receive the key value.
2. The Commodore 64 "Kernal" ROM software does not support ongoing keydown detection. Therefore calling TESTKEY in a loop will not return another non-0 result until the previous key is released on the Commodore. However, the space bar will auto-repeat at about 10 "hits" per second.
3. For the Apple II, all keys auto-repeat after a brief pause.

FUNC TOUPPER CONVERT LOWER CASE CHARACTER TO UPPER CASE

USAGE: Bytevar = **TOUPPER** (Char)

TOUPPER is a function which takes a single argument of type BYTE and returns an argument of type BYTE. If the argument is a lower case letter, the returned value is the upper case equivalent; otherwise, the argument is returned unchanged.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
PUT NL,"Do you wish to accept your mission? "
IF TOUPPER(GETC)='Y'  ; accept 'y' or 'Y'
  TAKEMISSION
...
```

EXAMPLE 2:

```
FUNC WORD UPPERSTRING ; String
  ; Convert all lowercase chars to uppercase in string.
  ; Return same string, updated in place.
ARG WORD STRING     ; String to convert to uppercase
WORD I              ; Address of character in string
BEGIN
I=STRING            ; Addr of 1st char of string
WHILE I@<           ; Not end of string
  M[I]=TOUPPER(I@<) ; Convert if is lowercase
  I=I+1             ; Address of next char
RETURN STRING
END
```

NOTE:
1. The argument for TOUPPER must be a single character, not a string.

PROC WORDSTR CONVERT UNSIGNED VALUE TO STRING

USAGE: **WORDSTR** Value, #Buf [,Radix [,Minfield [,Padding]]]

Procedure WORDSTR is the inverse function of STRVAL. It takes an unsigned value and generates the ASCII string representing the value. **Value** is the desired value to encode and **#Buf** is the address of the buffer to receive the ASCII characters. **Radix** is the optional base to be used, defaulting to 10. **Minfield** is the minimum field width to generate, defaulting to 0. If a minimum field width is specified, the number will always be right-justified in the field. If more characters are required to output the number than are specified for the minimum field width, they will be encoded without any error indication. **Padding** is an optional character (not string) argument which is the padding character desired to fill out the buffer to the minimum field width, defaulting to blank.

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE BUF[8]
BEGIN
ADDR=$FFFF-1
...
WORDSTR ADDR, BUF
```

This will install the string "65534" into BUF.

EXAMPLE 2:

```
WORDSTR $BD, BUF, 16, 4, '0'
```

This will install "00BD" into BUF.

NOTE:
1. If you wish to convert a real number, use procedure REALSTR. To convert a signed integer, use procedure INTSTR.

FUNC ZAPFILE DELETE FILE

USAGE: Bytevar = **ZAPFILE** (Filename [,Wildflag])

Function ZAPFILE deletes a file (or optionally, a group of files). **Filename** is a string argument specifying the file to be deleted. For the Commodore 64, it can have an optional drive number prefix. For the Apple II, it may have a pathname. The optional argument **Wildflag** is a byte value defaulting to FALSE. If TRUE, the **Filename** argument can include the wildcard characters ? and *. In this case, all files matching the pattern will be deleted. Wildcards are not supported for the Apple II. The function returns 0 if successful and an error number as described for OPEN if not. However, an attempt to delete a file which does not exist is not considered to be an error, because this is the way the Commodore ROMs work. If you wish to flag an

attempt to delete a non-existent file as an error, you can do it by first doing a DIR to determine if it exists and issuing an error message if it doesn't. For the Apple II, ZAPFILE can be used to delete a subdirectory, provided it has no files left in it (see note 2 below).

EXAMPLE:

```
INCLUDE LIBRARY
BYTE ZAPERROR
...
BEGIN
...
ZAPERROR=ZAPFILE(CARG[1])
CHOOSE ZAPERROR
0
  PUT NL,CARG[1]," deleted."
2
  PUT NL,CARG[1]," is not a legal file name."
7
  PUT NL,"Not deleted, disk is write-protected."
ELSE
  PUT NL,"Not deleted, error."
...
```

NOTES:
1. For Commodore 64, if you want to delete a file which is not type SEQ, does not have a file extension, or has any lower case letters, you will have to set the NOFNCHK flag to $80 (defined in file PROSYS.S).
2. For the Apple II, setting NOFNCHK=$80 will allow file names with no extension or empty subdirectories to be deleted. NOFNCHK is defined in file PROSYS.S

