TI-83 Plus System Routines

Third Release – Jan. 25, 2002

Important Information

Contents:

17. System Routines — Miscellaneous

Reference List — System Routines

Glossary

Overview — System Routines

The following is the format in which each of the entry points will appear. The entry points are listed alphabetically by category.

Entry point name:	Name used to identify the routine.
Category:	Each entry point is identified by function into a category.
Description:	Brief description of usage/purpose. How the routine works and additional information about the input.
Inputs:	
Registers:	Setup values in processor registers.
Flags:	Setup values in processor flags (F register).
Others:	OPX, stack or RAM locations initial conditions affecting results.
Outputs:	
Registers:	Return information in processor registers.
Flags:	Return information in process flags.
Others:	Return information in OPX, stack, or RAM.
Registers destroyed:	Processor registers whose initial values may be modified, so caller is responsible for preserving.
RAM used:	RAM space needed, where applicable.
Remarks:	Description of appropriate usage context, limitations, and any other useful information, side effects, assumptions, etc.
Example:	An example of how to set up initial conditions and use the routine.

NOTE () indicate indirection

1 System Routines — Display

Bit_VertSplit

Category: Display

Description: Tests if the TI–83 Plus is set to G-T (graph-table) display mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: NZ = 1 if G-T mode is set

Others: None

Registers destroyed: None

Remarks: Applications may want to reset the 83+ to full screen mode if graphing functionality is used. In G-T mode the screen is split vertically with 1/2 being the graph screen and the other the table display.

Example:

```
        B_CALL      Bit_VertSplit          ; test for G-T mode
        JR          NZ,Screen_is_Split     ; jump if G-T mode
```

CheckSplitFlag

Category: Display

Description: Checks if either horizontal or G-T split screen modes are active.

Inputs:

Registers: None

Flags: grfSplitOverride, (IY + sGrFlags) = 1 to ignore split mode settings
This flag is set to make system routines draw to the full screen even when in a split screen mode.

Others: None

Outputs:

Registers: None

Flags: Z = 1 if no split screen mode is active
= 0 if a split screen mode is active

Others: None

Registers destroyed: None

Remarks:

Example:

```
B_CALL    CheckSplitFlag
```

ClearRow

Category: Display

Description: Clears eight consecutive LCD display drive rows.

Inputs:

Registers: A = LCD display driver row coordinate (0x80 – 0xBF)

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Eight pixel rows cleared
Driver left in X increment mode

Registers destroyed: A, B, DE

Remarks: This routine requires A to be in LCD display driver row (X) coordinates, which have a valid range between 0x80 – 0xBF, with the top pixel row equal to 0x80 and the bottom pixel row equal to 0xBF. Passing in a value for A outside this range will cause unpredictable results and probably a lockup. This routine erases eight consecutive rows, so if you pass in A = 0x88, the 9th – 16th pixel rows from the top of the display are erased. If you pass in a value between 0xB9 – 0xBF, the erased rows wrap back to the top of the display. In normal usage, if you are erasing a line of large text, the A value should be a multiple of 0x08.

Example:

ClrLCD

Category: Display

Description: Clears the display.

Inputs:

Registers: None

Flags: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.
grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set
vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: This routine only acts on the display, not the ***textShadow***.

Example: Clear the display using the current split settings:

```
B_CALL     ClrLCD
```

ClrLCDFull

Category: Display

Description: Clears the display ignoring any split screen settings.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Entire display is cleared.

Registers destroyed: All

Remarks:

Example:

```
B_CALL     ClrLCDFull
```

ClrOP2S

Category: Display

Description: Sets the floating-point number in OP2 to be positive.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks:

Example:

```
B_CALL     ClrOP2S
```

ClrScrn

Category: Display

Description: Clears the display. If ***textShadow*** is in use clears it also.

Inputs:

Registers: None

Flags: appTextSave,(IY + appFlags) = 1 if the ***textShadow*** is to be cleared also

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set

vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set

grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: Display and possibly ***textShadow*** cleared.

Registers destroyed: All

Remarks:

Example:

```
B_CALL     ClrScrn
```

ClrScrnFull

Category: Display

Description: Clears the display entirely ignoring split screen settings. If ***textShadow*** is in use clears it also.

Inputs:

Registers: None

Flags: appTextSave, (IY + appFlags) = 1 if the ***textShadow*** is to be cleared also

Others: None

Outputs:

Registers: None

Flags: None

Others: Display and possibly ***textShadow*** cleared.

Registers destroyed: All

Remarks:

Example:

```
B_CALL     ClrScrnFull
```

ClrTxtShd

Category: Display

Description: Clears the ***textShadow*** buffer.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** ***textShadow*** set to spaces.
- **Flags:** None
- **Others:** None

Registers destroyed: BC, DE, HL

Remarks: ClrScrn falls into this routine which zeros out 128 bytes starting at ***textShadow*** (one byte for each 5 x 7 screen position (8 rows x16 columns)).

Example:

DispDone

Category: Display

Description: Displays Done on text screen.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: HL

Remarks:

Example:

```
B_CALL      DispDone
```

DispHL

Category: Display

Description: Converts the contents of HL to a decimal and writes it to the screen at current cursor position. The string displayed is always 5 characters and right justified. The large 5x7 font is used.

Inputs:

Registers: HL = two-byte value to convert

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: String displayed. (OP1) = start of five character decimal number string, right justified.

Registers destroyed: AF, DE, HL

Remarks: If the string does not fit on the current display row then it is truncated at the screen's edge.

Example: Set HL = 357 and display it starting in row 0 column 0.

```
             LD          HL,0
             LD          (curRow),HL     ; set cursor position
;
             LD          HL,357
             B_CALL      DispHL
             RET
```

what will be displayed is " 357", which has two leading spaces.

DisplayImage

Category: Display

Description: Displays a bitmap image stored in RAM.

Inputs:

Registers: HL = pointer to image structure
Height of image in pixels — one-byte
Width of image in pixels — one-byte
Image data by rows

The first byte contains the data for the first eight-pixels of the first row. Bit 7 is the left-most pixel of the first row.
Each new row starts on a byte boundary.

There may be unused bits in the last byte of each row if the image is not a multiple of eight in width.

DE = location on screen to place the upper left corner of the image. (row, column)

(0,0) = upper left corner of the screen.

The image can be oriented off of the screen: ffh = -1. The only restriction is that the image cannot be entirely off screen.

Flags:

plotLoc, (IY + plotFlags)	=	1 if image drawn to display only.
	=	0 if image drawn to display and graph buffer.
bufferOnly, (IY + plotFlags)	=	1 if image drawn to graph buffer only. This flag overrides the plotLoc flag.

Others: None

Outputs:

Registers: None

Flags: None

Others: Screen, graph buffer
RAM locations @ ioPrompt - ioPrompt + 7

Registers destroyed: All

Remarks:

(continued)

DisplayImage *(continued)*

Example: Display an image three-pixels high by 17 pixels wide at position (0,0) to the display only.

```
;
             LD          HL,ImageData             ; pointer to bitmap
             LD          DE,OP1
             LD          BC,11
             LDIR                                 ; copy image data to
                                                  ; RAM
;
             LD          HL,OP1                   ; pointer to image
             LD          DE,0                     ; position of upper
                                                  ; left corner
;
             SET         plotLoc,(IY+plotFlags)
;
             B_CALL      DisplayImage

ImageData:
             DB          3,17                     ; height, width
;
             DB          80h,3eh,10h              ; row 1, only bit 7
                                                  ; of the last byte
                                                  ; is used
             DB          11h,35h,0h               ; row 2
             DB          0ffh,01h,10h             ; row 3
```

DispOP1A

Category: Display

Description: Displays a floating-point number using either small variable width or large 5x7 font. The value is rounded to the current “fix” setting (on the mode screen) before it is displayed.

Inputs:

Registers: ACC = maximum number of digits to format for displaying

Flags: textInverse, (IY + textFlags) = 1 for reverse video
textEraseBelow, (IY + textFlags) = 1 to erase line below character
textWrite, (IY + sGrFlags) = 1 to write to graph buffer not display
fracDrawLFont, (IY + fontFlags) = 1 to use large font, not small font

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4

Remarks: Displaying stops if the right edge of the screen is reached.

Example:

EraseEOL

Category: Display

Description: Erases screen to end of line.

Inputs:

Registers: None

Flags: None

Others: curRow, curCol point to screen position.

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None, saves registers beforehand.

Remarks: curRow, curCol are also saved and restored.

If the sEditRunning, (IY + apiFlg3) flag is set (sfont running).

Example:

```
             LD          HL,0801h          ; curRow = 1, curCol = 8
             LD          (curRow),HL
             LD          A,'H'
             B_CALL      PutC
             LD          A,'I'
             B_CALL      PutC
             B_CALL      EraseEOL          ; clear to end of line
;
```

FormBase

Category: Display

Description: Converts a RealObj (single floating-point number) in OP1 into a displayable string.

Use the current mode settings SCI, ENG, NORMAL and FIX setting to format the string.

The output can also be formatted as a fraction or a Degrees, Minutes, Seconds (DMS) number.

Inputs:

Registers: None

Flags: To use the current format settings:
(Flags + fmtFlags) copies to (Flags + fmtOverride)

To override the current settings, modify the following flags:

Resetting the next two flags sets NORMAL display mode.
fmtExponent, (fmtOverride) = 1 for scientific display mode
fmtEng, (fmtOverride) = 1 for engineering display mode

Setting the next three flags will signify DMS formatting.
fmtBin, (fmtOverride)
fmtHex, (fmtOverride)
fmtOct, (fmtOverride)

Setting the next two flags will signify Fraction formatting.
fmtHex, (fmtOverride)
fmtOct, (fmtOverride)

Others: (fmtDigits) = 0FFh for FLOAT, no fix setting
= 0 – 9 if fix setting is specified
OP1 = value to format.

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated

Registers destroyed: All

Ram Used: OP1 – OP6

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

(continued)

FormBase *(continued)*

Example: Generate a random number and display it at the current cursor position. Use all the current format settings except force SCI formatting.

```
             B_CALL     Random                 ; OP1 = random number
;
             LD         A,(IY+fmtFlags)        ; get current format
                                               ; settings
             RES        fmtEng,A
             SET        fmtExponent, A         ; override current and
                                               ; set SCI formatting
             LD         (IY+fmtOverride),A     ; set override flags
;
             B_CALL     FormBase               ; generate the string
;
             LD         HL,OP3                 ; start of string
             B_CALL     PutS                   ; display string
```

FormDCplx

Category: Display

Description: Converts a CplxObj (pair of floating-point numbers) in OP1/OP2 into a displayable string.

Use the current mode settings SCI, ENG, NORMAL, FIX setting, and complex number display format to format the string.

The output can also be formatted as a fraction or a Degrees, Minutes, Seconds (DMS) number.

Inputs:

Registers: None

Flags: To use the current format settings:
(Flags + fmtFlags) copies to (Flags + fmtOverride)

To override the current settings, modify the following flags:

Resetting the next two flags sets the NORMAL display mode.
fmtExponent, (fmtOverride) = 1 for scientific display mode
fmtEng, (fmtOverride) = 1 for engineering display mode

These flags control the formatting of complex numbers.
rectMode, (fmtOverride) = 1 for rectangular complex display
fmtEng, (fmtOverride) = 1 for polar complex display

Setting the next three flags will signify DMS formatting.
fmtBin, (fmtOverride)
fmtHex, (fmtOverride)
fmtOct, (fmtOverride)

Setting the next two flags will signify Fraction formatting.
fmtHex, (fmtOverride)
fmtOct, (fmtOverride)

Others: (fmtDigits) = 0FFh for FLOAT, no fix setting
= 0 – 9 if fix setting is specified

OP1 = value to format

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in (fmtString), and is 0 terminated.

Registers destroyed: All

RAM used: OP1 – OP6

(continued)

FormDCplx *(continued)*

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

Example: Generate a random complex number and display it at the current cursor position. Use all the current format settings except force SCI formatting.

```
             B_CALL      Random                 ; OP1 = random number
             RST         rPushRealO1            ; save
;
             B_CALL      Random                 ; OP1 = random number
             B_CALL      PopRealO2              ; OP2 = 2nd part of
                                                ; floating-point number
;
             LD          A,(IY+fmtFlags)        ; get current format
                                                ; settings
             RES         fmtEng,A
             SET         FmtExponent, A         ; override current and
                                                ; set SCI formatting
             LD          (IY+fmtOverride),A     ; set override flags
;
             B_CALL      FormDCplx              ; generate the string
;
             LD          HL,fmtString           ; start of string
             B_CALL      PutS                   ; display string
```

FormEReal

Category: Display

Description: Converts a RealObj (single floating-point number) in OP1 into a displayable string.

This routine will ignore all format settings.

Specify the maximum width allowed for the string generated.

Inputs:

Registers: ACC = maximum width of output, minimum of six

Flags: None

Others: OP1 = value to format

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

Example: Generate a random number and display it with a maximum of six characters at the current cursor position. Ignore all format settings when generating the string to display.

```
             B_CALL     Random        ; OP1 = random number
             LD         A,6           ; max width to format value with
             B_CALL     FormEReal     ; generate the string
;
             LD         HL,OP3        ; start of string
             B_CALL     PutS          ; display string
```

FormReal

Category: Display

Description: Converts a RealObj (single floating-point number) in OP1 into a displayable string.

Specify the maximum width allowed for the string generated.

Inputs:

Registers: ACC = maximum width of output, minimum of six

Flags: fmtExponent, (fmtFlags) = 1 for scientific display mode
fmtEng, (fmtFlags) = 1 for engineering display mode

If both of the above flags are reset, then NORMAL display mode.

Others: (fmtDigits) = 0FFh for FLOAT, no fix setting
= 0 – 9 if fix setting is specified

OP1 = value to format

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

Example: Generate a random number and display it with a maximum of six characters at the current cursor position.

```
             B_CALL     Random       ; OP1 = random number
             LD         A,6          ; max width to format value with
             B_CALL     FormReal     ; generate the string
;
             LD         HL,OP3       ; start of string
             B_CALL     PutS         ; display string
```

LoadPattern

Category: Display

Description: Loads the font pattern for a character to RAM. Also includes the characters width in pixels. This will work for both variable width and 5x7 fonts.

Inputs:

Registers: ACC = character equate

Flags: fracDrawLFont, (IY + fontFlags) = 1 to use Large 5x7 font
= 0 to use variable width font

Others: None

Outputs:

Registers: None

Flags: None

Others: For large 5x7 font: RAM @ lFont_record = width of character, seven-byte font

For variable width font: RAM @ sFont_record = width of character, seven-byte font

The first byte of the font is the pixel mapping for the top row and each subsequent byte is the next row.

The LSB of each byte represents the right most pixel of a row.

Registers destroyed: All

RAM used:

Remarks: If fracDrawLFont is set, it must be reset.

Example:

Load_SFont

Category: Display

Description: Copies small font attributes to RAM for a particular display character.

Inputs:

Registers: HL = offset into small font table

Flags: None

Others: None

Outputs:

Registers: HL = pointer to sFont_record RAM

Flags: None

Others: sFont_record...sFont_record + 7 = font

Registers destroyed: DE, HL

Remarks: This might be useful, if you wish to write your own **LoadPattern** or **VPutMap** routine for displaying small display characters. The system character fonts (large and small) use eight-bytes per character.

To convert a character number to a table offset, multiply the number by eight.

Example: Find the width of the small display character f:

```
LD          A,'F'
LD          L,A
LD          H,0
ADD         HL,HL          ; * 2 turn character into an
                           ; offset.
ADD         HL,HL          ; * 4
ADD         HL,HL          ; * 8 multiply by 8 to get
                           ; table offset.
B_CALL      Load_SFont     ; sFont_record =
                           ; 03,00,02,04,06,04,04,00
LD          A,(HL)         ; 1st byte is width
```

NewLine

Category: Display

Description: Move cursor to beginning of next line and scroll the display if necessary.

Inputs:

Registers: None

Flags: appAutoScroll, (IY+appFlags) = 1 to automatically scroll display

Others: None

Outputs:

Registers: None

Flags: textScrolled, (IY+textFlags) = 1 if display scrolled

Others: (curRow) is incremented if display does not scroll.

(curCol) = 0.

Registers destroyed: All

Remarks: Presumes that (winTop) has been previously initialized to the top of the window and (winBtm) has been initialized to the bottom of the window. (eg.. usually winBtm = 8 and winTop = 0. In horizontal split screen, winTop = 4).

Reset the appAutoScroll (IY+appFlags) flag to avoid scrolling the screen if on the bottom line. But if doing so, curRow may be incremented to an invalid state (eg, row 8 or above), so this condition needs to be checked and curRow re-initialized if you use this flag.

Example:

OutputExpr

Category: Display

Description: Converts a numeric value, string or equation, into a string and displays it using the large 5x7 font. This routine should be used with the split screen setting to set to FullScreen.

Inputs:

Registers: H = column number to display at: e.g., 0...15
L = row number to display at: e.g., 0...7

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video
appTextSave, (IY + appFlags) = 1 to write character to ***textShadow*** also

Others: OP1/OP2 = what to display:
- Floating-point number in OP1
- Complex number in OP1/OP2
- A variable name in OP1 of type: complex, list (real/complex), matrix, string, equation.

Outputs:

Registers: None

Flags: None

Others: System errors can be generated, See the Error Handlers section in Chapter 2.

String output to display.

Registers destroyed: All

Remarks: Previous cursor setting is restored to curRow and curCol. Output will wrap to next line if complete string does not fit on a single line. Output will stop at bottom of screen.

Example: Output the contents of matrix variable [A] at cursor location row 2, column 3.

```
             LD          HL,matAname
             RST         rMov9ToOP1      ; OP1 = matrix [A] name
;
             AppOnErr    Catch_Error     ; install error handler
;
             LD          HL,3*256+2      ; row 2 column 3
             B_CALL      OutputExpr
;
             AppOffErr
;
Catch_Error:
             RET
```

PutC

Category: Display

Description: Displays a character and advance cursor.

Inputs:

Registers: A = character to display

Flags: textInverse, (IY+textFlags): 0 = normal character; 1 = invert character

Others: curRow, curCol = display row and column values

Outputs:

Registers: None

Flags: None

Others: curRow, curCol Updated

Registers destroyed: None

Remarks: This routine calls **PutMap** to do the character display.

This may cause a screen scroll if on the bottom line.

Example:

```
              LD          HL,0801h         ; curRow = 1, curCol = 8
              LD          (curRow),HL
              LD          A,"H"
              B_CALL      PutC
              LD          A,"I"
              B_CALL      PutC
;(PutS might be more useful for multiple characters)
```

PutMap

Category: Display

Description: Displays a character in the large font without affecting cursor position.

Inputs:

Registers: ACC = character to display, see TI83plus.inc

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video
appTextSave, (IY + appFlags) = 1 to write char to ***textShadow*** also
preClrForMode, (IY + newDispF) = 1 to preclear the character space before writing
This is done when toggling between inverted and uninverted.

Others: (curRow) = home screen row to display in, 0-7
(curCol) = home screen column to display in, 0-15

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: See: **PutC**.

Example: Display char C in row 3 column 4:

```
             LD          HL,4*256+3
             LD          (curRow),HL     ; set curRow & curCol
;
             LD          A,'C'
             B_CALL      PutMap
;
```

PutPS

Category: Display

Description: Displays a string with a leading length byte residing in RAM, at the current cursor position, and stops at the bottom of the display. This routine uses the large 5x7 font.

Inputs:

Registers: HL = pointer to length byte of string followed by the string

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video

appAutoScroll, (IY + appFlags) = 1 to scroll if need to display past the bottom of the display.

appTextSave, (IY + appFlags) = 1 to write character to ***textShadow*** also.

preClrForMode, (IY + newDispF) = 1 to preclear the character space before writing. This is done when toggling between inverted and noninverted.

Others: (curRow) = cursor row position, (0 – 7)
(curCol) = cursor column position, (0 – 15)

Outputs:

Registers: None

Flags: Carry = 1 if entire string was displayed
Carry = 0 if string did not fit in the display

Others: curRow and curCol are updated to the position after the last character displayed.

Registers destroyed: All but DE

Remarks: It is recommended that this routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

PutPS *(continued)*

Example:

```
PutPS:
            LD          A,(HL)      ; A = length of string
            INC         HL
            OR          A
            RET         Z           ; IF LENGTH IS 0 RET
PutPS10:
            LD          A,(HL)      ; get a character of string name
            INC         HL
PutPS20:
            B_CALL      PutC        ; display one character of string
PutPS30:
            LD          A,(curRow)
            LD          C,A
            LD          A,(winBtm)
            CP          C           ; IS CURSOR OFF SCREEN ?
            RET         Z           ; RET IF YES
;
            DJNZ        PutPS10     ; display rest of string
            RET
```

PutPSB

Category: Display

Description: Displays a string with a leading length byte residing in RAM, at the current cursor position, and stops at the right edge of the display. Ignores leading spaces. This routine uses the large 5x7 font.

Inputs:

Registers: HL = pointer to length byte of string followed by the string

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video

appTextSave, (IY + appFlags) = 1 to write character to ***textShadow*** also.

preClrForMode, (IY + newDispF) = 1 to preclear the character space before writing. This is done when toggling between inverted and noninverted.

Others: (curRow) = cursor row position, (0 – 7)
(curCol) = cursor column position, (0 – 15)

Outputs:

Registers: None

Flags: Carry = 1 if entire string was displayed
Carry = 0 if string did not fit in the display

Others: curRow and curCol are updated to the position after the last character displayed.

Registers destroyed: All but DE

Remarks: It is recommended that this routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

PutPSB *(continued)*

Example:

```
PutPSB:
            LD          A,(HL)       ; A = length of string
            LD          B, A
            INC         HL
            OR          A
            RET         Z            ; IF LENGTH IS 0 RET
            LD          A, (HL)
            CP          ' '
            JR          Z, PutPSB30
            JR          PutPSB20
PutPSB10:
            LD          A, (curCol)  ; get column to print string
            CP          15           ; last column?
            JR          C, PutPSB15  ; no, do regular PutC
            LD          A, (HL)      ; get a character of string name
            B_CALL      PutMap       ; output character without newline
            JR          PutPSB30
PutPSB15:
            LD          A, (HL)      ; get a character of the string
PutPSB20:
            B_CALL      PutC         ; display one character of string
PutPSB30:
            INC         HL
            DJNZ        PutPSB10     ; display rest of string
            RET
```

PutS

Category: Display

Description: Displays a zero (0) terminated string residing in RAM at the current cursor position. This routine uses the large 5x7 font.

Inputs:

Registers: HL = pointer to start of string

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video

appAutoScroll, (IY + appFlags) = 1 to scroll if need to display past the bottom of the display.

appTextSave, (IY + appFlags) = 1 to write character to ***textShadow*** also.

preClrForMode, (IY + newDispF) = 1 to preclear the character space before writing. This is done when toggling between inverted and noninverted.

Others: (curRow) = cursor row position, (0 – 7)
(curCol) = cursor column position, (0 – 15)

Outputs:

Registers: None

Flags: Carry = 1 if entire string was displayed
Carry = 0 if string did not fit in the display

Others: curRow and curCol are updated to the position after the last character displayed.

Registers destroyed: HL

Remarks: To avoid having to copy strings from an application to RAM before using this routine, it is much more efficient to place this routine inside of the application. By doing so, the application can display strings without first having to copy to RAM.

(continued)

PutS *(continued)*

Example:

```
PutS:
            PUSH      BC
            PUSH      AF
            LD        A,(winBtm)
            LD        B,A          ; B = bottom line of window
PutS10:
            LD        A,(HL)       ; get a character of string name
            INC       HL
            OR        A            ; end of string?
            SCF                    ; indicate entire string was
                                   ; displayed
            JR        Z, PutS20    ; yes --->
            B_CALL    PutC         ; display one character of string
;
            LD        A,(curRow)   ; check cursor position
            CP        B            ; off end of window?
            JR        C,PutS10     ; no, display rest of string
PutS20:
            POP       BC           ; restore A (but not F)
            LD        A,B
            POP       BC           ; restore BC
            RET
```

PutTokString

Category: Display

Description: Displays the string for a token at the current cursor location.

Inputs:

Registers: DE = token value. If a one-byte token then D = 0, E = token.

Flags: None

Others: (curRow) = home screen row to display in, 0 - 7
(curCol) = home screen column to display in, 0 - 5

Outputs:

Registers: None

Flags: None

Others: String displayed with wrapping.

Registers destroyed: All

Remarks:

Example: Display the string for the Sin(token at the current cursor location:

```
             LD        D,0
             LD        E,tSin          ; DE = token
;
             B_CALL    PutTokString    ; get its string and display
                                       ; it.
;
```

RestoreDisp

Category: Display

Description: Displays one to 64 rows of the display starting with the top row.

Inputs:

Registers: HL = pointer to ROM/RAM of the data for the first row to display, from left to right. This is followed by the remaining row's data. Each row is stored in 12-bytes, the first column is bit seven of the first byte for each row.

B = number of pixel rows to be displayed

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Data written to the display.

Interrupts are disabled, turn them back on if needed.

Registers destroyed: All

RAM used: curXRow — 1 byte

Remarks:

Example: Copy the first 10 lines of the graph buffer to the display.

```
             LD          HL,plotSScreen    ; start of buffer
             LD          B,10              ; 10 rows to display
;
             B_CALL      RestoreDisp
;
             EI                            ; re-enable interrupts
;
```

RunIndicOff

Category: Display

Description: Turns off run indicator.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** indicRun, (IY+indicFlags) = 0
- **Others:** None

Registers destroyed: Flag register

Remarks:

Example:

```
B_CALL     RunIndicOff
```

RunIndicOn

Category: Display

Description: Turns on run indicator.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: indicRun, (IY+indicFlags) = 1

Others: None

Registers destroyed:

Remarks:

Example:

```
B_CALL     RunIndicOn
```

SaveDisp

Category: Display

Description: Copies a bit image of the current display to RAM.

Inputs:

Registers: HL = pointer to RAM location to save the image — the bit image of the display is 768 bytes in size.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Contents of display written to RAM. Interrupts are disabled.

Registers destroyed: All

RAM used: curXRow

Remarks: Split screen modes are ignored, the entire display is copied.

Example: Copy the current display to the graph backup buffer, ***plotSScreen***.

```
LD          HL,plotSScreen
B_CALL      SaveDisp
RET
```

SetNorm_Vals

Category: Display

Description: Sets display attributes to full screen mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Display attributes set to full screen. Allows for full screen drawing and text displaying.

Registers destroyed: All

Remarks: This routine should only be used in combination with the setting of appropriate system flags that control the screen split settings. See the Display and Split Screen Modes sections in Chapter 2 for further information.

Example:

SFont_Len

Category: Display

Description: Returns the width, in pixels, a character would use if displayed using the small variable width font.

Inputs:

Registers: HL = offset into the font look-up table. This is generated by multiplying the character equate of a character by eight.

Flags: None

Others: None

Outputs:

Registers: ACC = number of pixels needed to display the character using the small font.

Flags: None

Others: None

Registers destroyed: All B

Remarks:

Example: Return the width in pixels of the small font character:

```
LD          HL,Scolon*8     ; compute offset
B_CALL      SFont_Len
```

SStringLength

Category: Display

Description: Returns the width in pixels a string would use if displayed using the small variable width font.

Inputs:

Registers: HL = pointer to the string, with the first byte being the number of characters in the string. The string must reside in RAM.

Flags: None

Others: None

Outputs:

Registers: ACC and B = number of pixels needed to display the string using the small font.

Flags: None

Others: None

Registers destroyed: All but HL

Remarks:

Example:

VPutMap

Category: Display

Description: Displays a character at the current pen location. Uses either the variable width font or the large 5x7 font.

The advantage to displaying the large font with this routine instead of the **PutC** routine is the character can be placed at any location on the screen. With **PutC** routine, the characters can only be displayed in the 8 row by 16 column grid specified by (curRow) and (curCol).

Inputs:

Registers: ACC = character to display

Flags: textInverse, (IY + textFlags) = 1 for reverse video
textEraseBelow, (IY + textFlags) = 1 to erase line below character applies to variable width font only
textWrite, (IY + sGrFlags) = 1 to write to graph buffer instead of the display
fracDrawLFont, (IY + fontFlags) = 1 to use large font, not small font

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: CA (carry) = 1 if could not fit in screen

Registers destroyed: All but BC and HL

Remarks: Pen location (0,0) is the upper left corner of the display.

The formatting flags are normally reset. An application should make sure that these flags are managed properly during execution and reset before returning to normal system operation.

Example: Draw the character C at pen location (0,0):

```
LD          HL,0
LD          (penCol),HL      ; set penRow and penCol
LD          A,'C'
B_CALL      VPutMap
```

VPutS

Category: Display

Description: Displays a zero (0) terminated string at the current pen location. Uses either the variable width font or the large 5x7 font.

The advantage to displaying the large font with this routine instead of the **PutS** routine is the string can be placed at any location on the screen. With the **PutS** routine, the string can only be displayed in the 8 row by 16 column grid specified by (curRow) and (curCol).

Inputs:

Registers: HL = pointer to 0 terminated string in RAM.

Flags:

textInverse, (IY + textFlags)	=	1 for reverse video
textEraseBelow, (IY + textFlags)	=	1 to erase line below character
textWrite, (IY + sGrFlags)	=	1 to write to graph buffer not display
fracDrawLFont, (IY + fontFlags)	=	1 use 5x7 font
	=	0 use variable width font (default)

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: CA = 1 if could not fit on the row of the screen entirely

Registers destroyed: HL

Remarks: Pen location (0,0) is the upper left corner of the display. If fracDrawLFont is set, it must be reset. It is recommended that the following routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

VPutS *(continued)*

```
VPutS:
            PUSH      AF
            PUSH      DE
            PUSH      IX
VPutS10:
            LD        A,(HL)        ; get a character of string name
            INC       HL
            OR        A             ; end of string?
            JR        Z, VputS20    ; yes --->
            B_CALL    VPutMap       ; display one character of string
            JR        NC, VPutS10   ; display rest of string IF FITS
VputS20:
            POP       IX
            POP       DE
            POP       AF
            RET
```

Example: Display Hello world in variable width font at the current pen location.

```
            LD          HL,Hellostr
            LD          DE,OP1
            LD          BC,14
            LDIR                          ; copy string to RAM
;
            LD          HL,OP1
            B_CALL      VPutS
;
            RET
;
Hellostr:
            DB          "Hello World",0
```

VPutSN

Category: Display

Description: Displays a string of known length at the current pen location. Uses either the variable width font or the large 5x7 font.

The advantage to displaying the large font with this routine instead of the **PutS** routine, is the string can be placed at any location on the screen. With the **PutS** routine, the string can only be displayed in the 8 row by 16 column grid specified by (curRow) and (curCol).

Inputs:

Registers: HL = pointer to first character of string in RAM
B = number of characters to display

Flags:

textInverse, (IY + textFlags)	=	1 for reverse video
textEraseBelow, (IY + textFlags)	=	1 to erase line below character
textWrite, (IY + sGrFlags)	=	1 to write to graph buffer not display
fracDrawLFont, (IY + fontFlags)	=	1 use 5x7 font
	=	0 use variable width font (default)

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: CA = 1 if could not fit on the row of the screen entirely

Registers destroyed: HL

Remarks: Pen location (0,0) is the upper left corner of the display. If fracDrawLFont is set, it must be reset. It is recommended that the following routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

VPutSN *(continued)*

```
VPutSN:
            PUSH      AF
            PUSH      DE
            PUSH      IX
PP10:
            LD        A,(HL)      ; get a character of string name
            INC       HL
            B_CALL    VPutMap     ; display one character of string
            JR        C, PP11     ; JUMP IF NO ROOM ON LINE
            DJNZ      PP10        ; display rest of string
PP11:
            POP       IX
            POP       DE
            POP       AF
            RET
```

Example: Display Hello world in variable width font at the current pen location.

```
            LD        HL,Hellostr
            LD        DE,OP1
            LD        BC,14
            LDIR                    ; copy string to RAM
;
            LD        HL,OP1
            LD        B,11          ; length of string
            B_CALL    VPutSN
;
            RET
;
Hellostr:
            DB        "Hello World"
```

2 System Routines — Edit

CloseEditBuf

Category: Edit

Description: Close and deletes edit buffer without parsing.

Inputs:

Registers: None

Flags: editOpen, (IY + editFlags) set if open

Others: None

Outputs:

Registers: None

Flags: None

Others: Adjusts free RAM pointers

Registers destroyed: All

Remarks: See **CloseEditBufNoR** for example.

CloseEditBufNoR

Category: Edit

Description: Closes edit buffer, but does not delete it.

Inputs:

Registers: None

Flags: editOpen, (IY + editFlags) set if open

Others: None

Outputs:

Registers: None

Flags: None

Others: Adjusts free RAM pointers

Registers destroyed: All

Remarks: An edit session allocates all available RAM, but generally only a portion of that RAM is actually used.

This routine is used to free up any extra RAM after an edit is finished and before the parser is invoked to evaluate the input.

Same as:

```
        B_CALL      CanAlphIns          ; cancel alpha and insert
                                        ; mode
        B_CALL      CloseEditEqu        ; return edit buffer to
                                        ; user memory
        RET
```

Example:

```
;
        B_CALL      IsEditEmpty         ; is edit buffer empty?
        JR          NZ, NotEmpty        ; no
        B_CALL      CloseEditBuf        ; close & delete buffer
                                        ; without parsing
        RET
NotEmpty:
        B_CALL      CloseEditBufNoR     ; close but do not delete
        CALL        AtName              ; Name of edit buffer
        B_CALL      ParseInp            ; parse. result -> OP1
                                        ; store result
        B_CALL      ReleaseBuffer       ; throw away edit buffer.
        RET
AtName:
        LD          HL, '@'
        LD          A, EquObj
        LD          (OP1), A
        LD          (OP1+1), HL
        XOR         A
        LD          (OP1+3), A
        RET
```

CloseEditEqu

Category: Edit

Description: Returns any unused portion of an edit buffer to memory.

Inputs:

Registers: None

Flags: editOpen,(IY+editFlags) = 1 if edit buffer is open

Others: None

Outputs:

Registers: None

Flags: None

Others: Adjusts free RAM pointers.

Registers destroyed: All

Remarks: See also: **CloseEditBufNoR**

CursorOff

Category: Edit

Description: Turns off the cursor if it is turned on and disable blinking.

Inputs:

Registers: None

Flags: curOn, (IY + curFlags) = 1 if cursor is currently on.

appCurGraphic, (IY + appFlags) = 1 if the graphic cursor
This mode should not be set by an application.

appCurWord, (IY + appFlags) = 1 if a full word cursor
This mode should not be set by an application.

Others: If a normal edit cursor:
(curRow), (curCol) = cursor location
(curUnder) = character the cursor is covering

If a graphic cursor:
(curGX), (curGY) = center pixel location of cursor
(curGStyle) = which graph cursor is active

If a full word cursor:
These are specific to the current context and entries are made in-line in the cursor blink routine.

Outputs:

Registers: None

Flags: curOn, (IY + curFlags) = is reset
curAble, (IY + curFlags) = is reset to disable future blinking

Others: None

Registers destroyed: All

Remarks:

Example:

CursorOn

Category: Edit

Description: Enables cursor blinking and show the cursor.

Inputs:

Registers: None

Flags: curLock, (IY + curFlags) = 1 if cursor is locked disabled, the cursor cannot be turned on to blink.

appCurGraphic, (IY + appFlags) = 1 if the graphic cursor
This mode should not be set by an application.

appCurWord, (IY + appFlags) = 1 if a full word cursor
This mode should not be set by an application.

Others: If a normal edit cursor:
(curRow), (curCol) = cursor location

If a graphic cursor:
(curGX), (curGY) = center pixel location of cursor
(curGStyle) = which graph cursor is active

If a full word cursor:
These are specific to the current context and entries are made in-line in the cursor blink routine.

Outputs:

Registers: None

Flags: curOn, (IY + curFlags) = is set
curAble, (IY + curFlags) = is set to enable future blinking

Others: (curUnder) = character the cursor is covering

Registers destroyed: All

Remarks:

Example:

DispEOL

Category: Edit

Description: Displays edit buffer to End of Line.

Inputs:

Registers: None

Flags: None

Others: editBuffer pointers

Outputs:

Registers: Display modified

Flags: None

Others: None

Registers destroyed: AF, BC, DE, HL

Remarks: Displays buffer from editTail to editBtm or until the end of the line is reached. If the buffer is finished before reaching the end of line, then **EraseEOL** is called to erase to the end of the line. Current curCol value is saved and restored by this routine; it is not modified. Since this routine only displays to the end of the current line, curRow is not modified.

Example:

IsEditEmpty

Category: Edit

Description: Tests if the Edit Buffer is empty. This is accomplished by confirming (editTail) equals (editBtm) AND (editCursor) equals (editTop).

Inputs:

Registers: None

Flags: None

Others: editTop, editCursor, editTail and editBtm pointer values must be valid - the edit session must be active.

Outputs:

Registers: None

Flags: Z = 1 (edit buffer is empty)

= 0 (edit buffer is not empty)

Others: None

Registers destroyed: A, DE, HL

Remarks: This module is essentially a B_Call to isAtBtm followed by a B_Call to isAtTop.

Refer to **isAtTop**, **isAtBtm** modules for additional operational details.

Example:

KeyToString

Category: Edit

Description: Converts key to a string value.

Inputs:

Registers: DE = key
D = 0 if a one-byte key

Flags: None

Others: None

Outputs:

Registers: HL = keyToStrRam (keyForStr + 1)

Flags: None

Others: keyForStr initialized to string

Registers destroyed: AF, BC, DE, HL

Remarks: Keys are converted to tokens (if possible) and the token string copied to the keyForStr RAM area (18 bytes).

HL points to the length byte of the string (in keyToStrRam).
See TI83plus.inc for key and token values.

Example: To display the string for the Continue key:

```
LD          D,0            ; "Continue" is a one byte key,
                           ; so set to 0.
LD          E,kCont        ; "Continue"
B_CALL      KeyToString    ; convert to string: HL points
                           ; to keyToStrRam.
B_CALL      PutPSB         ; display string preceded by a
                           ; length byte...
B_CALL      EraseEOL       ; erase the rest of the line if
                           ; need be.
```

keyToStrRam would appear as follows:

08h, 43h, 6Fh, 6Eh, 74h, 69h, 6Eh, 75h, 65h
(Length of string is eight bytes, followed by the ASCII characters Continue.)

See TI83plus.inc or Appendix B for the TI–83 Plus character set values.

ReleaseBuffer

Category: Edit

Description: Deletes numeric edit buffer.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: After evaluation and an edit buffer is no longer needed, it is important to delete that buffer so that it doesn't take up unnecessary RAM.

This routine can be included as part of an evaluation routine (if the buffer does not need to be redisplayed or edited), or as part of a putaway routine as you are leaving a context and returning back to the system.

See **CloseEditBufNoR** for example.

Example:

3 System Routines — Error

ErrArgument

Category: Error

Description: Jumps to system error handler routine with the message ERR: ARGUMENT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP    ErrArgument
```

ErrBadGuess

Category: Error

Description: Jumps to system error handler routine with the message ERR: BAD GUESS.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrBadGuess
```

ErrBreak

Category: Error

Description: Jumps to system error handler routine with the message ERR: BREAK.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrBreak
```

ErrD_OP1_0

Category: Error

Description: If OP1 = 0.0, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A

Remarks:

Example:

```
B_JUMP     ErrD_OP1_0
```

ErrD_OP1_LE_0

Category: Error

Description: If OP1   0 (not positive), domain error system will take over with message ERR: DOMAIN.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Registers destroyed: A

Remarks:

Example:

```
B_JUMP    ErrD_OP1_LE_0
```

ErrD_OP1Not_R

Category: Error

Description: If OP1 is not real, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A

Remarks:

Example:

```
B_JUMP     ErrD_OP1Not_R
```

ErrD_OP1NotPos

Category: Error

Description: If OP1 is not positive, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A

Remarks:

Example:

```
B_JUMP     ErrD_OP1NotPos
```

ErrD_OP1NotPosInt

Category: Error

Description: If OP1 is not positive integer, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A

Remarks:

Example:

```
B_JUMP     ErrD_OP1NotPosInt
```

ErrDataType

Category: Error

Description: Jumps to system error handler routine with the message ERR: DATA TYPE.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrDataType
```

ErrDimension

Category: Error

Description: Jumps to system error handler routine with the message ERR: INVALID DIM.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrDimension
```

ErrDimMismatch

Category: Error

Description: Jumps to system error handler routine with the message ERR: DIM MISMATCH.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrDimMismatch
```

ErrDivBy0

Category: Error

Description: Jumps to system error handler routine with the message ERR: DIVIDE BY 0.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrDivBy0
```

ErrDomain

Category: Error

Description: Jumps to system error handler routine with the message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrDomain
```

ErrIncrement

Category: Error

Description: Jumps to system error handler routine with the message ERR: INCREMENT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrIncrement
```

ErrInvalid

Category: Error

Description: Jumps to system error handler routine with the message ERR: INVALID.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrInvalid
```

ErrIterations

Category: Error

Description: Jumps to system error handler routine with the message ERR: ITERATIONS.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrIterations
```

ErrLinkXmit

Category: Error

Description: Jumps to system error handler routine with the message ERR: IN XMIT.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrLinkXmit
```

ErrMemory

Category: Error

Description: Jumps to system error handler routine with the message ERR: MEMORY.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrMemory
```

ErrNon_Real

Category: Error

Description: In Real mode, the result of a calculation yielded a complex result. This error is not returned during graphing. The TI–83 Plus allows for undefined values on a graph.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Registers destroyed:

Remarks: The error system will take over and report the error to the screen. Any application that was executing at that time will be aborted.

Example:

```
B_JUMP     ErrNon_Real
```

ErrNonReal

Category: Error

Description: Errors if nonreal input to command error. System will take over with message ERR: DATA TYPE.

Inputs:

Registers: B = number of arguments to check.

Flags: None

Others: Arguments on Floating Point Stack.

Outputs:

Registers: None

Flags: None

Others: Error if nonreal input to command.
Screen will have data type error menu.

Registers destroyed: A, B

Remarks:

Example:

```
B_JUMP     ErrNonReal
```

ErrNotEnoughMem

Category: Error

Description: If not enough memory, memory error system will take over with message ERR: MEMORY.

Inputs:

Registers: HL = number of bytes needed.

Flags: None

Others: None

Outputs:

Registers: DE = Amount of memory requested.

Flags: CA = 1 if not enough room.

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrNotEnoughMem
```

ErrOverflow

Category: Error

Description: Jumps to system error handler routine with the message ERR: OVERFLOW.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrOverflow
```

ErrSignChange

Category: Error

Description: Jumps to system error handler routine with the message ERR: NO SIGN CHANGE.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrSignChange
```

ErrSingularMat

Category: Error

Description: Jumps to system error handler routine with the message ERR: SINGULARITY.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP    ErrSingularMat
```

ErrStat

Category: Error

Description: Jumps to system error handler routine with the message ERR: STAT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP    ErrStat
```

ErrStatPlot

Category: Error

Description: Jumps to system error handler routine with the message ERR: STATPLOT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrStatPlot
```

ErrSyntax

Category: Error

Description: Jumps to system error handler routine with the message ERR: SYNTAX.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrSyntax
```

ErrTolTooSmall

Category: Error

Description: Jumps to system error handler routine with message ERR: TOL NOT MET.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrTolTooSmall
```

ErrUndefined

Category: Error

Description: Jumps to system error handler routine with the message ERR: UNDEFINED.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
B_JUMP     ErrUndefined
```

JError

Category: Error

Description: Entry point into system error routine. This entry is almost always used in conjunction with an error exception handler.

After an error exception handler is tripped and control is returned to an application, the application may choose to modify the error by changing the error to another or most likely removing the GoTo option. This entry point is where the application would B_JUMP to continue on with the error after modifying it.

See the Error Handers section in Chapter 2.

Inputs:

Registers: ACC bits (0 – 6) = error code
ACC bit (7) = 0 for no GoTo option
ACC bit (7) = 1 for allowing a GoTo option

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: (errNo) = error code (one-byte)
System error is displayed or another error.
Exception handler is tripped and the error is suppressed.

Registers destroyed: All

Remarks:

Example:

JErrorNo

Category: Error

Description: Same as JError except the error code is stored in the byte (errNo).

Remarks: See JError.

4 System Routines — Floating Point Stack

AllocFPS

Category: Floating Point Stack

Description: Allocates space on the Floating Point Stack by specifying a number of nine-byte entries.

Inputs:

Registers: HL = number of entries to allocate

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If no memory error, the new entries are allocated on the end of the FPS. FPST = last new entry allocated.

Registers destroyed: All

Remarks: No initialization of the allocated entries is done. See section on Floating Point Stack.

Example:

AllocFPS1

Category: Floating Point Stack

Description: Allocates space on the Floating Point Stack by specifying a number of bytes, THIS MUST BE A MULTIPLE OF NINE.

Inputs:

Registers: HL = number of bytes to allocate — a multiple of nine.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If no memory error, the new entries are allocated on the end of the FPS. FPST = last new entry allocated.

Registers destroyed: All

Remarks: No check is made for the number of bytes being a multiple of nine. No initialization of the allocated entries is done. See section on Floating Point Stack.

Example:

CpyStack

Category: Floating Point Stack

Description: Copies nine-bytes from one of the systems nine-byte stacks, FPS and ES. Only the FPS (Floating Point Stack) is documented for application use. This routine should be used in the manner described in the example.

Input:

Registers: C = number of bytes from the next free byte in the stack back to the entry copying from. This will always be a multiple of nine.

HL = address of next free byte for the stack, for the FPS the address is stored in the bytes (FPS).

DE = pointer to the nine-bytes of RAM to copy the entry to.

Flags: None

Others: None

Output:

Registers: HL = pointer to byte after the entry just copied from.
DE = DE + 9

Flags: None

Others: Nine bytes copied to the RAM from the stack entry.

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example: Copy from FPS10 to OP2.

```
             LD          HL,(FPS)    ; copy to FPS
             LD          DE,(OP2)    ; start of 9 bytes to copy to
                                     ; FPS10
;
             LD          C,(10+1)*9  ; C = offset back to FPS10,
                                     ; 11*9 bytes

;
             B_CALL      CpyStack    ; copy to OP2 from FPS10
;
```

CpyO1ToFPST, CpyO1ToFPS1, CpyO1ToFPS2, CpyO1ToFPS3, CpyO1ToFPS4, CpyO1ToFPS5, CpyO1ToFPS6, CpyO1ToFPS7, CpyO2ToFPST, CpyO2ToFPS1, CpyO2ToFPS2, CpyO2ToFPS3, CpyO2ToFPS4, CpyO3ToFPST, CpyO3ToFPS1, CpyO3ToFPS2, CpyO5ToFPS1, CpyO5ToFPS3, CpyO6ToFPST, CpyO6ToFPS2

Category: Floating Point Stack

Description: This description covers a group of routines that copies a single nine-byte OP register (OP1 – OP6), to an entry on the Floating Point Stack (FPS).
For example, CpyO1ToFPS2: OP1 is copied to (FPS2).

Inputs:

Registers: None

Flags: None

Others: OP register = 9 bytes to copy to FPS entry
For example, CpyO1ToFPS2: OP1 = nine-bytes to copy

Outputs:

Registers: DE = FPS entry following the one copied to
For example, CpyO1ToFPS2: DE = address of FPS1

HL = OP register + 9
For example, **CpyO1ToFPS2:** HL = OP1 + 9

Flags: None

Others: OP register = copy of the nine-byte FPS entry
For example, **CpyTo1FPS2:** OP1 = FPS2 entry

Registers destroyed: All

The OP register is written to.

Remarks: These routines do not allocate or deallocate entries. See entry point **CpyToStack**. See entry point **CpyTo1FPST**. See Floating Point Stack section of Chapter 2.

Example:

CpyTo1FPST, CpyTo1FPS1, CpyTo1FPS2, CpyTo1FPS3, CpyTo1FPS4, CpyTo1FPS5, CpyTo1FPS6, CpyTo1FPS7, CpyTo1FPS8, CpyTo1FPS9, CpyTo1FPS10, CpyTo1FPS11, CpyTo2FPST, CpyTo2FPS1, CpyTo2FPS2, CpyTo2FPS3, CpyTo2FPS4, CpyTo2FPS5, CpyTo2FPS6, CpyTo2FPS7, CpyTo2FPS8, CpyTo3FPST, CpyTo3FPS1, CpyTo3FPS2, CpyTo4FPST, CpyTo5FPST, CpyTo6FPST, CpyTo6FPS2, CpyTo6FPS3

Category: Floating Point Stack

Description: This description covers a group of routines that copies a single nine-byte entry from the Floating Point Stack (FPS), to one of the OP registers (OP1 – OP6). For example, CpyTo1FPS2: (FPS2) is copied to OP1.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: HL = FPS entry following one copied
For example, CpyTo1FPS2: HL = address of FPS1

DE = OP register + 9
For example, **CpyTo1FPS2:** DE = OP1 + 9

Flags: None

Others: OP register = copy of the nine-byte FPS entry
For example, **CpyTo1FPS2:** OP1 = FPS2 entry

Registers destroyed: All
The OP register is written to.

Remarks: These routines do not allocate or deallocate entries. See entry point **CpyStack**. See entry point **CpyO1ToFPST**. See Floating Point Stack section of Chapter 2.

Example:

CpyToFPST

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPST, Floating Point Stack top entry.

Input:

Registers: DE = address of nine-bytes to copy to FPST

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = (FPS), next free byte on the stack

Flags: None

Others: None

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example:

CpyToFPS1

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPS1, Floating Point Stack top entry -1.

Input:

Registers: DE = address of nine-bytes to copy to FPS1

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = pointer to FPST entry

Flags: None

Others: None

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example:

CpyToFPS2

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPS2, Floating Point Stack top entry -2.

Input:

Registers: DE = address of nine-bytes to copy to FPS2

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = pointer to FPS1 entry

Flags: None

Others: None

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example:

CpyToFPS3

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPS3, Floating Point Stack top entry -3.

Input:

Registers: DE = address of nine-bytes to copy to FPS3

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = pointer to FPS2 entry

Flags: None

Others: None

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example:

CpyToStack

Category: Floating Point Stack

Description: Copies nine-bytes to one of the systems nine-byte stacks, FPS and ES. Only the FPS (Floating Point Stack) is documented for application use. This routine should be used in the manner described in the example.

Input:

Registers: C = number of bytes from the next free byte in the stack back to the entry copying to. This will always be a multiple of nine.

HL = address of next free byte for the stack, for the FPS the address is stored in the bytes (FPS).

DE = pointer to the nine-bytes to copy to the stack.

Flags: None

Others: None

Output:

Registers: HL = pointer to byte after the entry just copied to.
DE = DE + 9

Flags: None

Others: Nine-bytes copied to the stack entry.

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example: Copy from OP2 to FPS10.

```
             LD          HL,(FPS)    ; copy to FPS
             LD          DE,(OP2)    ; start of 9 bytes to copy to
                                     ; FPS10
;
             LD          C,(10+1)*9  ; C = offset back to FPS10,
                                     ; 11*9 bytes

;
             B_CALL      CpyToStack  ; copy to FPS10
;
```

PopMCplxO1

Category: Floating Point Stack

Description: Pops a complex value from the FPS (FPS1 = real part; FPST = imaginary part). No checks are made on the data that is popped from the stack.

Inputs:

Registers: None

Flags: None

Others: FPS1 = real part of complex number

FPST = imaginary part of complex number

Outputs:

Registers: None

Flags: None

Others: OP1 contains 9 bytes of data from FPS1

OP2 contains 9 bytes of data from FPST

Registers destroyed: All

Remarks: This routine will remove 18 bytes of data from the FPS regardless of the data type.

See **PopRealO1**, **PopOP1**. See the Floating Point Stack section.

Example:

PopOP1, PopOP3, PopOP5

Category: Floating Point Stack

Description: This description covers three entry points that are similar. The description is given for PopOP1. The inputs/outputs are the same for the other two routines replacing OP1/OP2 with either OP3/OP4 or OP5/OP6.

These routines will pop either one or two floating-point numbers off of the top of the FPS. They are used to either pop a real or a complex value off of the top of the FPS without knowing in advance whether a real or a complex value is on the top of the stack.

The top entry (FPST) is popped into OP1. The sign byte of the popped value in OP1 is checked for CplxObj. If it is complex, OP1 is moved to OP2 and the new FPST is popped into OP1. If it is not complex, the floating-point number popped into OP1 is left there.

Input:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If the data type of FPST = RealObj then OP1 = FPST
If the data type of FPST = CplxObj then OP1 = FPS1,
the real part of the complex value
OP2 = FPST, the imaginary part of the complex value.

Registers destroyed: All

RAM used: OP1/OP2 or OP3/OP4 or OP5/OP6 depending on which of the routines is used.

Remarks: When using this routine make sure that the FPST entry is not a complex variable name. If it is, it will be interpreted as a complex value causing two floating-point numbers to be popped from the FPS. See **PopRealO1** and **PopMcplxO1**. See Floating Point Stack section.

Example:

PopReal

Category: Floating Point Stack

Description: Pops the last entry FPST, off of the FPS to an input RAM location. No matter what the data in FPST is only nine (9) bytes are popped off of the stack.

Inputs:

Registers: DE = pointer to RAM location to pop FPST into

Flags: None

Others: None

Outputs:

Registers: DE = DE + 9

Flags: None

Others: The nine-byte entry FPST is removed from the FPS and copied to the nine-bytes starting at address DE.

Registers destroyed: All but the ACC

Remarks: The entry is removed from the FPS shrinking the size of the FPS by nine-bytes. See the Floating Point Stack section.

Example:

PopRealO1, PopRealO2, PopRealO3, PopRealO4, PopRealO5, PopRealO6

Category: Floating Point Stack

Description: This description covers six entry points that are similar. The description is given for **PopRealO1**. The inputs/outputs are the same for the other five routines replacing OP1 with either OP2, OP3, OP4, OP5 or OP6.

Pops the last entry FPST, off of the FPS to OP1. No matter what the data in FPST is, only nine (9) bytes are popped off of the stack.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: The nine-byte entry FPST is removed from the FPS and copied to the nine-bytes starting at address OP1.

Registers destroyed:

Remarks: The entry is removed from the FPS shrinking the size of the FPS by nine-bytes. See **PopOP1**. See the Floating Point Stack section.

Example:

PushMCplxO1, PushMCplxO3

Category: Floating Point Stack

Description: **PushMCplxO1** pushes a complex value onto the FPS (OP1 = real part; OP2 = imaginary part). No checks are made on the data that is put onto the stack. **PushMCplxO3** accomplishes the same task, except inputs are OP3 and OP4.

Inputs:

Registers:

Flags: None

Others: None

Outputs: (OP1)…(OP1+8) and (OP2)…(OP2+8) contain 18 bytes of data to be pushed.

Registers: None

Flags: None

Others: FPS1 = 9 bytes from OP1

FPST = 9 bytes from OP2

Registers destroyed: All

Remarks: Memory error if not enough free RAM.

See **PushRealO1**, **PushOP1**. See the Floating Point Stack section.

Example:

PushOP1, PushOP3, PushOP5

Category: Floating Point Stack

Description: This description covers three entry points that are similar. The description is given for PushOP1. The inputs/outputs are the same for the other two routines replacing OP1/OP2 with either OP3/OP4 or OP5/OP6.

These routines will push either one or two floating-point numbers onto the FPS. It is used to either push a real or a complex value onto the FPS without knowing in advance whether a real or a complex value is being pushed onto the stack.

The sign byte of OP1 is checked for CplxObj. If it is Complex, OP1 is pushed on to the stack and the OP2 is pushed onto the stack. If it is not complex, the floating-point number in OP1 is only pushed onto the stack.

Input:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If the data type of OP1 = RealObj then FPST = OP1
If the data type of OP1 = CplxObj then FPS1 = OP1,
the real part of the complex value
FPST = OP2, the imaginary part of the complex value.

Registers destroyed: All

RAM used: None

Remarks: When using this routine make sure that the OP1 is not a complex variable name. If it is it will be interpreted as a complex value causing two floating-point numbers to be pushed onto the FPS. See **PushRealO1**, **PushMcplxO1**. See the Floating Point Stack section.

Example:

PushReal

Category: Floating Point Stack

Description: Pushes a new entry onto the FPS and copy the nine-bytes at address HL into the new entry. No checks are made on the data that is put onto the stack.

Inputs:

- **Registers:** HL = pointer to nine-bytes to push onto the FPS
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** HL = HL + 9
- **Flags:** None
- **Others:** FPST = nine-bytes at HL pushed onto the stack

Registers destroyed: All

Remarks: The previous FPST is now entry FPS1. See **PushRealO1**, **PushOP1**. See the Floating Point Stack section.

Example:

PushRealO1, PushRealO2, PushRealO3, PushRealO4, PushRealO5, PushRealO6

Category: Floating Point Stack

Description: This description covers six entry points that are similar. The description is given for PushRealO1. The inputs/outputs are the same for the other five routines replacing OP1 with either OP2, OP3, OP4, OP5 or OP6.

Pushes a new entry onto the FPS and copy the nine-bytes at OP1 into the new entry. No checks are made on the data that is put onto the stack.

Inputs:

Registers: None

Flags: None

Others: OP1 = nine-bytes to push onto the FPS

Outputs:

Registers: None

Flags: None

Others: FPST = nine-bytes at OP1 pushed onto the stack

Registers destroyed:

Remarks: The previous FPST is now entry FPS1. See **PushReal**, **PushOP1**. See the Floating Point Stack section.

Example:

5 System Routines — Graphing and Drawing

AllEq

Category: Graphing and Drawing

Description: Select or deselect all graph equations in the current graph mode.

Inputs:

Registers: ACC = 3 to select all equations in the current graph mode
= 4 to deselect all equations in the current graph mode

Flags: Current graph mode: IY + grfModeFlags = flag byte

Others: None

Outputs:

Registers: None

Flags: None

Others: All graph equations for the current mode are selected or deselected.

Registers destroyed: All

RAM used: OP1, OP2

Remarks:

Example:

BufClr

Category: Graphing and Drawing

Description: Executes the routine **GrBufClr** on a bitmap of the graph screen other than ***plotSScreen***, the system graph backup buffer.

Inputs:

Registers: HL = pointer to start of graph buffer to clear, 768 bytes

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: RAM cleared.

Registers destroyed: All

Remarks: G-T and Horizontal modes will affect how much of the buffer is cleared. In order to have the entire buffer cleared set to full screen mode.

There are two additional bit image display buffers allocated other than ***plotSScreen***, they start at addresses ***appBackUpScreen*** and ***saveSScreen***.

Example:

```
LD          HL,appBackUpScreen
B_CALL      BufClr                  ; clear backup
```

BufCpy

Category: Graphing and Drawing

Description: Executes the routine **GrBufCpy** on a bitmap of the graph screen other that ***plotSScreen***, the system graph backup buffer. The contents of the buffer are displayed.

Inputs:

Registers: HL = pointer to start of graph buffer to display, 768 bytes

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: G-T and Horizontal modes will affect how much of the buffer is displayed. In order to have the entire buffer displayed, set to full screen mode.

There are two additional bit image display buffers allocated other than ***plotSScreen***, they start at addresses ***appBackUpScreen*** and ***saveSScreen***.

Example:

```
LD        HL,appBackUpScreen
B_CALL    BufCpy                ; display backup buffer
```

CircCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a circle on the graph screen given the center and the radius, relative to the current window settings.

Inputs:

Registers: None

Flags: useFastCirc, (IY + plotFlag3) = 1 for fast circle routine that draws the circle in sections simultaneously

useFastCirc, (IY + plotFlag3) = 0 for normal circle routine that draws in a circular direction

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: FPST = radius, a floating-point number
FPS1 = Y value of center, a floating-point number
FPS2 = X value of center, a floating-point number

The center specified is with respect to the current window settings.

Outputs:

Registers: None

Flags: None

Others: Current graph, and point operation are drawn to the screen and the graph backup buffer, ***plotSScreen***.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

Remarks: If a zoom square is not done before using this routine the output circle will most likely not look circular but skewed in either the X or Y axis direction.

If useFastCirc is used, the flag must be reset by the caller.

(continued)

CircCmd *(continued)*

Example: Execute a zoom standard and then draw a circle at (0,0) with radius 3 using the alternate fast circle draw.

```
              B_CALL     ZooDefault                  ; standard window
              B_CALL     OP1Set0                     ; OP1 = 0
              RST        rPushRealO1
              RST        rPushRealO1                 ; (0,0) pushed
                                                     ; onto FPS
;
              B_CALL     OP1Set3                     ; radius is 3
              RST        rPushRealO1                 ; 3 pushed onto
                                                     ; FPS
;
              SET        useFastCirc,(IY+plotFlag3)  ; fast circle
                                                     ; routine
;
              AppOnErr   ClrFlag                     ; set up error
                                                     ; handler to clear
                                                     ; fast circle flag
;
              B_CALL     CircCmd                     ;
;
              AppOffErr                              ; remove no error

              RES        useFastCirc,(IY+plotFlag3)  ; reset flag
              RET

;
;   come here if error
;
ClrFlag:
              RES        useFastCirc,(IY+plotFlag3)  ; reset flag
;
              B_JUMP     JErrorNo                    ; continue on with
                                                     ; system error
                                                     ; handle
;
```

ClearRect

Category: Graphing and Drawing

Description: Clears a rectangular area on the screen (to Pixel off).

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is upper left corner of screen and row = 63, column = 95 is lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D >= H, E >= L.

Modifies ***saveSScreen*** RAM area.

Example:

```
LD         HL,0000h
LD         DE,3F5Fh
B_CALL     FillRect         ; Make the whole screen
                            ; black
LD         H,0
LD         L,48
LD         D,31
LD         E,95
B_CALL     ClearRect        ; Clear the screen's top
                            ; right quarter
B_CALL     GetKey           ; Get key press
B_JUMP     JForceCmdNoChar  ; Exit app
```

CLine

Category: Graphing and Drawing

Description: Draws a line between two points specified by graph coordinates. The line is plotted according to the current window settings Xmin, Xmax, Ymin, Ymax.

The points do not need to lie within the current window settings this routine will clip the line to the screen edges if any portion of the line goes through the current window settings.

This routine should only be used to draw lines in reference to the window settings.
ILine can be used to draw lines by defining points with pixel coordinates, which will be a faster draw.

Inputs:

Registers: OP4 — Y1-coordinate
OP3 — X1-coordinate
OP2 — Y2-coordinate
OP1 — X2-coordinate

Flags: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set
vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes
plotLoc, (IY + plotFlags) = 1 to draw to the display only
= 0 to draw to display and ***plotSScreen*** buffer.
bufferOnly, (IY + plotFlag3) = 1 to draw to ***plotSScreen*** buffer only.

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CLine *(continued)*

Example:

```
                                                ; Draw a line between
                                                ; the points (1.5,3)
                                                ; & (4,6):
             LD        HL,Point_1               ; point to (1.5,3) in
                                                ; ROM
             LD        DE,OP3
             LD        BC,18
             LDIR
             LD        HL,Point_2               ; point to (4,6) in
                                                ; ROM
             B_CALL    Mov9OP1OP2               ; OP1 = 4 OP2 = 6
             B_CALL    PushMCplxO1
;
             B_CALL    CLine                    ; draw the line
             RET
Point_1:
             DB        0,80h,15h,0,0,0,0,0,0    ; 1.5
             DB        0,80h,30h,0,0,0,0,0,0    ; 3
Point_2:
             DB        0,80h,40h,0,0,0,0,0,0    ; 4
             DB        0,80h,60h,0,0,0,0,0,0    ; 6
```

CLineS

Category: Graphing and Drawing

Description: Draws a line between two points specified by graph coordinates. The line is plotted according to the current window settings Xmin, Xmax, Ymin, Ymax.

The points do not need to lie within the current window settings this routine will clip the line to the screen edges if any portion of the line goes through the current window settings.

This routine should only be used to draw lines in reference to the window settings.
ILine can be used to draw lines by defining points with pixel coordinates, which will be a faster draw.

Inputs:

Registers: FPS2 — Y1-coordinate
FPS3 — X1-coordinate
FPST — Y2-coordinate
FPS1 — X2-coordinate

Flags:

plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this turn off the split screen modes.
See **ForceFullScreen**.

grfSplit, (IY + sGrFlags)	=	1 if horizontal split mode set
vertSplit, (IY + sGrFlags)	=	1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags)	=	1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CLineS *(continued)*

Example:

```
                                                 ; Draw a line between
                                                 ; the points (1.5,3)
                                                 ; & (4,6):
             LD          HL,Point_1              ; point to (1.5,3) in
                                                 ; ROM
             B_CALL      Mov9OP1OP2              ; OP1 = 1.5 OP2 = 3
             B_CALL      PushMCplxO1             ; push OP1 and then
                                                 ; OP2 onto the FPS
;
             LD          HL,Point_2              ; point to (4,6) in
                                                 ; ROM
             B_CALL      Mov9OP1OP2              ; OP1 = 4 OP2 = 6
             B_CALL      PushMCplxO1             ; push OP1 and then
                                                 ; OP2 onto the FPS
;
             B_CALL      CLineS                  ; draw the line
             RET
Point_1:
             DB          0,80h,15h,0,0,0,0,0,0   ; 1.5
             DB          0,80h,30h,0,0,0,0,0,0   ; 3
Point_2:
             DB          0,80h,40h,0,0,0,0,0,0   ; 4
             DB          0,80h,60h,0,0,0,0,0,0   ; 6
```

ClrGraphRef

Category: Graphing and Drawing

Description: Clears all graph reference flags in the symtable and the temporary symtable.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** Graph reference reset

Registers destroyed: HL, DE, BC

Remarks:

Example:

```
B_CALL     ClrGraphRef
```

CPoint

Category: Graphing and Drawing

Description: Turns on, turns off, or inverts a point in the display specified by graph coordinates. The point is plotted according to the current window settings: Xmin, Xmax, Ymin, Ymax.

This routine should only be used to draw points in reference to the window settings.
IPoint can be used to draw points by defining points with pixel coordinates, which causes a faster draw.

Inputs:

Registers: ACC = what to do
0: turn point off
1: turn point on
2: invert point

Flags: G-T and HORIZ split-screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags)	=	1 if horizontal split mode set
vertSplit, (IY + sGrFlags)	=	1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags)	=	1 to ignore split modes
plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

Others: OP1 — X Coordinate of point
OP2 — Y Coordinate of point

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CPoint *(continued)*

Example: Draw a point in the graph window at coordinates (1.5,3):

```
            LD          HL,Point_1              ; point to (1.5,3)
            B_CALL      Mov9OP1OP2              ; OP1 = 1.5 OP2 = 3
;
            LD          A,1                     ; turn on
            B_CALL      CPoint                  ; draw the point
            RET
Point_1:
            DB          0,80H,15H,0,0,0,0,0,0   ; 1.5
            DB          0,80H,30H,0,0,0,0,0,0   ; 3
```

CPointS

Category: Graphing and Drawing

Description: Turns on, turns off or inverts a point in the display specified by graph coordinates. The point is plotted according to the current window settings: Xmin, Xmax, Ymin, Ymax.

This routine should only be used to draw points in reference to the window settings.
IPoint can be used to draw points by defining points with pixel coordinates, which causes a faster draw.

Inputs:

Registers: ACC = what to do
0: turn point off
1: turn point on
2: invert point

Flags: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes.
See **ForceFullScreen**.

grfSplit, (IY + sGrFlags)	=	1 if horizontal split mode set
vertSplit, (IY + sGrFlags)	=	1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags)	=	1 to ignore split modes
plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

Others: FPS1 — X Coordinate of point
FPST — Y Coordinate of point

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CPointS *(continued)*

Example: Draw a point in the graph window at coordinates (1.5,3)

```
;
            LD          HL,Point_1              ; point to (1.5,3)
                                                ; in ROM
            B_CALL      Mov9OP1OP2              ; OP1 = 1.5 OP2 = 3
            B_CALL      PushMCplxO1             ; push OP1 and then
                                                ; OP2 onto the FPS
;
            LD          A,1                     ; turn on
            B_CALL      CPointS                 ; draw the point
            RET
Point_1:
            DB          0,80H,15H,0,0,0,0,0,0   ; 1.5
            DB          0,80H,30H,0,0,0,0,0,0   ; 3
```

DarkLine

Category: Graphing and Drawing

Description: Draws a line between two pixel points defined by their pixel coordinates.

Inputs: The graph window is defined with the lower left corner of the display to be pixel coordinate (0,0).
The system graphing routines do not normally draw in the last column and the bottom row of the screen, column 95 and row 0.
This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: X = column
Y = row

B = X-coordinate of first point — 0...94 (95) see above
C = Y-coordinate of first point — 1(0)...63
D = X-coordinate of second point — 0...94 (95)
E = Y-coordinate of second point — 1(0)...63

Flags:

fullScrnDraw, (IY + apiFlg4)	=	1 to use column 95 and row 0
plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

Others: None

Outputs:

Registers: None

Flags: None

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

Remarks: If the draw is going to the buffer then the contents of the buffer are used to draw the line and copied to the screen.
No clipping, X, Y points assumed to be defined on the screen.

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes.
See **ForceFullScreen**.

(continued)

DarkLine *(continued)*

Example:

```
; Clear the screen.
; Draw a line in the display only, between pixel coordinates (25,30)
; and (62,50):
             B_CALL     ClrLCD                  ; clear the screen;
             LD         BC,25*256+30            ; 1st point, B = 25,
                                                ; C = 30
             LD         DE,62*256+50            ; 2nd point, D = 62,
                                                ; E = 50
;
             SET        plotLoc,(IY+plotFlags)  ; display only
;
             B_CALL     DarkLine                ; draw the line
```

DarkPnt

Category: Graphing and Drawing

Description: Turns on a point in the display specified by graph coordinates. The point is plotted according to the current window settings: Xmin, Xmax, Ymin, Ymax.

This routine should only be used to draw points in reference to the window settings.
IPoint can be used to draw points by defining points with pixel coordinates, which causes a faster draw.

Inputs:

Registers: None

Flags: G-T and HORIZ split screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split-screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags)	=	1 if horizontal split mode set
vertSplit, (IY + sGrFlags)	=	1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags)	=	1 to ignore split modes
plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer.
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

Others: OP1 — X Coordinate of point
OP2 — Y Coordinate of point

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph, if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

DarkPnt *(continued)*

Example: Draw a point in the graph window at coordinates (1.5,3):

```
             LD          HL,Point_1                ; point to (1.5, 3)
                                                   ; in ROM
             B_CALL      Mov9OP1OP2                ; OP1 = 1.5 OP2 = 3
;
             B_CALL      DarkPnt                   ; draw the point
             RET
Point_1:
             DB          0,80h,15h,0,0,0,0,0,0     ; 1.5
             DB          0,80h,30h,0,0,0,0,0,0     ; 3
```

Disp

Category: Graphing and Drawing

Description: Checks if graph screen is in the display. If it is, restores the text shadow to the screen.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: shiftFlags, textFlags

Others: curRow, curCol, winTop

Registers destroyed: All

Remarks: This is intended to be used when an application uses both the home screen and the graph screen.

Using this routine allows the application to switch between the home screen and the graph screen without having to rebuild the home screen.

When switching to the graph screen, all of the text previously written to the home screen should have been also written to the text shadow.

The plotLoc flag should be set when switching to the graph screen.

Example:

DrawCirc2

Category: Graphing and Drawing

Description: Draws a circle given the center and the radius, relative to the current window settings.

The current graph screen is not put into the display by this routine.

This icircle routine is one of two available, and is the faster of the two.

Inputs:

Registers: None

Flags: plotLoc, (IY + plotFlags) = 1 to draw to the display only
plotLoc, (IY + plotFlags) = 0 to draw to display and buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to buffer only

Others: FPST = radius, a floating-point number
FPS1 = Y value of center, a floating-point number
FPS2 = X value of center, a floating-point number

The center specified is with respect to the current window settings.

Outputs:

Registers: None

Flags: None

Others: Circle is drawn either to the display, the buffer, or both.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

Remarks: If a zoom square is not done before using this routine the output circle will most likely not look circular but skewed in either the X or Y axis direction. See **CircCmd**. See Floating Point Stack section.

(continued)

DrawCirc2 *(continued)*

Example: Execute a zoom standard and then draw a circle at (0,0) with radius 3.

```
             B_CALL      ZooDefault    ; standard window
             B_CALL      PDspGrph      ; get current graph to the
                                       ; display
;
             B_CALL      OP1Set0       ; OP1 = 0
             RST         rPushRealO1
             RST         rPushRealO1   ; (0,0) pushed onto FPS
;
             B_CALL      OP1Set3       ; radius is 3
             RST         rPushRealO1   ; 3 pushed onto FPS
;
             AppOnErr    circerr       ; set up error handler
;
             B_CALL      DrawCirc2     ;
;
             AppOffErr                 ; remove no error

             RET

;
;   come here if error
;
Circerr:
;
```

DrawCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a function on it. Same as TI–83 Plus instruction DrawF.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: FPST = name of equation to evaluate and draw, with X being the independent variable.

Outputs:

Registers: None

Flags: None

Others: Current graph and function are drawn to the screen and the graph backup buffer, ***plotSScreen***.

FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw, see Error Handlers section. See section on Floating Point Stack

Example: Draw Y1 on the graph screen.

```
             LD         HL,Y1name
             B_CALL     Mov9ToOP1      ; OP1 = Y1
             B_CALL     PushRealO1     ; push Y1 into FPST
;
             B_CALL     DrawCmd        ; draw
;
             B_CALL     PopRealO1      ; clean Y1 off of FPS
;
```

DrawRectBorder

Category: Graphing and Drawing

Description: Draws a rectangular outline on the screen.

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D >= H, E >= L

Modifies ***saveSScreen*** RAM area.

Example: ;

```
        LD          HL,0000h
        LD          DE,3F5Fh
        B_CALL      DrawRectBorder     ; Draw an outline around
                                       ; the screen
        B_CALL      GetKey             ; Get key press
        B_JUMP      JForceCmdNoChar    ; Exit app
```

DrawRectBorderClear

Category: Graphing and Drawing

Description: Draws a rectangular outline on the screen and clears the area inside the outline.

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D >= H, E >= L.

Modifies ***saveSScreen*** RAM area.

Example:

```
;
          B_CALL    ClrLCDFull
          LD        H,32
          LD        L,48
          LD        D,63
          LD        E,95
          B_CALL    FillRect           ; Blacken the screen's
                                       ; lower right quarter
          B_CALL    GetKey             ; Get key press
          LD        HL,0000h
          LD        DE,3F5Fh
          B_CALL    DrawRectBorderClear ; Draw an outline
                                       ; around the screen and
                                       ; clear inside
          B_CALL    GetKey             ; Get key press
          B_JUMP    JForceCmdNoChar    ; Exit app
```

EraseRectBorder

Category: Graphing and Drawing

Description: Erases a rectangular outline on the screen (to white).

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: None

Outputs:

Registers: None

Flags None

Others: None

Registers destroyed: All

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D >= H, E >= L

Modifies ***saveSScreen*** RAM area.

Example: ;

```
        LD          HL,0000h
        LD          DE,3F5Fh
        B_CALL      DrawRectBorder    ; Draw an outline around the
                                      ; screen
        B_CALL      GetKey            ; Get key press
        B_CALL      EraseRectBorder   ; Erase an outline around
                                      ; the screen
        B_CALL      GetKey            ; Get key press
        B_JUMP      JForceCmdNoChar   ; Exit app
```

FillRect

Category: Graphing and Drawing

Description: Fills a rectangular area on the screen (to black).

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D >= H, E >= L

Modifies ***saveSScreen*** RAM area.

(continued)

FillRect *(continued)*

Example: ;

```
B_CALL      ClrLCDFull          ; Clear the whole screen
LD          HL,1C2Ch
LD          DE,2232h
B_CALL      FillRect            ; Put black square in
                                ; screen center
B_CALL      GetKey              ; Get key press
LD          H,0
LD          L,0
LD          D,63
LD          E,95
B_CALL      InvertRect          ; Turn to white square on
                                ; black background
B_CALL      GetKey              ; Get key press
LD          H,0000h
LD          D,3F5Fh
B_CALL      InvertRect          ; Return to black square on
                                ; white background
B_CALL      GetKey              ; Get key press
B_JUMP      JforceCmdNoChar     ; Exit app
```

FillRectPattern

Category: Graphing and Drawing

Description: Fills a rectangular area on the screen with a pattern.

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: RectFillPHeight = pattern's height in pixel rows (byte, 1 – 8)
RectFillPWidth = pattern's width in pixel columns (byte, 1 – 8)
RectFillPattern = one-byte for each pattern pixel row

Pattern is right justified — bit 0 is right-most pixel in pattern row. First byte is the top row of the pattern.

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is upper left corner of screen and row = 63, column = 95 is lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D   H, E   L.
You should not use the right-most column (95). This routine fails if you try to use it.

Modifies ***saveSScreen*** RAM area.

The pattern is written across the screen and is truncated at the right edge of the specified rectangle. The pattern will also be truncated at the bottom of the rectangle if needed.

(continued)

FillRectPattern *(continued)*

Example:

```
            B_CALL   ClrLCDFull          ; Clear the whole screen
            LD       A,6                 ; Pattern is 6 pixels
                                         ; high
            LD       (RectFillPHeight),
                     A
            LD       A,4                 ; Pattern is 4 pixels
                                         ; wide
            LD       (RectFillPWidth),A
            LD       HL,MyPattern        ; Copy source is the
                                         ; pattern in this code
            LD       DE,RectFillPattern  ; Copy destination is the
                                         ; pattern buffer
            LD       BC,6                ; Copy 6 bytes
            LDIR                         ; Copy pattern to pattern
                                         ; buffer
            LD       HL,1F2Fh
            LD       DE,3F5Eh            ; Coordinates of the full
                                         ; screen except last
                                         ; column
            B_CALL   FillRectPattern     ; Fill it with the
                                         ; pattern
            B_CALL   GetKey              ; Get key press
            B_JUMP   JForceCmdNoChar     ; Exit app
MyPattern:  DB       0Fh, 07h, 03h, 01h, 03h, 07h
```

GrBufClr

Category: Graphing and Drawing

Description: Clears out the graph backup buffer ***plotSScreen***.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: All 768 bytes of ***plotSScreen*** set to 0.

Registers destroyed: All

Remarks:

Example:

GrBufCpy

Category: Graphing and Drawing

Description: Copies the graph backup buffer ***plotSScreen*** to the display.

Inputs:

Registers: None

Flags: None

Others: (winBtm) should be = 8

Outputs:

Registers: None

Flags: None

Others: Graph buffer sent to display.

Registers destroyed: All

Remarks: Both vertical and horizontal split setting will affect what is copied to the screen. See **ForceFullScreen**. See **RestoreDisp**.

Example:

GrphCirc

Category: Graphing and Drawing

Description: Draws a circle on the screen given the pixel coordinates of the center and a point on the circle.

Inputs:

Registers: None

Flags: useFastCirc, (IY + plotFlag3) = 1 for fast circle routine that draws the circle in sections simultaneously

useFastCirc, (IY + plotFlag3) = 0 for normal circle routine that draws in a circular direction

plotLoc, (IY + plotFlags) = 1 to draw to the display only
plotLoc, (IY + plotFlags) = 0 to draw to display and buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to buffer only.

Others: Pixel coordinates for the center and a point on the circle. Coordinate (0,0) is the pixel in the lower left corner of the display, (x,y).

(curGX2) = x coordinate of center
(curGY2) = y coordinate of center

(curGX) = x coordinate of point on the circle
(curGY) = y coordinate of point on the circle

Outputs:

Registers: None

Flags: None

Others: Circle drawn on the display.

Registers destroyed: All

Remarks: The graph screen does not have to be displayed. The current window settings have no affect. If useFastCirc is used, the flag must be reset by the caller. See **CircCmd** and **DrawCirc2** routines.

Example:

HorizCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a horizontal line at X = OP1.

Same as TI–83 Plus instruction Horizontal.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty, and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen.

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: OP1 = X value to draw the horizontal line at.

Outputs:

Registers: None

Flags: None

Others: Current graph and the line are drawn to the screen and the graph backup buffer, ***plotSScreen***.

FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example: Draw a horizontal line at X = 3 on the graph screen.

```
             B_CALL     OP1Set3      ; OP1 = 3
;
             B_CALL     HorizCmd     ; draw the line
;
```

IBounds

Category: Graphing and Drawing

Description: Tests if a pixel coordinate lies within the graph window defined by the current split mode settings.

Inputs:

Registers: B = X pixel coordinate
C = Y pixel coordinate

Flags: The current split screen setting.

Others: None

Outputs:

Registers: None

Flags: CA = 1 if out of graph window
= 0 if in graph window

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen** and **IBoundsFull** routines for further information.

Example:

IBoundsFull

Category: Graphing and Drawing

Description: Tests if a pixel coordinate lies within the full pixel range of the display. Full screen mode should be active when using this routine. Valid values will include all 64 rows and 96 columns of the display. Normally only 63 rows and 95 columns are valid.

Inputs:

Registers: B = X pixel coordinate
C = Y pixel coordinate

Flags: The current split screen setting.

Others: None

Outputs:

Registers: None

Flags: CA = 1 if out of graph window
= 0 if in graph window

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See the **ForceFullScreen** and **IBounds** routines for further information.

ILine

Category: Graphing and Drawing

Description: Draws a line between two-pixel points defined by their pixel coordinates. The line drawn can be on, off, or inverted.

Inputs: The graph window is defined with the lower left corner of the display to be pixel coordinates (0,0).
The system graphing routines do not normally draw in the last column and the bottom row of the screen, column 95 and row 0.
This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: X = column
Y = row

B — X Coordinate of first point — 0...94 (95) see above
C — Y Coordinate of first point — 1(0)...63
D — X Coordinate of second point — 0...94 (95)
E — Y Coordinate of second point — 1(0)...63

H — Type of line to draw
0 — Set points to light, on-line
1 — Set points to dark
2 — Invert points (XOR operation)

Flags:

fullScrnDraw, (IY + apiFlg4)	=	1 to use column 95 and row 0
plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

Others: None

Outputs:

Registers: None

Flags: None

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

(continued)

ILine *(continued)*

Remarks: If the draw is going to the buffer, then the contents of the buffer are used to draw the line and copied to the screen.

G-T and HORIZ split-screen modes affect how this routine maps the coordinates specified. To avoid this. turn off the split-screen modes. See **ForceFullScreen**.

No clipping, X, Y points assumed to be defined on the screen.

Example: Erase a line in the display only, between pixel coordinates (25,30) and (62,50).

```
;
        LD      BC,25*256+30            ; 1st point, B=25,
                                        ; C=30
        LD      DE,62*256+50            ; 2nd point, D=62,
                                        ; E=50
;
        SET     plotLoc,(IY+plotFlags)  ; display only
;
        LD      H,0                     ; signal turn pixels
                                        ; off
        B_CALL  ILine                   ; draw the line
;
```

InvCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a function along the Y-axis.

The equation is evaluated with respect to X, but the value of X will range between Ymin and Ymax, and the result of each evaluation will be the X coordinate, and the Y coordinate will be the value of X. It is the same as switching X and Y, and having Y be the independent variable. But it is important to write the expression in terms of X.

Same as TI–83 Plus instruction DrawInv.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: FPST = name of equation to evaluate and draw

Outputs:

Registers: None

Flags: None

Others: Current graph and function are drawn to the screen and the graph backup buffer, ***plotSScreen***.

FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw — see Error Handlers section.

See section on Floating Point Stack.

Example: Draw Y1 on the graph screen along the Y-axis.

```
             LD          HL,Y1name
             B_CALL      Mov9ToOP1       ; OP1 = Y1
             B_CALL      PushRealO1      ; push Y1 into FPST
;
             B_CALL      InvCmd          ; draw
;
             B_CALL      PopRealO1       ; clean Y1 off of FPS
;
```

InvertRect

Category: Graphing and Drawing

Description: Inverts a rectangular area on the screen (black pixels to white; white pixels to black).

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: None

Others: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.
Inputs must satisfy conditions: D   H, E   L.

Modifies ***saveSScreen*** RAM area.

Example:

```
B_CALL    ClrLCDFull    ; Clear the screen
LD        HL,0          ; HL = upper left corner
LD        DE,3F5Fh      ; DE = lower right corner
B_CALL    InvertRect    ; Blacken entire screen
LD        HL,2030h      ; HL = middle of screen
LD        DE,3F5Fh      ; DE = lower right corner
B_CALL    InvertRect    ; Whiten lower right quadrant
B_CALL    GetKey        ; Get key press
```

IOffset

Category: Graphing and Drawing

Description: Given a pixel location, computes the offset to add to the start address of the graph buffer to the byte in the buffer containing that pixel.

Also returns the bit number in that byte for that pixel.

Also computes the row and column commands to set the LCD driver to the display byte for that pixel.

Inputs:

Registers: Pixel's row and column coordinate, (0,0) = lower left pixel of the display.
B — Column coordinate value, (0 – 95)
C — Row coordinate value, (0 – 63)

Flags: None

Others: None

Outputs:

Registers: ACC = bit that corresponds to the pixel's location in the byte it resides in is set. For example, pixel (0,0) would return with ACC = 80h, bit 7 is set.

HL = byte offset to add to the start address of the display buffer to the byte that contains the pixel's bit.

(curXRow) = row command to send to the LCD driver for that pixel.
(curY) = column command to send to the LCD driver for that pixel.

Flags: None

Others: None

Registers destroyed: All but DE

Remarks:

Example: Test if pixel (23,14) is set in the graph buffer ***plotSScreen***.

```
             LD        BC,23*256+14     ; BC = 23,14
             B_CALL    IOffset
;
             LD        DE,plotSScreen   ; start of graph buffer
             ADD       HL,DE            ; add offset to byte with
                                        ; pixel
             AND       (HL)             ; and pixels bit with byte
                                        ; in buffer
             JR        Z,Pixel_is_Off   ; jump if pixel is not set
                                        ; in buffer
```

IPoint

Category: Graphing and Drawing

Description: Executes one of the following pixel operations without displaying the current graph screen:

Turn Off
Turn On
Change (invert)
Test
Copy

Inputs: The pixels are addressed with the lower left corner of the display being pixel (0,0), (row,col)

The system does not normally draw in the last column, and the bottom row of the screen, column 95 and row 0.

This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: B = pixel row address — 0...94 (95 if full screen) see above
C = Y Coordinate of first point — 1(0)...63 (64 if full screen)
D = Function to perform

0 — Turn point off
1 — Turn point on
2 — Invert point (XOR operation)
3 — Test point
4 — Copy a point from buffer to the display

Flags: fullScrnDraw, (IY + apiFlg4) = 1 to use column 95 and row 0

plotLoc, (IY + plotFlags) = 1 to draw to the display only
plotLoc, (IY + plotFlags) = 0 to draw to display and buffer

bufferOnly, (IY + plotFlag3) = 1 to draw to buffer only

Others: None

Outputs:

Registers: None

Flags: For option 3 (test)

Z = 1 for point off
Z = 0 for point on

Others: None

Registers destroyed: None, except for option 3 (test) then all.

(continued)

IPoint *(continued)*

Remarks: The test option always tests the buffer not the display. This means that in order to use the test option the pixel tested must have been written to the graph buffer.

If the buffer is specified then the contents of the buffer are used to draw/copy, not what is in the screen.

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this turn off the split screen modes. See **ForceFullScreen**.

If G-T mode is set then this routine will turn on pixels if the display byte containing the center column of pixels is accessed. This is done to keep the center line in G-T drawn.

Example: Turn on the point specified by pixel coordinates at (5,10).

```
              LD          BC,5*256+10
              LD          D,1            ; point on cmd
;
              B_CALL      IPoint         ; turn on the point
;
```

LineCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a line defined by two points.

These points are graph coordinates with respect to the current range settings. They do not have to be points on the screen. If they are not on the screen the line will still be drawn if it passes through the screen with the current range settings.

Same as TI–83 Plus instruction Line(.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: points (X1, Y1) (X2, Y2), all are floating-point numbers
FPST = Y2 COORDINATE
FPS1 = X2 COORDINATE
FPS2 = Y1 COORDINATE
FPS3 = X1 COORDINATE

See Floating Point Stack section.

Outputs:

Registers: None

Flags: None

Others: Current graph and line are drawn to the screen and the graph backup buffer, ***plotSScreen***.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw. See Error Handlers section. See **CLine** and **ILine** to draw lines without graphing. See section on Floating Point Stack.

(continued)

LineCmd *(continued)*

Example: Draw a line on the current graph screen between (1,2) and (3,4)

```
              B_CALL     OP1Set1        ; OP1 = X1
              B_CALL     PushRealO1     ; to FPS
;
              B_CALL     Plus1          ; OP1 = OP1 + 1, = Y1
              B_CALL     PushRealO1     ; to FPS
;
              B_CALL     Plus1          ; OP1 = OP1 + 1, = X2
              B_CALL     PushRealO1     ; to FPS
;
              B_CALL     Plus1          ; OP1 = OP1 + 1, = Y2
              B_CALL     PushRealO1     ; to FPS
;
              B_CALL     LineCmd        ; copy graph to screen and
                                        ; draw line
;
```

PDspGrph

Category: Graphing and Drawing

Description: Tests if the graph of the current mode needs to be regraphed. If so, the graph is regraphed, otherwise copies ***plotSScreen*** to the display.

Inputs:

Registers: None

Flags: bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: Current graph window settings and equations

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this situation, turn off the split screen modes. See the **ForceFullScreen** routine for further information.

Example: Generate the current graph screen in the display.

```
B_CALL     PDspGrph
```

PixelTest

Category: Graphing and Drawing

Description: Tests a pixel in the graph buffer specified by pixel coordinates without copying the graph to the display.

Inputs: Pixel coordinate (0,0), (row,col), is the upper left most pixel.

FPST = Pixel coordinate’s column value, a floating-point number
(0 – 94) in full screen and horizontal split
(0 – 46) in vertical split

FPS1 = Pixel coordinate’s row value, a floating-point number
(0 – 62) in full screen
(0 – 30) in horizontal split
(0 – 50) in vertical split

See Floating Point Stack section.

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: Z = 1 for point off
Z = 0 for point on

Others: None

Registers destroyed: All

Remarks:

Example: Test on the point specified by pixel coordinates at (5,10).

```
              LD          BC,5*256+10
;
              B_CALL      PixelTest      ; test the point
;
```

PointCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and executes one of the following point operations:

Turn Off
Turn On
Change (invert)

The point is defined by graph coordinates with respect to the current range settings. The point does not need to be on the screen, and if it is not, then nothing will be drawn.

Same as TI–83 Plus instructions Pt-On(, Pt-Off(, Pt-Change(.

Inputs:

Registers: ACC = point command
0 = On
1 = Off
2 = Change

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: Bit 5 of RAM location (OP1 + 2) MUST = 0
FPST = Y coordinate of the point, a floating-point number
FPS1 = X coordinate of the point, a floating-point number

Outputs:

Registers: None

Flags: None

Others: Current graph and point operation are drawn to the screen and the graph backup buffer ***plotSScreen***.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw. See Error Handlers section. See **CPoint**, **CPointS**, and **IPoint** for point commands without graphing.

(continued)

PointCmd *(continued)*

Example: Invert point at coordinate (1.5,2)

```
              LD          HL,fp_1p5      ;
              B_CALL      Mov9ToOP1      ; OP1 = X coordinate, 1.5
              B_CALL      PushRealO1     ; to FPS
;
              B_CALL      OP1Set2        ; OP1 = Y coordinate, 2, resets
                                         ; bit 5 (OP1 + 2)
              B_CALL      PushRealO1     ; to FPS
;
              LD          A,2            ; command to invert
              B_CALL      PointCmd       ; copy graph to screen and
                                         ; invert point
;
```

PointOn

Category: Graphing and Drawing

Description: Turns on a point specified by its pixel coordinates.

Inputs: The graph window is defined with the lower left corner of the display to be pixel coordinates (0,0).
The system graphing routines do not normally draw in the last column and the bottom row of the screen, column 95 and row 0.
This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: X = column
Y = row

B — X Coordinate of first point — 0...94 (95) see above
C — Y Coordinate of first point — 1(0)...63

Flags:

fullScrnDraw, (IY + apiFlg4)	=	1 to use column 95 and row 0
plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to display and ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to ***plotSScreen*** buffer only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: D

Remarks: If the buffer is specified, then the contents of the buffer are used to draw the point.

G-T and HORIZ split-screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split-screen modes.

See **ForceFullScreen**.

Example: Turn on the point specified by pixel coordinates at (5,10):

```
              LD          BC,5*256+10
;
              B_CALL      PointOn         ; turn on the point
```

Regraph

Category: Graphing and Drawing

Description: Graphs any selected equations in the current graph mode along with any selected statplots.

Inputs:

Registers: None

Flags: smartGraph_inv, (IY + smartFlags) = 1 to defeat smart regraphing feature and force all equations to be regraphed, not just new ones.

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display.

Others: Current graph equations
Current window settings

Outputs:

Registers: None

Flags: None

Others: Graph redrawn to the display and backup buffer ***plotSScreen***, or the ***plotSScreen*** only.

Registers destroyed: All but AF

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. . To avoid this situation, turn off the split screen modes. See the **ForceFullScreen** routine for further information. Also, see the Smart Regraphing section.

Example:

```
B_CALL     Regraph
```

SetAllPlots

Category: Graphing and Drawing

Description: Selects or deselects all statplots.

Inputs:

Registers: B = 0 to unselect
B = 1 to select

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If any plot's selection stat changes then the graph is marked dirty.

Registers destroyed: All

Remarks:

Example: Turn off all stat plots.

```
LD          B,0
B_CALL      SetAllPlots
```

SetFuncM

Category: Graphing and Drawing

Description: Changes from current graph mode to function mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode, function mode flags and pointers set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example:

```
B_CALL     SetFuncM
```

SetParM

Category: Graphing and Drawing

Description: Changes from current graph mode to parametric mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode. Parametric mode flags and pointer set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example:

```
B_CALL     SetParM
```

SetPolM

Category: Graphing and Drawing

Description: Changes from current graph mode to polar mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode, polar mode flags and pointers set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example:

```
B_CALL    SetPolM
```

SetSeqM

Category: Graphing and Drawing

Description: Changes from current graph mode to sequence mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode, sequence mode flags and pointers set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example:

```
B_CALL    SetSeqM
```

SetTblGraphDraw

Category: Graphing and Drawing

Description: Sets the current graph to dirty to cause a complete regraph the next time the graph needs to be displayed. Also marks the table of values as dirty, unless a graph is currently being graphed.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: smartGraph_inv, (IY + smartFlags) is set to invalidate smart graph
reTable, (IY + tblFlags) is set to dirty the table, if not graphing
graphDraw, (IY + graphFlags) is set to dirty the graph

Others: None

Registers destroyed: None

Remarks:

Example:

TanLnF

Category: Graphing and Drawing

Description: Draws the tangent line for given equation at a given point.

The equation itself is not drawn only the tangent line.

The graph screen is not displayed — it is assumed to be displayed already.

Inputs:

Registers: None

Flags: None

Others: FPST = equation name, X is the independent variable
Variable X = X coordinate of point
OP1 = Y coordinate of point, a floating-point number

Window settings for the current graph are used

Outputs:

Registers: None

Flags: None

Others: Tangent line drawn to the display.

Equation name removed from the FPS.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: See section on the Floating Point Stack in Chapter 2.

Example:

UCLineS

Category: Graphing and Drawing

Description: Draws a WHITE line between two points specified by graph coordinates.

The line is plotted according to the current window settings Xmin, Xmax, Ymin, Ymax.

The points do not need to lie within the current window settings. This routine will clip the line to the screen edges if any portion of the line goes through the current window settings.

This routine should only be used to draw lines in reference to the window settings.

ILine can be used to draw lines by defining points with pixel coordinates, which will be a faster draw.

Inputs:

Registers: FPS2 — Y1 Coordinate
FPS3 — X1 Coordinate
FPS1 — Y2 Coordinate
FPST — X2 Coordinate

Flags:

plotLoc, (IY + plotFlags)	=	1 to draw to the display only
	=	0 to draw to the display and the ***plotSScreen*** buffer
bufferOnly, (IY + plotFlag3)	=	1 to draw to the ***plotSScreen*** buffer only

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See the **ForceFullScreen** routine.

grfSplit, (IY + sGrFlags)	=	1 if horizontal split mode set
vertSplit, (IY + sGrFlags)	=	1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags)	=	1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

Example: See the **CLineS** routine.

UnLineCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and erases a line defined by two points.

These points are graph coordinates with respect to the current range settings. They do not have to be points on the screen. If they are not on the screen, the line will still be drawn if it passes through the screen with the current range settings.

Same as the TI–83 Plus instruction Line(with the last argument = 0 for unline.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: Points (X1,Y1) (X2,Y2), all are floating-point numbers
FPST = Y2 COORDINATE
FPS1 = X2 COORDINATE
FPS2 = Y1 COORDINATE
FPS3 = X1 COORDINATE

See the Floating Point Stack section.

Outputs:

Registers: None

Flags: None

Others: Current graph and line are drawn to the screen and the graph backup buffer, ***plotSScreen***.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw — see the Error Handlers section. See **UCLineS** to draw lines without graphing. See the Floating Point Stack section.

Example: See **LineCmd**.

VertCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a vertical line at Y = OP1. Same as TI–83 Plus instruction Vertical.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: OP1 = Y value to draw the vertical line at

Outputs:

Registers: None

Flags: None

Others: Current graph and the line are drawn to the screen and the graph backup buffer, ***plotSScreen***.

FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example: Draw a vertical line at Y = 3 on the graph screen.

```
             B_CALL    OP1Set3       ; OP1 = 3
;
             B_CALL    VertCmd       ; draw the line
;
```

VtoWHLDE

Category: Graphing and Drawing

Description: In the current graph window converts a pixel point to its corresponding X and Y values, floating-point numbers.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: B = X pixel value, 0 – 94, 0 = left most pixel column
C = Y pixel value, 1 – 62, 1 = next to last row of pixels from bottom

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point value representing X pixel coordinate
OP4 = floating-point value representing Y pixel coordinate

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4

Remarks: The bottom row of pixels is not used. Graph is up to date.

Example:

Xftol

Category: Graphing and Drawing

Description: In the current graph window, converts a floating-point value to an X pixel coordinate.

This is used by the graphing routines to plot points in the current graph.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: HL = pointer to floating-point number representing the X coordinate

Flags: None

Others: None

Outputs:

Registers: ACC = X pixel value, 0 – 94, 0 = left most pixel column

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: The right most column is not used. Graph is up to date.

Example:

Xitof

Category: Graphing and Drawing

Description: In the current graph window converts an X pixel coordinate to the floating-point value of X for that pixel.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: ACC = X pixel value, 0 – 94, 0 = left most pixel column
HL = pointer to location to return floating-point value

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Floating-point value representing X pixel coordinate returned at input HL to HL + 8.

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: The bottom row of pixels is not used. Graph is up to date.

Example:

YftoI

Category: Graphing and Drawing

Description: In the current graph window, converts a floating-point value to an Y pixel coordinate.

This is used by the graphing routines to plot points in the current graph.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: HL = pointer to floating-point number representing the Y coordinate

Flags: None

Others: None

Outputs:

Registers: ACC = Y pixel value, 1 – 62, 1 = next to last row of pixels from bottom

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: The bottom row of pixels is not used. Graph is up to date.

Example:

ZmDecml

Category: Graphing and Drawing

Description: Changes the window settings such that (0,0) is in the center of the display and ≅X and ≅Y = 0.1. See the ZDecimal selection in the TI–83 Plus ZOOM menu.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** Current window settings.

Outputs:

- **Registers:** None
- **Flags:** graphDraw, (IY + graphFlags) = 1, dirty the graph
- **Others:** Current window settings are moved to ZPrevious. New windows settings set to X: -4.7 to 4.7, Y: -3.1 to 3.1

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmFit

Category: Graphing and Drawing

Description: Changes the window settings such that the minimum and maximum Y value for all selected functions fit in the graph window.

The same ZoomFit under the ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.

New windows settings set so that all selected functions Y values fit in the display when regraphed.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmInt

Category: Graphing and Drawing

Description: Changes the window settings such that ΔX and ΔY = 1.0, given the coordinates in the center of the screen. The coordinates of the center of the screen are rounded to the closest integer before the window range is set. See the ZInteger selection in the TI–83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: OP1 = X coordinate of new center of the screen, floating-point number
OP5 = Y coordinate of new center of the screen, floating-point number

Current window settings.

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.

New windows settings set.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmPrev

Category: Graphing and Drawing

Description: Changes the window settings back to the settings before the last zoom command was executed, if one was. See the ZPrevious selection in TI–83 Plus ZOOM/MEMORY menu.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** graphDraw, (IY + graphFlags) = 1, dirty the graph
- **Others:** If ZPrevious values exist they are copied to the current window settings.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmSquare

Category: Graphing and Drawing

Description: Changes the window settings in either the X or Y direction such that ΔX = ΔY. Doing this operation will make a circle drawn have the shape of a circle instead of an ellipse. See the ZSquare selection in the TI–83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.

New windows settings set.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmStats

Category: Graphing and Drawing

Description: Changes the window settings such that all selected Statplots will be visible in the graph window. See the ZoomStat in the TI–83 Plus ZOOM menu.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** graphDraw, (IY + graphFlags) = 1, dirty the graph
- **Others:** Current window settings are moved to ZPrevious.

  New windows settings set.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmTrig

Category: Graphing and Drawing

Description: Changes the window settings to preset values that are appropriate for trigonometrical function graphs. See the ZTrig selection in the TI–83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.
New windows settings set to X: -(47/24) * pi, Y: (47/24) * pi

If the current angle mode setting is radians, then those values are used. If the current angle mode setting is degrees, then those values are converted from radians to degrees.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmUsr

Category: Graphing and Drawing

Description: Recalls the window settings stored by the last ZoomSto command. See the ZoomRcl selection in the TI–83 Plus ZOOM/MEMORY menu.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.

New windows settings set.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZooDefault

Category: Graphing and Drawing

Description: Changes the window settings back to the default settings of (-10,10) for both X and Y ranges.

The same ZStandard under the ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: New windows settings set to X: -10 to 10, Y: -10 to 10

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

6 System Routines — Interrupt

DivHLBy10

Category: Interrupt

Description: Divides HL by 10.

Inputs:

Registers: HL = dividend

Flags: None

Others: None

Outputs:

Registers: HL = Int(HL/10)
A = mod(HL/10)

Flags: None

Others: None

Registers destroyed: None

Remarks: None

Example:

DivHLByA

Category: Interrupt

Description: Divides HL by accumulator.

Inputs:

Registers: HL = dividend
A = divisor

Flags: None

Others: None

Outputs:

Registers: HL = Int(HL/A)
A = mod(HL/A) (remainder)

Flags: None

Others: None

Registers destroyed: None

Remarks: None

Example:

7 System Routines — IO

AppGetCalc

Category: IO

Description: Executes the basic **GetCalc** command to retrieve a variable from another TI–83 Plus or a TI–83.

Inputs:

Registers: OP1 = name of variable to attempt to retrieve

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: comFailed, (IY + getSendFlg) = 0 if variable received
comFailed, (IY + getSendFlg) = 1 if variable not received
Variable updated or created if received

Registers destroyed: All

Remarks: Variables can be received from both an TI–83 Plus and a TI–83.

Example:

```
;
        B_CALL    AnsName                   ; OP1 = Ans
                                            ; variable name
        B_CALL    AppGetCalc                ; attempt to get
                                            ; Ans
        BIT       comFailed,(IY+getSendFlg) ; did it work?
        JP        NZ,GetFailed              ; jump if no
```

AppGetCbl

Category: IO

Description: Executes the basic **GetCbl** command to retrieve data from a CBL/CBL2 or CBR device.

Inputs:

Registers: OP1 = name of variable to attempt to retrieve

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: comFailed, (IY + getSendFlg) = 0 if variable received
comFailed, (IY + getSendFlg) = 1 if variable not received
Variable updated or created if received

Registers destroyed: All

Remarks:

Example:

```
              LD          HL,L1name
              RST         rMov9ToOP1                ; OP1 = L1 variable
                                                    ; name
              B_CALL      AppGetCbl                 ; attempt to get
                                                    ; data
              BIT         comFailed,(IY+getSendFlg) ; did it work?
              JP          NZ,GetFailed              ; jump if no
L1name:       DB          ListObj,tVarLst,tL1,0,0
```

Rec1stByte

Category: IO

Description: Polls the link port for activity until either a byte is received, the [ON] key is pressed, or an error occurred during communications. The cursor is turned on for updates.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: ACC = byte received if one

Flags: None

Others: Error will be generated if communications fail.
An error is also generated if the [ON] key is pressed.

Registers destroyed: All

RAM used:

Remarks: APD can occur while waiting for link activity. See Chapter 2 for Error Handlers and Link Port. See entry points **Rec1stByteNC**, **RecAByte**, and **SendAByte**.

Example: See Chapter 2.

Rec1stByteNC

Category: IO

Description: Polls the link port for activity until either a byte is received, the [ON] key is pressed, or an error occurred during communications. The cursor is not turned on for updates.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: ACC = byte received if one

Flags: None

Others: Error will be generated if communications fail. An error is also generated if the [ON] key is pressed.

Registers destroyed: All

RAM used:

Remarks: APD can occur while waiting for link activity. See Chapter 2 for Error Handlers and Link Port. See entry points **Rec1stByte**, **RecAByte**, and **SendAByte**.

Example: See Chapter 2.

RecAByteIO

Category: IO

Description: Attempts to read a byte of data over the link port.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: ACC = byte if successful

Flags: None

Others: None

Registers destroyed: All

Remarks: If no link activity is detected within about 1.1 seconds, a system error is generated. See entry points **Rec1stByte**, **Rec1stByteNC**, and **SendAByte**.

Example: See Chapter 2.

SendAByte

Category: IO

Description: Attempts to send a byte of data over the link port.

Inputs:

Registers: ACC = byte to send.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: If no link activity is detected within about 1.1 seconds, a system error is generated. See entry points **Rec1stByte**, **Rec1stByteNC**, and **RecAByte**.

Example: See Chapter 2.

SendVarCmd

Category: IO

Description: Attempts to send a variable whose name is in OP1 to CBL/CBL2 or CBR device.

Inputs:

Registers: None

Flags: None

Others: OP1 contains name of variable to send.

Outputs:

Registers: None

Flags: ComFailed, (IY+getSendFlg) = 1 if send failed.

ComFailed, (IY+getSendFlg) = 0 if successful.

Others: OP1 is left intact.

Registers destroyed: All

Remarks: No system error is generated if link is not successful.

Example: Check status of Channel 1 on CBL2.

```
             LD          HL,L1name
             RST         rMov9ToOP1                 ; OP1 = L1 name
             RST         rFindSym                   ; Look up L1
             JR          C, CreateIt                ; jump if it doesn't
                                                    ; exist
             B_CALL      DelVarArc                  ; delete L1
CreateIt:
             LD          HL, 3                      ; 3 elements in L1
             B_CALL      CreateRList                ; L1 created
             INC         DE
             INC         DE                         ; move past size bytes
             LD          HL, Command8
             LD          BC, 27
             LDIR                                   ; L1 = {8,1,0}
             B_CALL      Op4ToOp1                   ; OP1 = L1 name
             B_CALL      SendVarCmd                 ; send L1 to CBL2
             BIT         comFailed,(IY+getSendFlg)  ; did it work?
             JP          NZ, SendFailed             ; no, jump
             B_CALL      AppGetCbl                  ; attempt to get L1
             BIT         comFailed,(IY+getSendFlg)  ; did it work?
             JP          NZ,GetFailed               ; jump if no
L1name:
             DB          ListObj,tVarLst,tL1,0,0
Command8:
             DB          00h, 80h, 80h, 00h, 00h,   00h, 00h, 00h, 00h
             DB          00h, 80h, 10h, 00h, 00h,   00h, 00h, 00h, 00h
             DB          00h, 80h, 00h, 00h, 00h,   00h, 00h, 00h, 00h
```

8 System Routines — Keyboard

ApdSetup

Category: Keyboard

Description: Resets the Automatic Power Down timer.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: (apdTimer)

Flags: None

Others: None

Registers destroyed: HL

Remarks:

Example:

CanAlphIns

Category: Keyboard

Description: Cancels alpha, alpha lock, and insert mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: textInsMode (In textFlags) and shiftALock (In shiftFlags) cleared
shiftAlpha (In shiftFlags) and shiftLwrAlph (In shiftFlags) may also be cleared depends on flag shiftKeepAlph (In shiftFlags)

Others: None

Registers destroyed: None

Remarks:

Example:

```
B_CALL      CanAlphIns
```

GetCSC

Category: Keyboard

Description: Gets and clears keyboard scan code. This routine should be used to read the keyboard only when an app does not care about second keys or alpha keys or pull down menus.

This routine only returns to the application which physical key on the keyboard was last pressed.

Inputs:

Registers: None

Flags: None

Others: None

Outputs: This routine does not wait for a key press to return back to the app. Key presses are detected in the interrupt handler, this routine returns that value. A 0 value is returned if no key has been pressed since the previous call to **GetCSC**.

Registers: A = (kbdScanCode) value

Flags: None

Others: (kbdScanCode) set to 0. kbdSCR flag reset.

Registers destroyed: AF, HL

(continued)

GetCSC *(continued)*

Remarks: No silent link activity will be detected if this routine is used to poll for keys. Below are the scan code equates.

```
;
;
skDown      equ  01h    skCos       equ  1Eh
skLeft      equ  02h    skPrgm      equ  1Fh
skRight     equ  03h    skStat      equ  20h
skUp        equ  04h    sk0         equ  21h
skEnter     equ  09h    sk1         equ  22h
skAdd       equ  0Ah    sk4         equ  23h
skSub       equ  0Bh    sk7         equ  24h
skMul       equ  0Ch    skComma     equ  25h
skDiv       equ  0Dh    skSin       equ  26h
skPower     equ  0Eh    skMatrix    equ  27h
skClear     equ  0Fh    skGraphvar  equ  28h
skChs       equ  11h    skStore     equ  2Ah
sk3         equ  12h    skLn        equ  2Bh
sk6         equ  13h    skLog       equ  2Ch
sk9         equ  14h    skSquare    equ  2Dh
skRParen    equ  15h    skRecip     equ  2Eh
skTan       equ  16h    skMath      equ  2Fh
skVars      equ  17h    skAlpha     equ  30h
skDecPnt    equ  19h    skGraph     equ  31h
sk2         equ  1Ah    skTrace     equ  32h
sk5         equ  1Bh    skZoom      equ  33h
sk8         equ  1Ch    skWindow    equ  34h
skLParen    equ  1Dh    skYEqu      equ  35h
                        sk2nd       equ  36h
                        skMode      equ  37h
                        skDel       equ  38h
```

(continued)

GetCSC *(continued)*

Example: Poll for the 2nd key.

```
             EI                          ; enable interrupts
;
                                         ; the halt is optional, this
                                         ; will help save battery life.
;
                                         ; you can still use GetCSC at
                                         ; anytime without the halt.
;
sleep:
             HALT                        ; sleep in low power for a
                                         ; little
;
             B_CALL      GetCSC          ; check for a scan code
             CP          ksk2nd          ; 2nd key ?
             JR          NZ, sleep       ; jump if no
;
```

GetKey

Category: Keyboard

Description: Keyboard entry routine that will return second keys, alpha keys — both capital and lower case, the on key, APD, and link communication. Contrast adjustment is also handled by this routine.

When called, this routine scans for keys until one is pressed, or an APD occurs, or the unit is turned off, or link activity is detected.

Inputs:

Registers: None

Flags:

indicOnly, (IY + indicFlags) = MUST BE RESET, otherwise no key presses will be detected.

indicRun, (IY + indicFlags) = 1 to show the run indicator while waiting for a key press.

apdAble, (IY + apdFlags) = 1 if APD is enabled
= 0 if APD is disabled

lwrCaseActive, (IY + appLwrCaseFlag) = 1 for the key sequence [alpha] [alpha] to access lower case alpha key presses
= 0 for normal alpha key operation

Others: None

Outputs:

Registers: ACC = key code, 0 = ON key

See TI83plus.inc file.

Flags: onInterrupt, (IY + onFlags) = 1 if ON key, this should be reset

Others:

APD: If the auto power down occurs the application will not be notified. Once the unit is turned back on control is returned to the GetKey routine.

OFF: If the unit is turned off the application is put away. When the unit is turned back on the home screen will be in control.

Link Activity: When link activity is initiated, control is given to the silent link handler. If the communication is from the GRAPH LINK, the application will be shut down in most cases. The only exception is getting screen snap shots, in that case the application is not shut down. After the screen is sent control returns to GetKey.

Registers destroyed: DE, HL

Remarks: If APD is disabled, it should be re-enabled before exiting the application. If lower case is enabled, it should be disabled upon exiting the application.

Example:

9 System Routines — List

AdrLEle

Category: List

Description: Computes the RAM address of an element of a list.

Inputs:

Registers: DE = pointer to start of list's data storage, output of **FindSym**
HL = element number in list to compute address of. List element number one is checked for real or complex data type to determine if the list is real or complex.

Flags: None

Others: None

Outputs:

Registers: HL = pointer in RAM to the start of the desired element

Flags: None

Others: None

Registers destroyed: AF, BC

Remarks: This routine does not check to see if the element's address requested is within the current size of the list.
Do not use this routine on a list that does not have element number 1 initialized.

Example: Compute the address of element number 23 of list L1.

```
             LD          HL,L1Name
             RST         rMov9ToOP1      ; OP1 = L1 name
             B_CALL      FindSym         ; look it up
             JP          C,UndefinedL1   ; jump out if L1 is not
                                         ; defined;
             LD          A,B             ; if b<>0 then L1 is archived
                                         ; in Flash ROM
             OR          A
             JP          NZ,ArchivedL1   ; jump if not in RAM
; DE = pointer to start of list data storage;
             LD          HL,23d          ; element's address desired
             B_CALL      AdrLEle         ; RET HL = pointer to 23rd
                                         ; element
             RET
L1Name:
             DB          ListObj,tVarLst,tL1,0,0
```

ConvDim

Category: List

Description: Converts floating-point value in OP1 to a two-byte hex value — make sure valid matrix or vec dimension. Less than 100 is valid dimension

Inputs:

Registers: None

Flags: None

Others: OP1 = FP number

Outputs:

Registers: A = LSB HEX VALUE, DE = ENTIRE HEX VALUE

Flags: None

Others: None

Registers destroyed: A, BC, DE, HL, OP1

Remarks: Error if negative, non-integer, or greater than 99.

Example:

```
B_CALL       ConvDim
```

ConvLcToLr

Category: List

Description: Converts an existing complex list variable to a real list variable.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of complex list variable to convert

Outputs:

Registers: None

Flags: None

Others: Error if the list was undefined.
OP1 = name of list with type set to ListObj. The imaginary part of each element is deleted and the data storage area is compressed. All symbol table pointers are updated.

Registers destroyed: All

Remarks: Do not use this routine if the input list is already a real list.

Example:

ConvLrToLc

Category: List

Description: Converts an existing real list variable to a complex list variable.

Inputs:

Registers: DE = pointer to data storage for list, output of **ChkFindSym**

Flags: None

Others: FPST = name of variable converted, see Floating Point Stack

Outputs:

Registers: DE = pointer to data storage of converted list

Flags: None

Others: Error if not enough free RAM to convert to complex.
Each element of the list is converted to a complex number with a 0 imaginary part.
FPST = name of variable converted, see Floating Point Stack.
All symbol table pointers are updated.

Registers destroyed: All

Remarks: Do not use this routine if the input list is already a complex list.

Example: Convert real list L1 to a complex list.

```
             LD          HL,L1Name
             RST         rMov9ToOP1       ; OP1 = L1 name
             B_CALL      PushRealO1       ; FPST = name of list
;
             B_CALL      FindSym          ; look it up, DE = pointer
                                          ; to data storage
             JP          C,convertError   ; jump out if L1 is not
                                          ; defined
;
             AppOnErr    convertError     ; install error handler in
                                          ; case not enough RAM
;
             B_CALL      ConvLrToLc       ; attempt to convert to
                                          ; complex
;
             AppOffErr                    ; remove error handler,
                                          ; successful
;
convertError:
             B_CALL      PopRealO1        ; remove name of list from
                                          ; FPST
;
             RET
;

L1Name:
             DB          ListObj,tVarLst,tL1,0,0
```

DelListEl

Category: List

Description: Deletes one or more elements from an existing list, residing in RAM.

Input:

Registers: A = ListObj if the list has real elements
= CListObj if the list has complex elements
DE = pointer to start of list's data storage, output of **FindSym**
HL = number of elements to delete
BC = element number to start deleting at

Flags: None

Others: None

Output:

Registers: HL = pointer to start of list's data storage, output of **FindSym**
DE = new dimension of the list.

Flags: None

Others: (insDelPtr) = pointer to start of the list

Registers destroyed: All

Remarks: DO NOT ATTEMPT ON AN ARCHIVED LIST. The size bytes of the list are adjusted. All pointers in the symbol table are updated

Example: Delete three elements from list L1 starting with element number two.

```
             LD        HL,L1Name
             RST       rMov9ToOP1        ; OP1 = L1 name
             B_CALL    FindSym           ; look it up, DE = pointer
                                         ; to data storage
             JP        C,UndefinedL1     ; jump out if L1 is not
                                         ; defined
;
             LD        C,A               ; save type
             LD        A,B               ; get archived status
             OR        A                 ; in RAM or archived
             JP        NZ,errArchived    ; cannot insert if archived
;
             LD        A,C               ; get type back
             AND       1Fh               ; mask type of list in ACC
             LD        HL,3              ; want to delete 3 elements
             LD        BC,2              ; delete 2nd element on
;
             B_CALL    DelListEl         ; delete elements
;

L1Name:
             DB        ListObj,tVarLst,tL1,0,0
```

Find_Parse_Formula

Category: List

Description: Checks if a list variable has a formula attached to it and parses the formula and stores it back into the list data.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of list

Outputs:

Registers: None

Flags: None

Others: If no error, then the list values are updated.

Registers destroyed: All

Remarks: If no formula is attached, nothing is done to the existing list data.

Any error that occurs during the parsing of the formula will cause an error screen to be displayed if no error handler is invoked.

If the resulting type from the formula parsing is not a list, this will also generate an error.

See Error Handlers.

Example:

GetLToOP1

Category: List

Description: Copies a list element to OP1 or OP1/OP2.

Inputs:

Registers: HL = element number to copy
DE = pointer to start of list's data storage

Flags: None

Others: None

Outputs:

Registers: HL = pointer to next element in the list

Flags: None

Others: OP1 = list element if a real list
OP1/OP2 = list element if a complex list

Registers destroyed: All

Remarks:

Example:

IncLstSize

Category: List

Description: Increments the size of an existing list in RAM by adding one element at the end of the list. No value is stored in the new element.

Input:

Registers: A = ListObj if the list has real elements
= CListObj if the list has complex elements
DE = pointer to start of list's data storage, output of **FindSym**

Flags: None

Others: None

Output:

Registers: DE = intact
HL = new dimension of the list

Flags: None

Others: (insDelPtr) = pointer to start of the list

Registers destroyed: All

Remarks: DO NOT ATTEMPT ON AN ARCHIVED LIST. A memory error will be generated if insufficient RAM. The size bytes of the list are adjusted. All pointers in the symbol table are updated.

(continued)

IncLstSize *(continued)*

Example: Increment real list L1 and store a 3 in the new element.

```
             LD        HL,L1Name
             RST       rMov9ToOP1       ; OP1 = L1 name
             B_CALL    FindSym          ; look it up, DE = pointer to
                                        ; data storage
             JP        C,UndefinedL1    ; jump out if L1 is not
                                        ; defined
;
             LD        A,B              ; get archived status
             OR        A                ; in RAM or archived
             JP        NZ,errArchived   ; cannot insert if archived
;
             LD        A,ListObj        ; type of list in ACC
;
             B_CALL    IncLstSize       ; insert element at end
;
             PUSH      DE               ; save pointer to list
             PUSH      HL               ; save last element #, just
                                        ; inserted
;                                       ;
             B_CALL    OP1Set3          ; OP1 = 3
;
             POP       HL
             POP       DE               ; restore
;
             B_CALL    PutToL           ; store OP1 to inserted
                                        ; element
;
L1Name:
             DB        ListObj,tVarLst,tL1,0,0
```

InsertList

Category: List

Description: Inserts one or more elements into an existing list, residing in RAM.

Inputs:

Registers: A = ListObj if the list has real elements
A = CListObj if the list has complex elements
DE = pointer to start of list's data storage, output of **FindSym**
HL = number of elements to insert
BC = List element number to insert after

Flags: CA = 0 to set new elements to 0
CA = 1 to set new elements to 1

Others: None

Outputs:

Registers: DE = intact
HL = new dimension of the list.

Flags: None

Others: (insDelPtr) = pointer to start of the list

Registers destroyed: All

Remarks: DO NOT ATTEMPT ON AN ARCHIVED LIST. A memory error will be generated if insufficient RAM. The size bytes of the list are adjusted. All pointers in the symbol table are updated

(continued)

InsertList *(continued)*

Example: Insert three new elements in list L1 after its second element, set the new elements to 0’s.

```
             LD        HL,L1Name
             RST       rMov9ToOP1      ; OP1 = L1 name
             B_CALL    FindSym         ; look it up, DE = pointer to
                                       ; data storage
             JP        C,UndefinedL1   ; jump out if L1 is not
                                       ; defined
;
             LD        C,A             ; save type
             LD        A,B             ; get archived status
             OR        A               ; in RAM or archived
             JP        NZ,errArchived  ; cannot insert if archived
;
             LD        A,C             ; get type back
             AND       1Fh             ; mask type of list in ACC
             LD        HL,3            ; want to insert 3 elements
             LD        BC,2            ; insert after 2nd element
             OR        A               ; CA = 0, to set new elements
                                       ; to 0
;
             B_CALL    InsertList      ; insert elements
;
L1Name:
             DB        ListObj,tVarLst,tL1,0,0
```

PutToL

Category: List

Description: Stores either a floating-point number or a complex pair to an existing element of a list.

Inputs:

Registers: HL = element number to store to
There is no check to see if this element is valid for the list.

DE = pointer to the start of the list's data area, output of **FindSym**

Flags: None

Others: None

OP1 = floating-point number set to RealObj to store to a real list

OP1/OP2 = floating-point numbers representing a complex number to store to a complex list

There are no checks made that the correct data type is being stored to the correct type of list (real/complex).

Outputs:

Registers: DE = pointer to next element in the list

Flags: None

Others: OP1/OP2 = intact

Registers destroyed: All

Remarks:

Example:

```
; Look up L1 and store 1 to element 30.
             LD          HL,L1name
             B_CALL      Mov9ToOP1    ; OP1 = name
;
             B_CALL      FindSym      ; look up
             RET         C            ; return if undefined
;
                                      ; DE = pointer to data area of list
;
             PUSH        DE           ; save pointer
             B_CALL      OP1Set1      ; OP1 = 1
;
             POP         DE
             LD          HL,30d       ; element to store to
             B_CALL      PutToL       ; store 1 to element 30
             RET
;
L1name:
             DB          ListObj,tVarLst,tL1,0
```

10 System Routines — Math

(continued)

AbsO1O2Cp

Category: Math

Description: Compares Abs(OP1) to Abs(OP2).

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point
OP2 = floating point

Outputs:

Registers: None

Flags: Z = 1: Abs(OP1) = Abs(OP2)
Z = 0, CA = 1: Abs(OP1) < Abs(OP2)
Z = 0, CA = 0: Abs(OP1) >= Abs(OP2)

Others: OP1 = Abs(OP1)
OP2 = Abs(OP2)

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

AbsO1PAbsO2

Category: Math

Description: Calculates the sum of the absolute values of the floating point in OP1 plus the floating point in OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point
OP2 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value (Abs(OP1) + Abs(OP2))

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

ACos

Category: Math

Description: Computes the inverse cosine of a floating point. The answer will not go complex.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks: Domain error if answer is complex.

Example:

ACosH

Category: Math

Description: Computes inverse hyperbolic cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse hyperbolic cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks: Domain error if OP1 is negative.

Example:

ACosRad

Category: Math

Description: Computes the inverse cosine of a floating point and force radian mode.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

Angle

Category: Math

Description: Calculates a polar complex angle from a rectangular complex.

Input:

Registers: None

Flags: None

Others: OP1 = real representing complex X
OP2 = real representing complex Y

Outputs:

Registers: None

Flags: None

Others: OP1 = real representing complex angle

Registers destroyed: All

Remarks: OP1 is not modified.

Example:

ASin

Category: Math

Description: Computes the inverse sine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse sine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ASinH

Category: Math

Description: Computes the inverse hyperbolic sine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse sine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ASinRad

Category: Math

Description: Computes the inverse sine of a floating point and force radian mode.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating point

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = inverse sine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ATan

Category: Math

Description: Computes the inverse tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse tangent (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ATan2

Category: Math

Description: Returns the angle portion of a complex number in rectangular form.

Inputs:

Registers: None

Flags: trigDeg, (IY + trigFlags) = 1 to return angle in degrees
= 0 to return angle in radians

Others: OP1 = imaginary part of complex number, floating-point number
OP2 = real part of complex number, floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = the angle portion of the polar form of the input rectangular complex number.

Registers destroyed: All

RAM used: OP1 – OP5

Remarks:

Example:

ATan2Rad

Category: Math

Description: Returns the angle portion of a complex number in rectangular form — forced to return the angle in radians no matter what the current system angle settings are.

Inputs:

Registers: None

Flags: None

Others: OP1 = imaginary part of complex number, floating-point number
OP2 = real part of complex number, floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = the angle portion of the polar form of the input rectangular complex number.

Registers destroyed: All

RAM used: OP1 – OP5

Remarks:

Example:

ATanH

Category: Math

Description: Computes the inverse hyperbolic tangent of a floating point.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating point

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = inverse hyperbolic tangent (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks: Initial input rules:

- If floating point = 0, then output = 0.
- If the absolute value of input is greater than 1 then domain error.
- FOR |OP1| <.7 Use Cordic; otherwise, use Logs.

Example:

ATanRad

Category: Math

Description: Computes the inverse tangent of a floating point and forces radian mode.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse tangent (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

CAbs

Category: Math

Description: Computes the magnitude of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result, a real number

Registers destroyed: All

RAM used: OP1 – OP4

Remarks: SqRoot(OP1^2 + OP2^2).

Example:

```
B_CALL    CAbs
```

CAdd

Category:	Math
Description:	Addition of two complex numbers.
Inputs:	
Registers:	None
Flags:	None
Others:	OP1/OP2 = second argument FPS1/FPST = first argument
Outputs:	
Registers:	None
Flags:	None
Others:	OP1/OP2 = complex result (first argument) + (second Argument)
Registers destroyed:	All
RAM used:	OP1 – OP2
Remarks:	First argument is removed from the FPS (Floating Point Stack).
Example:	See **CSub**.

CDiv

Category: Math

Description: Division of two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) / (second Argument)

Registers destroyed: All

RAM used: OP1 – OP4

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

CDivByReal

Category: Math

Description: Divides a complex number by a real number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number
OP3 = floating point real number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result, OP1/OP2 / OP3
OP3 = intact

Registers destroyed: All

RAM used: OP1 – OP4

Remarks:

Example:

```
B_CALL     CDivByReal
```

CEtoX

Category: Math

Description: Returns e^X where X is a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

```
B_CALL     CEtoX
```

CFrac

Category: Math

Description: Returns the fractional part of both the real and imaginary components of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1, OP2

Remarks:

Example:

```
B_CALL    CFrac
```

CIntgr

Category: Math

Description: Executes the Intgr function on a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1, OP2

Remarks: Return the next integer less than or equal to, for both the real and imaginary parts of the complex number.

See **Intgr**.

Example:

```
B_CALL     CIntgr
```

CkInt

Category: Math

Description: Tests floating-point number to be an integer.

Inputs:

Registers: HL = pointer to the exponent of the number to check

Flags: None

Others: None

Outputs:

Registers: Z = 1 if integer, Z = 0 if noninteger

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP5

Remarks: If exponent of OP1 > 13 then it is considered to be an integer.

Example:

CkOdd

Category: Math

Description: Tests if a floating-point number is odd or even.

Inputs:

Registers: HL = pointer to exponent of number to check

Flags: None

Others: None

Outputs:

Registers: None

Flags: If even, then Z = 1. If odd, then Z = 0.

Others: None

Registers destroyed: All

RAM used: None

Remarks: If exponent of OP1 > 13, then it is considered to be an even.
If 0 < Abs(OP1) < 1, then it is considered odd, negative exponent.

Example: Test a floating-point number in OP1 for add/even.

```
LD          HL,OP1+1
B_CALL      CkOdd
JP          Z,Is_Even
```

CkOP1C0

Category: Math

Description: Tests a complex number in OP1/OP2 to be (0,0).

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1/OP2 = complex number

Outputs:

- **Registers:** None
- **Flags:** If (0,0), then Z = 1; otherwise, Z = 0.
- **Others:** None

Registers destroyed: A

Remarks:

Example:

```
B_CALL     CkOP1C0
```

CkOP1Cplx

Category: Math

Description: Tests value in OP1 for complex data type.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: None

Flags: If OP1 contains a complex number, then Z = 1; otherwise, Z = 0.

Others: None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL     CkOP1Cplx
```

CkOP1FP0

Category: Math

Description: Tests floating-point number in OP1 to be 0.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating-point number

Outputs:

- **Registers:** None
- **Flags:** Z = 1: OP1 = 0
Z = 0: OP1 <> 0
- **Others:** None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL     CkOP1FP0
```

CkOP1Pos

Category: Math

Description: Tests floating-point number in OP1 to be positive.

Inputs:

Registers: None

Flags: None

Others: (OP1) = sign byte of floating-point number in OP1

Outputs:

Registers: ACC bit 7 = sign bit

Flags: If OP1 > 0, Z = 1; otherwise, Z = 0.

Others: None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL     CkOP1Pos
```

CkOP1Real

Category: Math

Description: Tests object in OP1 to be a real data type.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: ACC = data type of object in OP1

Flags: If OP1 contains a real number, then Z = 1; otherwise, Z = 0.

Others: None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL     CkOP1Real
```

CkOP2FP0

Category: Math

Description: Tests floating-point number in OP2 to be 0.

Inputs:

Registers: None

Flags: None

Others: OP2 = floating-point number

Outputs:

Registers: None

Flags: If OP2 = 0, then Z = 1; otherwise, Z = 0.

Others: None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL    CkOP2FP0
```

CkOP2Pos

Category: Math

Description: Tests floating-point number in OP2 to be positive.

Inputs:

Registers: None

Flags: None

Others: (OP2) = sign byte of floating-point number in OP2

Outputs:

Registers: ACC bit 7 = sign bit

Flags: If OP2 > 0, then Z = 1; otherwise, Z = 0.

Others: None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL     CkOP2Pos
```

CkOP2Real

Category: Math

Description: Tests object in OP2 to be a real data type.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: ACC = data type of object in OP2

Flags: If OP2 contains a real number, then Z = 1; otherwise, Z = 0.

Others: None

Registers destroyed: A

RAM used: None

Remarks:

Example:

```
B_CALL     CkOP2Real
```

CkPosInt

Category: Math

Description: Tests floating-point number in OP1 to be a positive integer.

Inputs:

Registers: OP1 = floating-point number

Flags: None

Others: None

Outputs:

Registers: If OP1 is a positive integer, then Z = 1.

Flags: None

Others: None

Registers destroyed: All

RAM used: None

Remarks:

Example:

```
B_CALL      CkPosInt      ; check OP1 = positive integer
JR          Z,PosInt      ; jump if positive integer
```

CkValidNum

Category: Math

Description: Checks for a valid number for a real or complex number in OP1/OP2.

Inputs:

Registers: OP1, if real
OP1 and OP2, if complex

Flags: None

Others: None

Outputs:

Registers: Err: Overflow if exponent > 100
Value set to 0 if exponent < -99

Flags: None

Others: None

Registers destroyed: AF, HL

Remarks: This should be used before storing a real or complex to a user variable or a system variable.

Intermediate results from the math operations can generate values outside of the valid exponent range for the TI–83 Plus. This routine will catch those cases.
If this is not done, then problems can occur when trying to display the invalid numbers.

This does not need to be done after every floating-point operation. The core math routines can handle exponents in the range or +/- 127.

Example: After a floating-point multiply, check the result for validity before stringing to variable X. Assume OP1 and OP2 have values already.

```
            B_CALL      FPMult       ; generate value to store to 'X'
;
            B_CALL      CkValidNum   ; make sure valid exponent
;
            B_CALL      StoX         ; store to 'X'
```

CLN

Category: Math

Description: Computes the natural log of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

```
B_CALL     CLN
```

CLog

Category: Math

Description: Computes the base 10 log of a complex number.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1/OP2 = complex number

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

```
B_CALL     CLog
```

ClrLp

Category: Math

Description: Clears a memory block (to 00h's).

Inputs:

Registers: HL = address of start of memory block
B = number of bytes to clear

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block cleared

Registers destroyed: B, HL

Remarks: None

Example:

ClrOP1S

Category: Math

Description: Clears the mantissa sign bit in OP1.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Registers destroyed:

Remarks: This routine only acts on the display, not the ***textShadow***.

Example:

CMltByReal

Category: Math

Description: Multiplies a complex number by a real number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number
OP3 = floating point real number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result, OP1/OP2   OP3
OP3 = intact

Registers destroyed: All

RAM used: OP1 – OP4

Remarks:

Example:

```
B_CALL     CMltByReal
```

CMult

Category: Math

Description: Multiplication of two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) * (second argument)

Registers destroyed: All

RAM used: OP1 – OP4

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

Conj

Category: Math

Description: Computes the complex conjugate of a real complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = real complex number

Outputs:

Registers: None

Flags: None

Others: OP2 = -OP2, negate imaginary
Set OP1/OP2 = current complex mode

Registers destroyed: All

Remarks: No error checking. Sets Ans to the current complex mode.

Example:

COP1Set0

Category: Math

Description: Puts a complex (0,0) in OP1/OP2.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex (0,0)

Registers destroyed: A, HL

Remarks: OP1 is not modified.

Example:

Cos

Category: Math

Description: Computes the cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: Current angle mode
OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

CosH

Category: Math

Description: Computes the hyperbolic cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

CpOP1OP2

Category: Math

Description: Compares floating-point values in OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point value
OP2 = floating-point value

Outputs:

Registers: None

Flags: Z = 1: OP1 = OP2
Z = 0, CA = 1: OP1 < OP2
Z = 0, CA = 0: OP1 >= OP2

Others: None

Registers destroyed: A, BC, DE, HL

Remarks: OP1 and OP2 are preserved.

Example:

CpOP4OP3

Category: Math

Description: Compares floating-point values in OP4 and OP3.

Inputs:

Registers: None

Flags: None

Others: OP4 = floating-point value
OP3 = floating-point value

Outputs:

Registers: None

Flags: Z = 1: OP4 = OP3
Z = 0, CA = 1: OP4 < OP3
Z = 0, CA = 0: OP4 >= OP3

Others: None

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks: OP4 and OP3 are preserved.

Example:

CRecip

Category: Math

Description: Computes the reciprocal of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = input complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = resulting complex number

Registers destroyed: All

RAM used: OP1 – OP4

Remarks:

Example:

```
B_CALL     CRecip
```

CSqRoot

Category: Math

Description: Computes the square root of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

```
B_CALL     CSqRoot
```

CSquare

Category: Math

Description: Computes the square of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1 – OP4

Remarks:

Example:

```
B_CALL     CSquare
```

CSub

Category: Math

Description: Subtracts two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) - (second argument)

Registers destroyed: All

RAM used: OP1 – OP3

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: Assume that variable X and Y both have complex values.

Recall the contents and subtract Y from X, such that OP1/OP2 = X - Y

```
             B_CALL      RclX          ; OP1/OP2 = complex value of X
;
;  This next call pushes OP1 the real part of the complex #, onto FPST;
;  then pushes OP2, the imaginary part, onto the FPST which pushes the
;  real part to FPS1 position.
;
;  FPS1 = 1st argument real part
;  FPST = 1st argument imaginary part
;
             B_CALL      PushMCplxO1   ; push 1st argument on FPS, X
;
             B_CALL      RclY          ; OP1/OP2 = complex value of Y
;
             B_CALL      CSub          ; OP1/OP2 = result X N Y, FPS
                                       ; is cleaned
```

CTenX

Category: Math

Description: Returns 10^X where X is a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

```
B_CALL    CTenX
```

CTrunc

Category: Math

Description: Returns the integer part of both the real and imaginary components of a complex number; no rounding is done.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

Registers destroyed: All

RAM used: OP1, OP2

Remarks: No rounding is done; for example, Trunc (1.5 + 3i) returns 1 + 3i.

Example:

```
B_CALL    CTrunc
```

Cube

Category: Math

Description: Computes the cube of a floating-point number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = OP1^3

Registers destroyed: A, BC, DE, HL

RAM used: OP1 – OP3

Remarks:

Example:

```
B_CALL   Cube
```

CXrootY

Category: Math

Description: Returns the complex root of a complex number, y^(1/x).

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument (y)
FPS1/FPST = first argument (x)

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result second_argument^(1/(first_argument))

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

CYtoX

Category: Math

Description: Raises a complex number to a complex power, y^x.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument (x)
FPS1/FPST = first argument (y)

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result first_argument^(second_argument)

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

DecO1Exp

Category: Math

Description: Decrements OP1 exponent.

Inputs:

Registers: None

Flags: None

Others: OP1

Outputs:

Registers: None

Flags: None

Others: Decrement OP1 exponent by one.

Registers destroyed: A

Remarks:

Example:

```
B_CALL     DecO1Exp
```

DToR

Category: Math

Description: Converts the floating-point number in OP1 from a degrees angle to a radian angle.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to convert

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point number representing the radian angle of the input value

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks:

Example:

EToX

Category: Math

Description: Computes e^OP1 = 10^(OP1*LOG(e)).

Inputs:

Registers: None

Flags: None

Others: OP1 = value e is raised to

Outputs:

Registers: None

Flags: None

Others: OP1 = result

Registers destroyed: All, OP2, OP3, OP4

Remarks:

Example:

ExpToHex

Category: Math

Description: Converts absolute value of one-byte. Exponent (in HL) to hexadecimal.

Inputs:

Registers: (HL) = exponent to convert

Flags: None

Others: None

Outputs:

Registers: (HL) = absolute value of exponent

Flags: None

Others: None

Registers destroyed: A

Remarks: This converts the floating point exponent value from the offset type (e.g., 7Fh = 10^-1, 80h = 10^0, 81h = 10^1,...) to a value of 0...n. It treats positive and negative exponents the same:

e.g., 80h = 0
81h = 1
82h = 2
7Fh = Λ1
7Eh = Λ2

See **OP1ExpToDec** for another exponent conversion routine.

Example:

```
        LD          HL,Exponent
        LD          (HL),7Eh
        B_CALL      ExpToHex        ; change (HL) from FEh N> 02h.
```

Factorial

Category: Math

Description: Computes the factorial of an integer or a multiple of .5.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating-point number, must be an integer or a multiple of .5 in the range of -.5 to 69

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = factorial of input, floating-point number. Else, error if input is out of range.

Registers destroyed: All

RAM used: OP1 – OP3

Remarks:

Example:

FPAdd

Category: Math

Description: Floating point addition of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result OP1 + OP2

Registers destroyed: All

RAM used: OP1, OP2

Remarks:

Example:

```
B_CALL    FPAdd
```

FPDiv

Category: Math

Description: Floating point division of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result OP1 / OP2
OP2 = intact

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks:

Example:

```
B_CALL     FPDiv
```

FPMult

Category: Math

Description: Floating point multiplication of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result OP1 * OP2
OP2 = intact

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks:

Example:

```
B_CALL    FPMult
```

FPRecip

Category: Math

Description: Floating point reciprocal of OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result 1 / OP1
OP2 = input OP1

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks:

Example:

```
B_CALL     FPRecip
```

FPSquare

Category: Math

Description: Floating point square of OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result OP1   OP1
OP2 = input OP1

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks:

Example:

```
B_CALL      FPSquare
```

FPSub

Category: Math

Description: Floating point subtraction of OP1 and OP2.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating-point number, argument one
  OP2 = floating-point number, argument two

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating point result OP1 N OP2

Registers destroyed: All

RAM used: OP1, OP2

Remarks:

Example:

```
B_CALL      FPSub
```

Frac

Category: Math

Description: Returns the fractional part of a floating-point number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result

Registers destroyed: All

RAM used: OP1

Remarks: No rounding; for example, Frac (1.5) = .5

Example:

```
B_CALL    Frac
```

HLTimes9

Category: Math

Description: Multiplies HL by nine.

Inputs:

Registers: HL = multiplicand

Flags: None

Others: None

Outputs:

Registers: HL = HL * 9 modulo 65536

Flags: CA = 1: answer larger than 65535
CA = 0: answer less than 65535

Others: None

Registers destroyed: BC

Remarks: None

Example:

HTimesL

Category: Math

Description: Multiplies H (register) * L (register).

Inputs:

- **Registers:** H, L
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** HL = product of (original H) * (original L)
- **Flags:** None
- **Others:** None

Registers destroyed: B, DE

Remarks: Restriction: H cannot be 0; If H is 0, performs 256 * L.
Cannot overflow if H > 0.

Example:

Int

Category: Math

Description: Rounds a floating-point number to an integer.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to round

Outputs:

Registers: None

Flags: None

Others: OP1 = Int (OP1)

Registers destroyed: All

RAM used: OP1

Remarks: The mantissa sign of the input has no affect on the result.

Example:

```
B_CALL     Int
```

Intgr

Category: Math

Description: Returns the integer.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result

Registers destroyed: A, BC, DE, HL

Remarks: If OP1 is an integer, then result = OP1. Otherwise, for positive numbers, returns the same as Trunc (OP1); for negative numbers, returns the Trunc (OP1 - 1).

Example:

InvOP1S

Category: Math

Description: Negates a floating-point number OP1, if OP1 = 0 then set OP1 = positive.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number. No check is made for a valid floating-point number.

Outputs:

Registers: None

Flags: None

Others: OP1 = Λ(OP1), unless 0 then it is set to positive.

Registers destroyed: A

Remarks:

Example: Set OP1 = Λ1

```
B_CALL      OP1Set1      ; OP1 = floating point 1
B_CALL      InvOP1S      ; OP1 = -1
```

InvOP1SC

Category: Math

Description: Used to negate a complex number in OP1/OP2 by negating both OP1 and OP2. If OP1 or OP2 = 0, then that OP register is set positive.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = two floating-point numbers that make up a complex number

Outputs:

Registers: None

Flags: None

Others: OP1 = -(OP1), unless 0 then it is set to positive
OP2 = -(OP2), unless 0 then it is set to positive

Registers destroyed: A

Remarks:

Example:

InvOP2S

Category: Math

Description: Negates a floating-point number OP2, if OP2 = 0 then set OP2 = positive.

Inputs:

Registers: None

Flags: None

Others: OP2 = floating-point number, no check is made for a valid floating-point number.

Outputs:

Registers: None

Flags: None

Others: OP2 = -(OP2), unless 0 then it is set to positive

Registers destroyed: A

Remarks:

Example: Set OP2 = -1

```
B_CALL      OP2Set1        ; OP2 = floating point 1
B_CALL      InvOP2S        ; OP2 = -1
```

InvSub

Category: Math

Description: Negates OP1 and add to OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point
OP2 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value (-OP1) + OP2

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks: None

Example:

LnX

Category: Math

Description: Returns natural log of a floating-point number in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be positive

Outputs:

Registers: None

Flags: None

Others: Error if OP1 is negative
Else OP1 = Ln(OP1)

Registers destroyed: All

RAM used: OP1 – OP5

Remarks: A system error can be generated. See section on Error Handlers.

Example: Compute the Ln(OP1), install an error handler to avoid the system reporting the error.

```
             AppOnErr    CatchError     ; install error handler
;
             B_CALL      LnX            ; compute Ln(OP1)
;
             AppOffErr                  ; remove error handler, no
                                        ; error occurred
;
             RET
;
; come here if LnX generated an error
;
CatchError:
```

LogX

Category: Math

Description: Returns log base 10 of a floating-point number in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be positive

Outputs:

Registers: None

Flags: None

Others: Error if OP1 is negative
Else OP1 = Log(OP1)

Registers destroyed: All

RAM used: OP1 – OP5

Remarks: A system error can be generated. See section on Error Handlers.

Example: See **LnX**.

Max

Category: Math

Description: Returns the maximum (OP1, OP2), two floating-point numbers.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number
OP2 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = maximum (OP1, OP2)
OP2 = intact

Registers destroyed: All

RAM used: OP1 – OP4

Remarks: See **CpOP1OP2**, for non destructive compare.

Example:

Min

Category: Math

Description: Computes the minimum of two floating-point numbers.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number argument one
OP2 = floating-point number argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = minimum (OP1, OP2)
OP2 = intact
OP3 = argument one
OP4 = argument two

Registers destroyed: A, BC, DE, HL

RAM used: OP1 – OP4

Remarks:

Example:

Minus1

Category:	Math
Description:	Floating point subtraction of one from OP1.
Inputs:	
Registers:	None
Flags:	None
Others:	OP1 = floating-point number
Outputs:	
Registers:	None
Flags:	None
Others:	OP1 = floating-point result OP1 - 1
Registers destroyed:	All
RAM used:	OP1, OP2
Remarks:	
Example:	`B_CALL Minus1`

OP1ExpToDec

Category: Math

Description: Converts absolute value of exponent to a bcd number.

Inputs:

Registers: None

Flags: None

Others: OP1 + 1 = exponent to convert

Outputs:

Registers: (HL) = OP1 + 1 = |Exp| as hex
A = |Exp| as bcd

Flags: None

Others: OP1 + 1 = |Exp| as hex

Registers destroyed: A, BC

Remarks: Overflow Error if |Exp| > 99

Example:

```
; Input OP1 + 1 value -> Output OP1 + 1 and A register
81h (10^1  )   ->    01h        &  01h
7Fh (10^-1 )   ->    01h        &  01h
8Dh (10^13 )   ->    0Dh        &  13h
73h (10^-13)   ->    0Dh        &  13h
```

OP1Set0, OP1Set1, OP1Set2, OP1Set3, OP1Set4, OP2Set0, OP2Set1, OP2Set2, OP2Set3, OP2Set4, OP2Set5, OP2Set60, OP3Set0, OP3Set1, OP3Set2, OP4Set0, OP4Set1, OP5Set0

Category: Math Utility

Description: Sets value of OP(x) to floating point (value).

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(x) = floating-point value

Registers destroyed: A, HL

Remarks: Combinations Available:

Value	0	1	2	3	4	5	60
Register							
OP1	X	X	X	X	X		
OP2	X	X	X	X	X	X	X
OP3	X	X	X				
OP4	X	X					
OP5	X						

Example:

```
B_CALL    OP2Set5
```

OP2Set8

Category: Math

Description: Sets OP2 = floating point 8.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 = floating point 8

Registers destroyed: A, HL

Remarks:

Example:

OP2SetA

Category: Math

Description: Sets OP2 = floating-point value between 0 and 9.9.

Inputs:

Registers: ACC = two digits of mantissa to set OP2 to

Flags: None

Others: OP2 set to floating-point value

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A, HL

Remarks:

Example:

```
; Set OP2 = 7.6
             LD          A,76h         ; mantissa digits
             B_CALL      OP2SetA       ; OP2 = 7.6
```

Plus1

Category: Math

Description: Floating point addition of one to OP1.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating-point number

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating-point result OP1 + 1

Registers destroyed: All

RAM used: OP1, OP2

Remarks:

Example:

```
B_CALL     Plus1
```

PToR

Category: Math

Description: Converts complex number in OP1/OP2 from a polar complex number to a rectangular complex number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number radius part of complex number
OP2 = floating-point number angle part of complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = rectangular representation of input polar complex number

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

RandInit

Category: Math

Description: Initializes random number seeds to default value.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: HL, DE, BC

Remarks: Seeds initialized.

Example:

Random

Category: Math

Description: Returns a random floating-point number, 0 < number < 1.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 = floating point random number

Registers destroyed: All

RAM used: OP1 – OP3

Remarks: See **RnFx** and **Round** routines.

Example:

RName

Category: Math

Description: Constructs a name for real variable R in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for R in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

RndGuard

Category: Math

Description: Rounds a floating-point number to 10 mantissa digits. The exponent value has no effect on this routine.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to round to 10 mantissa digits

(fmtDigits) = current fix value
0ffh = floating, no rounding will be done
Otherwise, the value is the number of decimal
Digits to round to, 0 – 9

Outputs:

Registers: None

Flags: None

Others: OP1 = input floating point rounded to 10 mantissas digits

Registers destroyed: All

RAM used: OP1

Remarks: See the **RnFx** and **Round** routines.

Example:

RnFx

Category: Math

Description: Rounds a floating-point number to the current FIX setting for the calculator. This will round the digits following the decimal point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to round

Outputs:

Registers: None

Flags: None

Others: OP1 = input rounded to at maximum of 10 mantissa digits

(fmtDigits) = current fix value
0ffh = floating, no rounding will be done
Otherwise, the value is the number of decimal
Digits to round to, 0 – 9

Registers destroyed: All

RAM used: OP1

Remarks: See **Round** and **RndGuard** routines.

Example:

Round

Category: Math

Description: Rounds a floating-point number to a specified number of decimal places. This will round the digits following the decimal point.

Inputs:

Registers: D = number of decimal places to round to, 0 – 9

Flags: None

Others: OP1 = floating-point number to round

(fmtDigits) = current fix value
0ffh = floating, no rounding will be done
Otherwise, the value is the number of decimal digits to round to, 0 – 9

Outputs:

Registers: None

Flags: None

Others: OP1 = input rounded to at maximum of 10 mantissa digits

Registers destroyed: All

RAM used: OP1

Remarks: See **RnFx** and **RndGuard** routines.

Example:

RToD

Category: Math

Description: Converts the floating-point number in OP1 from a radian angle to a degree angle.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to convert

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point number representing the degree angle of the input value.

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: See **DToR** routine.

Example:

RToP

Category: Math

Description: Converts complex number in OP1/OP2 from a rectangular complex number to a polar complex number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number X part of complex number
OP2 = floating-point number Y part of complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = polar representation of input rectangular complex number

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: See **RToP** routine.

Example:

Sin

Category: Math

Description: Computes the sine and cosine of a floating point.

Inputs:

Registers: Current angle mode
OP1 = floating point

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = sine (floating point)
OP2 = cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

```
B_CALL     Sin
```

SinCosRad

Category: Math

Description: Computes the sine and cosine of a floating point and radian mode is forced.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = sine (floating point)
OP2 = cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

```
B_CALL     SinCosRad
```

SinH

Category: Math

Description: Computes hyperbolic sine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic sine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

```
B_CALL    SinH
```

SinHCosH

Category: Math

Description: Computes the hyperbolic sine and cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic sine (floating point)
OP2 = hyperbolic cosine (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

```
B_CALL     SinHCosH
```

SqRoot

Category: Math

Description: Returns the square root of OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be positive

Outputs:

Registers: None

Flags: None

Others: Error if OP1 is negative, else OP1 = Sqrt(OP1)

Registers destroyed: All

RAM used: OP1 – OP3

Remarks: See section on Error Handlers.

Example:

Tan

Category: Math

Description: Computes the tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: Current angle mode
OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = tangent (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

```
B_CALL     Tan
```

TanH

Category: Math

Description: Computes the hyperbolic tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic tangent (floating point)

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

```
B_CALL    TanH
```

TenX

Category: Math

Description: Returns 10^(OP1).

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = 10^(OP1)

Registers destroyed: All

RAM used: OP1 – OP4

Remarks:

Example:

ThetaName

Category: Math

Description: Constructs a name for real variable Theta in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for Theta in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

Times2

Category: Math

Description: Calculates OP1 times two.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value OP1 * 2.0
OP2 = floating point 2

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks: None

Example:

TimesPt5

Category: Math

Description: Calculates OP1 times 0.5.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value OP1 * 0.5
OP2 = floating point 0.5

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks:

Example:

TName

Category: Math

Description: Constructs a name for real variable T in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for T in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

ToFrac

Category: Math

Description: Converts a floating-point number to the integer numerator and integer denominator of the equivalent fraction.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Carry = 0: Success
= 1: Failure.

Others: OP1:
On Failure — unchanged.
On Success — Numerator (floating-point integer)

OP2:
On Failure — unchanged.
On Success — Denominator (floating-point integer)

Registers destroyed: All

Remarks: Also modifies OP3, OP4, OP5, OP6.
Smallest possible denominator is created.
Fails if denominator must be > 999.

Example:

```
             LD          HL,ExampleNum
             RST         rMov9ToOP1
; OP1 = 1.25
             B_CALL      ToFrac
; Convert to fraction form

; Carry is now 0 (success)

; OP1 now contains: 00h 80h 50h 00h 00h 00h 00h 00h 00h = 5

; OP2 now contains: 00h 80h 40h 00h 00h 00h 00h 00h 00h = 4
             LD          HL,ExampleNum2
             RST         rMov9ToOP1
; OP1 = 1.2345678901234
             B_CALL      ToFrac
; Convert to fraction form
; Carry is now 1 (failure)
; ExampleNum = 1.25
ExampleNum:  DB 00h, 80h, 12h, 50h, 00h, 00h, 00h, 00h, 00h
; ExampleNum2 = 1.2345678901234
ExampleNum:  DB 00h, 80h, 12h, 34h, 56h, 78h, 90h, 12h, 34h
```

Trunc

Category: Math

Description: Truncates the fractional portion of a floating-point number returning the integer portion with no rounding.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = Trunc (OP1)

Registers destroyed: All

RAM used: OP1 – OP2

Remarks:

Example:

```
Trunc(1.5) = 1
```

XName

Category: Math

Description: Constructs a name for real variable X in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for X in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

XRootY

Category: Math

Description: Inverses power function and returns OP1^(1/OP2).

Inputs:

Registers: None

Flags: None

Others: OP1 = number to find root of, floating point
OP2 = root to find, floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = result if no error, floating point

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

YName

Category: Math

Description: Constructs a name for real variable Y in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for Y in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

YToX

Category: Math

Description: Power function, returns OP1^OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = number to raise to a power, floating point
OP2 = power, floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = result if no error, floating point

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

Zero16D

Category: Math

Description: Sets eight-byte memory block to all 00h’s.

Inputs:

Registers: HL = start of target block in memory

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block starting at original HL is all 00h’s

Registers destroyed: A, HL

Remarks:

Example:

ZeroOP

Category: Math

Description: Sets 11 bytes in OP(x) to 00h.
Note that this does not set the value to floating point 0.0.

Inputs:

Registers: HL = pointer to OP(x), x = 1...6

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(x) = all 11 bytes 00h

Registers destroyed: A (= 0), HL

Remarks:

Example:

```
; Set OP2 contents to all 00h:
; OP2+0 OP2+1 OP2+3 OP2+4 OP2+5 OP2+6 OP2+7 OP2+8 OP2+9 OP2+10
; 00h   00h   00h   00h   00h   00h   00h   00h   00h   00h
             LD          HL,OP2
             B_CALL      ZeroOP
```

ZeroOP1, ZeroOP2, ZeroOP3

Category: Math

Description: Sets 11 bytes in OP(x) to 00h.
Note that this does not set the value to floating point 0.0.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(x) = all 11 bytes 00h

Registers destroyed: A(= 0), HL

Remarks: Combinations Available:
(x) = 1, 2, 3

Example:

```
; Set OP2 contents to all 00h:
; OP2+0 OP2+1 OP2+3 OP2+4 OP2+5 OP2+6 OP2+7 OP2+8 OP2+9 OP2+10
; 00h   00h   00h   00h   00h   00h   00h   00h   00h   00h
             B_CALL      ZeroOP2
```

11 System Routines — Matrix

AdrMEle

Category: List

Description: Computes the RAM address of an element of a matrix.

Inputs:

Registers: DE = pointer to start of matrix's data storage, output of **FindSym**
BC = element's (row, column) to compute address of Matrix Element (1,1) is checked for real or complex data type to determine if the matrix is real or complex.

Flags: None

Others: None

Outputs:

Registers: HL = pointer in RAM to start of desired element

Flags: None

Other: None

Registers destroyed: All

Remarks: This routine does not check to see if the element's address requested is within the current dimension of the matrix.
Do not use this routine on a matrix that does not have element (1,1) initialized.

Example: Compute the address of element (5,6) of matrix [A].

```
             LD          HL,MatAName
             RST         rMov9ToOP1         ; OP1 = [A] name
             B_CALL      FindSym            ; look it up
             JP          C,Undefined_A      ; jump out if [A] is not
                                            ; defined;
             LD          A,B                ; if b<>0 then [A] is
                                            ; archived in Flash ROM
             OR          A
             JP          NZ,Archived_A      ; jump if not in RAM;
                                            ; DE = pointer to start of
                                            ; matrix data storage;
             LD          BC,5*256+6         ; element's address
                                            ; desired
             B_CALL      AdrMEle            ; RET HL = pointer to
                                            ; element (5,6)
             RET
MatAName:
             DB          MatObj,tVarMat,tMatA,0,0
```

AdrMRow

Category: Matrix

Description: Computes the RAM address of the start of a row of a matrix.

Input:

Registers: DE = pointer to start of matrix's data storage, output of **FindSym**
B = row to compute address of

Matrix Element (1,1) is checked for real or complex data type to determine if the matrix is real or complex.

Do not use this routine on a matrix that does not have element (1,1) initialized.

Flags: None

Others: None

Output:

Registers: HL = pointer in RAM to start of desired element

Flags: None

Others: None

Registers destroyed: All

Remarks: This routine does not check to see if the row address requested is within the current dimension of the matrix. See **AdrMEle** routine.

Example:

GetMToOP1

Category: Matrix

Description: Copies an element from a matrix to OP1.

Input:

Registers: BC = element to get, row,col
DE = pointer to start of matrix's data storage

Flags: None

Others: None

Output:

Registers: HL = pointer to next element in the same row, or the start of the next row of the matrix.

Flags: None

Other: OP1 = matrix element, floating-point number

Registers destroyed: All

Remarks:

Example:

PutToMat

Category: Matrix

Description: Stores a floating-point number to an existing element of a matrix.

Inputs:

Registers: BC = (row, column) to store to
There is no check to see if this element is valid for the matrix.

DE = pointer to the start of the matrix's data area, output of **FindSym**

Flags: None

Others: None
OP1 = floating-point number

Outputs:

Registers: DE = pointer to next element in the matrix. This will be the next element in the same row or the start of the next row.

Flags: None

Others: OP1 = intact

Registers destroyed: All

Remarks:

Example: Look up MatA and store 1 to element (5,7).

```
             LD        HL,MatAname
             B_CALL    Mov9ToOP1     ; OP1 = name
;
             B_CALL    FindSym       ; look up
             RET       C             ; return if undefined
;
                                     ; DE = pointer to data area of
                                     ; matrix
;
             PUSH      DE            ; save pointer
             B_CALL    OP1Set1       ; OP1 = 1
;
             POP       DE
             LD        BC,5*257+7    ; element to store to (5,7)
             B_CALL    PutToMat      ; store 1 to element (5,7)
             RET
MatAname:
             DB        MatObj,tVarMat,tMatA,0
```

12 System Routines — Memory

Arc_Unarc

Category: Memory

Description: Swaps a variable between RAM and archive.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 contains variable name

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** Symbol table and data area (RAM and Flash) modified.

Registers destroyed: All

Remarks: Destroys OP3 as well.
Will unarchive a variable already archived and will archive a variable that is currently unarchived.

Gives an Err: Variable for any name that is not archivable or unarchivable (e.g., Groups cannot be unarchived and X cannot be archived).
Gives an Err: Undefined for any name that does not already exist.

Does memory checking to make sure there is enough space (in RAM or in Archive) to store the variable. Generates a memory error if not.

Example:

```
                                          ; unarchive variable A (real
                                          ; or complex) if it is
                                          ; archived:
              B_CALL      ZeroOP1         ; set OP1 to all 0s
              LD          (OP1+1),tA      ; want to look for floating
                                          ; point number named 'A'
              RST         rFindSym        ; Data pointer -> DE
                                          ; System pointer -> HL
                                          ; C if none
              JR          C, skip         ; does not exist, so skip
              CALL        NzIfArchived    ; NZ if was in RAM already.
              JR          Z, skip         ; not archived, so no need to
                                          ; unarchive
              B_CALL      Arc_Unarc       ; unarchive variable.
.....
NzlfArchived:
              LD          A,B             ; B has page information, NZ
                                          ; if archived.
              OR          A
              RET
```

ChkFindSym

Category: Memory

Description: Searches the symbol table structure for a variable.

This particular search routine must be used if the variable to search for is either a Program, AppVar, or Group. It will also work for variables of other types as long as the data type in OP1 input is correct.

This is used to determine if a variable is created and also to return pointers to both its symbol table entry and data storage area.
This will also indicate whether or not the variable is located in RAM or has been archived in Flash ROM.

Inputs:

Registers: (OP1) = one-byte, data type of variable to search for.
This routine will fail if this data type is not correct.
(OP1 + 1) to (OP1 + 8) = variable name

Flags: None

Others: None

Outputs:

Registers: CA flag = 1 if symbol was not found
= 0 if symbol was found

Also if found:
ACC lower 5 bits = data type
ACC upper 3 bits = system flags about variable, do “AND 1Fh” to get type only
B = 0 if variable is located in RAM else variable is archived
B = ROM page located on
If variable is archived then its data cannot be accessed directly, it must be unarchived first.
HL = pointer to the start of the variables symbol table entry
DE = pointer to the start of the variables data area if in RAM

Flags: None

Others: OP1 = variable name

Registers destroyed: All

Remarks: This will not find system variables that are preallocated in system RAM such as Xmin, Xmax etc. Use **RclSysTok** to retrieve their values.

Note: ChkFindSym will not find Applications.

(continued)

ChkFindSym *(continued)*

Example: Look for AppVar MYAPPVAR in the symbol table.
If it exists and is archived then unarchive it and relook it up.

If it does not exist ; create it with a size of 100 ; bytes.

```
Relook:
              LD          HL,VarName
              B_CALL      Mov9ToOP1       ; OP1 = variable name
              B_CALL      ChkFindSym      ; look up
              JR          NC,VarCreated   ; jump if it exists
;
              LD          HL,100          ; size to create at
              B_CALL      CreateAppVar    ; create it, HL = pointer to
                                          ; sym entry, DE = pointer to
                                          ; data
              PUSH        HL
              PUSH        DE              ; save during move
              B_CALL      OP4ToOP1        ; OP1 = name
              POP         DE              ; restore
              POP         HL
              JR          Done
VarCreated:
              LD          A,B             ; check for archived
              OR          A               ; in RAM ?
              JR          Z,done          ; yes
              B_CALL      Arc_Unarc       ; unarchive if enough RAM
              JR          Relook          ; look up pointers again in
                                          ; RAM now done:
              RET
;
VarName:
              DB          AppVarObj,'MYAPPVAR',0
```

CleanAll

Category: Memory

Description: Deletes all temporary variables from RAM.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Temporary variables are all deleted

Registers destroyed: All

Remarks: This routine should only be used when there are no temporary variables that exist and are still being used. See the Temporary Variables section in Chapter 2 for further information. See the **ParseInp** and **MemChk** routines.

Example:

CloseProg

Category: Memory

Description: This routine is used after **EditProg** to return unused RAM back to free RAM. The size bytes of the variable are updated by this routine. An application should not update them.

Inputs:

Registers: Each of these are two-bytes:

(iMathPtr1) = pointer to the start of the variables data storage area

(iMathPtr2) = pointer to the byte following the variable data, this will be used to calculate the new size of the variable

(iMathPtr3) = pointer to the byte AFTER the last byte of free RAM inserted

(iMathPtr4) = size of RAM block moved to allow the RAM to be inserted DO NOT CHANGE THIS VALUE.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: The variable's size is changed. Unused RAM returned to free RAM. Normal allocating and deallocating of RAM can resume.

Registers destroyed: All

Remarks:

Example:

CmpSyms

Category: Memory

Description: Compares Name @HL with Name @DE.

Inputs:

Registers: HL = end of first name in RAM
DE = end of second name in RAM
B = length of name

Flags: None

Others: None

Outputs:

Registers: C = number of letters that match
C = original B if all letters match

Flags: Carry set if Sym2 (HL) > Sym1 (DE)

Others: None

Registers destroyed: AF, BC, DE, HL

Remarks: The names must be the same size. The name lengths should have already been compared before calling this routine.

Example:

```
; See if the name last used for the Xlist variable in statistics is
; the name "ZEBRA"
             LD          HL,StZebra
             RST         rMov9ToOP1     ; Move 9 bytes to OP1:
                                        ; "ZEBRA" + junk
             LD          DE,OP1+4
             LD          HL,StatX+4
             LD          B,5            ; compare 5 bytes
             B_CALL      CmpSyms        ; If C = 5 then OP1 = StatX
                                        ; name
             LD          A,C
             CP          5
             JR          Z,Match
             JR          NoMatch

             StZebra:    DB  "ZEBRA"
```

Create0Equ

Category: Memory

Description: Creates an equation variable of size 0 in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of equation to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create an empty Y1 equation.

```
             LD          HL,Y1name
             RST         rMov9ToOP1  ; OP1 = name
             B_CALL      Create0Equ  ; if returns then variable created

Y1name:      DB          EquObj,tVarEqu,tY1,0,0
```

CreateAppVar

Category: Memory

Description: Creates an AppVar variable in RAM.

Inputs:

Registers: HL = size of AppVar to create in bytes

Flags: None

Others: OP1 = name of AppVar to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. Users can only delete and link AppVars. They are intended for Apps to use for state saving upon exiting. See section on Creating Variables.

Example: Create AppVar DOG, 50 bytes in size.

```
             LD          HL,DOGname
             RST         rMov9ToOP1      ; OP1 = name
;
             LD          HL,50
             B_CALL      CreateAppVar    ; if returns then variable
                                         ; created

DOGname:     DB          AppVarObj,'DOG',0
```

CreateCList

Category: Memory

Description: Creates a complex list variable in RAM.

Inputs:

Registers: HL = number of elements in the list

Flags: None

Others: OP1 = name of list to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the elements is done, assume random. See section on Creating Variables.

Example: Create complex list L1 with 50 elements.

```
             LD          HL,L1name
             RST         rMov9ToOP1    ; OP1 = name
;
             LD          HL,50
             B_CALL      CreateCList   ; if returns then variable
                                       ; created

L1name:      DB          CListObj,tVarLst,tL1,0,0
```

CreateCplx

Category: Memory

Description: Creates a complex variable in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of complex to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. This should not be used to create temp storage space, A-Z or THETA. No initialization is done, assume random. See section on Creating Variables.

Example: Create complex A.

```
             LD          HL,Aname
             RST         rMov9ToOP1        ; OP1 = name
;
             B_CALL      CreateCplx        ; if returns then variable
                                           ; created

Aname:       DB          CplxObj,'A',0,0
```

CreateEqu

Category: Memory

Description: Creates an equation variable in RAM.

Inputs:

Registers: HL = size of equation to create in bytes

Flags: None

Others: OP1 = name of equation to create

Outputs:

Registers: HL = pointer to variable’s symbol table entry
DE = pointer to variable’s data storage, size bytes

Flags: None

Others: OP4 = variable’s name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create Y1 equation 50 bytes in size.

```
              LD          HL,Y1name
              RST         rMov9ToOP1   ; OP1 = name
;
              LD          HL,50
              B_CALL      CreateEqu    ; if returns then variable created

Y1name:       DB          EquObj,tVarEqu,tY1,0,0
```

CreatePair

Category: Memory

Description: Creates a pair of parametric graph equations.

There should never be a situation where only 1 of a pair of parametric equations is created without the other. This routine will check that there is enough memory to create both equations before creating any.

Inputs:

Registers: HL = size to create the equation specified in OP1, either xt or yt. The member of the pair not specified will be created empty.

Flags: None

Others: OP1 = pair member name to create with the specified size

Outputs:

Registers: HL = size of pair member specified

Flags: None

Others: OP1 = pair member name specified
OP4 = pair member name not specified

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM to create the pair.
If xt# is specified then yt# is created empty. If yt# is specified then xt# is created empty.
No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create parametric pair of equations xt1 and yt1, yt1 at size 50.

```
             LD          HL,yt1name
             RST         rMov9ToOP1   ; OP1 = name
;
             LD          HL,50
             B_CALL      CreatePair   ; if returns then variables
                                      ; created
                                      ; OP1 = yt1, OP4 = xt1, HL = 50

yt1name:     DB          EquObj,tVarEqu,ty1t,0,0
```

CreatePict

Category: Memory

Description: Creates a picture variable in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of picture to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. The size of a Pic var is 756 bytes, it does not allocate space for the last row of pixels, that row is never used by the system graph routines.

If you need to save a bitmap of the entire display to a variable then an AppVar should be used. The only drawback to using an AppVar is that the Pic could not be displayed by the user when the app is not executing.

No initialization is done, assume random. See section on Creating Variables.

Example: Create Pic Pic1.

```
             LD          HL,Pic1name
             RST         rMov9ToOP1    ; OP1 = name
;
             B_CALL      CreatePict    ; if returns then variable
                                       ; created

Pic1name:    DB          PictObj,tVarPict,tPic1,0,0
```

CreateProg

Category: Memory

Description: Creates a program variable in RAM.

Inputs:

Registers: HL = size of program to create in bytes

Flags: None

Others: OP1 = name of program to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create Program DOG, 50 bytes in size.

```
             LD        HL,DOGname
             RST       rMov9ToOP1       ; OP1 = name
;
             LD        HL,50
             B_CALL    CreateProg       ; if returns then
                                        ; variable created

DOGname:     DB        ProgObj,'DOG',0
```

CreateProtProg

Category: Memory

Description: Creates a protected program variable in RAM.

Inputs:

Registers: HL = size of program to create in bytes

Flags: None

Others: OP1 = name of program to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. Users cannot delete or edit protected programs, they can be deleted from an application. See section on Creating Variables.

Example: Create protected Program DOG, 50 bytes in size.

```
             LD          HL,DOGname
             RST         rMov9ToOP1      ; OP1 = name
;
             LD          HL,50
             B_CALL      CreateProtProg  ; if returns then variable
                                         ; created

DOGname:     DB          ProtProgObj,'DOG',0
```

CreateReal

Category: Memory

Description: Creates a real variable in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of real to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. This should not be used to create temp storage space, only A-Z and theta. No initialization is done, assume random. See section on Creating Variables.

Example: Create real A.

```
              LD          HL,Aname
              RST         rMov9ToOP1     ; OP1 = name
;
              B_CALL      CreateReal     ; if returns then variable
                                         ; created

Aname:        DB          RealObj,'A',0,0
```

CreateRList

Category: Memory

Description: Creates a real list variable in RAM.

Inputs:

Registers: HL = number of elements in the list

Flags: None

Others: OP1 = name of list to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the elements is done, assume random. See section on Creating Variables.

Example: Create real list CAT with 50 elements.

```
              LD          HL,CATname
              RST         rMov9ToOP1    ; OP1 = name
;
              LD          HL,50
              B_CALL      CreateRList   ; if returns then variable
                                        ; created

CATname:      DB          ListObj,tVarLst,'CAT',0
```

CreateRMat

Category: Memory

Description: Creates a real matrix variable in RAM.

Inputs:

Registers: HL = dimension of matrix, (row,col), 99 is maximum row or column

Flags: None

Others: OP1 = name of matrix to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, dimension

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the elements is done, assume random. See section on Creating Variables.

Example: Create matrix [A] with 5 rows and 8 columns.

```
             LD          HL,MatAname
             RST         rMov9ToOP1    ; OP1 = name
;
             LD          HL,5*256+8    ; 5 x 8
             B_CALL      CreateRMat    ; if returns then variable
                                       ; created

MatAname:    DB          MatObj,tVarMat,tMatA,0,0
```

CreateStrng

Category: Memory

Description: Creates a string variable in RAM.

Inputs:

Registers: HL = number bytes in string

Flags: None

Others: OP1 = name of string to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the string contents is done, assume random. See section on Creating Variables.

Example: Create string Str1 100 bytes in length.

```
             LD          HL,Str1name
             RST         rMov9ToOP1       ; OP1 = name
;
             LD          HL,100           ; size of string
             B_CALL      CreateStrng      ; if returns then variable
                                          ; created

Str1name:    DB          StrngObj,tVarStrng,tStr1,0,0
```

DataSize

Category: Memory

Description: Computes the size, in bytes, of the data portion of a variable in RAM.

Inputs:

Registers: ACC = data type
HL = pointer to first byte of data storage

Flags: None

Others: None

Outputs:

Registers: DE = size of data storage in bytes
HL = intact

Flags: None

Others: None

Registers destroyed: A, BC

Remarks: This routine cannot be used on archived variables or applications.

If the variable's data area has size information, like a list has two-bytes for number of elements, then those bytes are included in the computation.

Example:

```
; Find the size in bytes of the data area for list L1.
L1Name:
             DB         ListObj,tVarLst,tL1,0,0
;
             LD         HL,L1name
             RST        rMov9ToOP1    ; OP1 = L1
;
             B_CALL     FindSym       ; find in symbol table,
                                      ; DE = pointer to data
             AND        1Fh           ; ACC = data type information,
                                      ; real or complex list
             EX         DE,HL         ; HL = pointer to data storage
             B_CALL     DataSize      ; DE = size of data storage
                                      ; If L1 were a real list with 5
                                      ; elements then the size
                                      ; returned would be 47 bytes.
;       5 elements *9 for each = 45
;       2 size bytes           =  2
;                               ---
;                                47
```

DataSizeA

Category: Memory

Description: Computes the size, in bytes, of the data portion of a variable that has two size bytes as part of its data storage.
This routine applies to equations, lists, matrices, programs, AppVars.

Inputs:

Registers: ACC = data type
BC = two byte size information: dimension, number of bytes, number of elements

Flags: None

Others: None

Outputs:

Registers: DE = size of data storage in bytes

Flags: None

Others: None

Registers destroyed: All

Remarks: If the variable's data area has size information, like a list has two bytes for number of elements, then those bytes are included in the computation.

Example:

```
; Find the size in bytes of a complex list with 5 elements:
             LD          A,CListObj     ; ACC = data type information,
                                        ; cplx list
             LD          BC,5           ; number elements
;
             B_CALL      DataSizeA      ; DE = size of data storage
;
;
;       5 elements *18 for each = 90
;       2 size bytes            =  2
;                               ----
;                                 92
```

DeallocFPS

Category: Memory

Description: Removes space in nine-byte chunks from the Floating Point Stack.

Inputs:

- **Registers:** HL = number of chunks to remove
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** FPS (Floating Point Stack top) decreased by HL 9

Registers destroyed: DE, HL

Remarks: No values are removed from the deallocated space.

Example:

DeallocFPS1

Category:	Memory
Description:	Removes space in bytes from the Floating Point Stack.
Inputs:	
Registers:	DE = number of bytes to remove
Flags:	None
Others:	None
Outputs:	
Registers:	None
Flags:	None
Others:	FPS (Floating Point Stack top) decreased by HL
Registers destroyed:	HL
Remarks:	No values are removed from the deallocated space.
Example:	

DelMem

Category: Memory

Description: Deletes RAM from an existing variable. This routine will only delete the RAM. If the variable deleting from has a size field, it is NOT UPDATED. Updating must be done by the application.

Inputs:

Registers: HL = address of first byte to delete
DE = number of bytes to delete

Flags: None

Others: None

Outputs:

Registers: DE = intact
BC = amount deleted
RAM deleted

Flags: None

Others: None

Registers destroyed: All

Remarks: See **InsertMem** routine.

(continued)

DelMem *(continued)*

Example: Delete 10 bytes at the beginning of an AppVar.

```
;
             LD          HL,AppVarName
             RST         rMov9ToOP1       ; OP1 = name of AppVar
             B_CALL      ChkFindSym       ; look up in symTable
             JR          NC, Created      ; jump if it exists
;
             B_JUMP      ErrUndefined     ; error if not there
;
;  DE = pointer to size bytes of AppVar
;
Created:
             PUSH        DE               ; save pointer to start of
                                          ; size bytes of data
;
             INC         DE
             INC         DE               ; move DE to 1st byte of
                                          ; AppVar Data
;
             LD          HL,10            ; number bytes to insert
;
             EX          DE,HL            ; HL = pointer to start of
                                          ; delete, DE number bytes
             B_CALL      DelMem           ; delete the memory
;
             POP         HL               ; HL = pointer to size bytes
             PUSH        HL               ; save
;
             B_CALL      ldHLind          ; HL = size of AppVar,
                                          ; number bytes
;
             XOR         A                ; clear CA
;
             LD          BC,10
             SBC         HL,BC            ; decr by amount deleted
;
             EX          DE,HL
             POP         HL               ; pointer to size bytes
                                          ; location
;
             LD          (HL),E
             INC         HL
             LD          (HL),D           ; write new size.
;
;
AppVarName:  DB          AppVarObj,'AVAR',0
```

DelVar

Category: Memory

Description: Deletes a variable stored in RAM.

Inputs: All of the inputs for this routine are the outputs of **FindSym** and **ChkFindSym**. It is common to call one of these routines and then call **DelVar** immediately after.

Registers: HL = pointer to start of symbol table entry of variable
DE = pointer to start of data storage of variable
B = 0 if variable resides in RAM else it is the page in the archive it is stored

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: OP1 – OP6 are preserved.
Variable's symbol entry and data are deleted.
Graph is marked dirty if variable was used during graphing.
All global memory pointers are adjusted.
Error if the variable resides in the archive.

Example:

```
; Delete the variable 'A' if it exists
              LD          HL,AName
              RST         rMov9ToOP1    ; OP1 = variable a
;
              B_CALL      FindSym       ; look up
              JR          C, Deleted    ; jump if variable is not
                                        ; created
;
              B_CALL      DelVar
Deleted:
AName:
              DB          RealObj,'A',0,0
```

DelVarArc

Category: Memory

Description: Deletes a variable from RAM or the archive.

Inputs:

Registers: HL = pointer to symbol table entry of variable to delete
DE = pointer to start of data for variable
B = archived status
0 = RAM otherwise the ROM page in Flash for the variable

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Variable’s symbol entry and data deleted if in RAM, otherwise the symbol table entry is only deleted and the variable data is marked for deletion on the next garbage collection.

Graph is marked dirty if variable was used during graphing.

All global memory pointers are adjusted.

Registers destroyed: All

Remarks: See **DelVar** and **DelVarNoArc** routines.

Example:

DelVarNoArc

Category: Memory

Description: Deletes variable from RAM.
No archive checking performed.

Inputs:

Registers: HL = pointer to symbol table entry of variable to delete
DE = data pointer to data

Flags: None

Others: None

Outputs:

Registers: None

Flags: Regraph flag set if varGraphRef flag of symbol was set.

Others: None

Registers destroyed: All

Remarks: See **DelVar** for more information.
This routine should only be called if you are sure that your variable will never be archived. Generally, it is better to use the **DelVarArc** or **DelVar** routines.

Variable's symbol entry and data are deleted.
Graph is marked dirty if variable was used during graphing.
All global memory pointers are adjusted.
Error if the variable resides in the archive.

Example:

```
; Delete the variable 'A' if it exists:
             LD          HL,Aname
             RST         rMov9ToOP1       ; OP1 = variable a
;
             B_CALL      FindSym          ; look up
             JR          C, Deleted       ; jump if variable is not
                                          ; created
;
             B_CALL      DelVarNoArc
Deleted:
Aname:
             DB          RealObj,'A',0,0
```

EditProg

Category: Memory

Description: This routine will insert all of free RAM into a Program, Equation, or AppVar. The intent is for the variable to be able to be edited without having to continuously allocate and deallocate memory. Once the edit is completed, a call to **CloseProg** is made to return what is not used back to free RAM.

Inputs:

Registers: DE = pointer to start of variables data storage area

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Each of following are two-bytes:

(iMathPtr1) = pointer to the start of the variables data storage area. THIS MUST STAY INTACT WHILE THE EDIT IS IN SESSION.

(iMathPtr2) = pointer to the byte following the variable data. This is the next location the data area can grow into.

(iMathPtr3) = pointer to the byte AFTER the last byte of free RAM inserted. The data being input cannot be written into this RAM location.

(iMathPtr4) = size of RAM block moved to allow the RAM to be inserted. DO NOT CHANGE THIS VALUE.

Registers destroyed: All

Remarks: The application can must change the pointer value in (iMathPtr2) as the variables data size grows or shrinks. This value is needed by the close routine.

No memory allocation/deallocation can be done in this state.

Contents of variables may by copied or changed, but not their sizes.

The Floating Point Stack may be copied to/from, but not grown or shrunk.

The hardware stack may change, calls, RET, push, and pop.

Example:

EnoughMem

Category: Memory

Description: Checks if an imputed amount of RAM is available. This routine will also attempt to free RAM that is taken by temporary variables that have been marked dirty but not yet deleted.

Inputs:

- **Registers:** HL = amount of RAM to check for being available
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** DE = amount of RAM to check for being available
- **Flags:** CA = one (set) if there is insufficient RAM available
- **Others:** None

Registers destroyed: All

RAM used: None

Remarks: No error is generated.

See **MemChk**.

Example:

Exch9

Category: Memory

Description: Exchanges (swaps) two nine-byte blocks of memory.

Inputs:

Registers: DE = address of start of one nine-byte block
HL = address of start of second nine-byte block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Nine bytes originally at DE are now at original HL
Nine bytes originally at HL are now at original DE

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

ExLp

Category: Memory

Description: Exchanges blocks of memory of up to 256 bytes.

Inputs:

Registers: B = number of bytes; 0 = 256
DE = address of start of one nine-byte block
HL = address of start of second nine-byte block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Block originally at DE is now at original HL
Block originally at HL is now at original DE

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

FindAlphaDn

Category: Memory

Description: This is used to search the symbol table, for all of the variables of a certain type, alphabetically in descending order.

Each call to this routine returns the variable name preceding the one input in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = variable name to find the previous before, usually output from the last call to this routine.

(OP1) must have the type of variable searching for set.

The name input in order to have the very last name for a certain type varies by the variable's type:

Real, Complex, Programs, AppVars, Group Vars:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	**0FEh**	**?**	**?**	**?**	**?**	**?**	**?**	**?**

All other types:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	**variable token**	**0FEh**	**?**	**?**	**?**	**?**	**?**	**?**

Outputs:

Registers: If a previous variable name is found then:
HL = pointer to the symbol table entry of the variable found

Flags: CA = 0 if a previous variable name was found
= 1 if no previous variable name exists

Others: If a previous variable name is found then:
OP1 and OP3 = the variable name found

Otherwise :
OP1 = variable name input

Registers destroyed: All

(continued)

FindAlphaDn *(continued)*

RAM used: OP2, OP3
upDownPtr — two byte pointer

Remarks: ProgObj, ProtProgObj, and TempProgObj are grouped together.
ListObj and CListObj are grouped together.
NewEquObj and EquObj are grouped together.
See **FindAlphaUp**, **SrchVLstUp**, **SrchVLstDn**.

Example: Find all of the programs that are currently created, search alphabetically in descending order.

```
FindPrograms:
             B_CALL     ZeroOP1
             LD         A,ProgObj
             LD         (OP1),A         ; looking for a list
             LD         A,0FEh          ; name = FEh, so the last
                                        ; program alphabetically is
                                        ; found
             LD         (OP1+1),A
FindLoop:
             B_CALL     FindAlphaDn     ; see if find another program
                                        ; name
             RET        C               ; return if no more program
                                        ; names found yet
;
;  OP1 = next list name
;
             JR         FindLoop        ; find previous using one just
                                        ; found as input
```

FindAlphaUp

Category: Memory

Description: This is used to search the symbol table, for all of the variables of a certain type, alphabetically in ascending order.

Each call to this routine returns the next variable name following the one input in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = variable name to find the next after, usually output from the last call to this routine.

(OP1) must have the type of variable searching for set.

The name input in order to have the very first name for a certain type varies by the variable's type:

Real, Complex, Programs, AppVars, Group Vars:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	**00**	**?**	**?**	**?**	**?**	**?**	**?**	**?**

All other types:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	**variable token**	**0FFh**	**?**	**?**	**?**	**?**	**?**	**?**

Outputs:

Registers: If a next variable name is found then:
HL = pointer to the symbol table entry of the variable found

Flags: CA = 0 if a next variable name was found
= 1 if no next variable name exists

Others: If a next variable name is found then:
OP1 and OP3 = the variable name found

Otherwise:
OP1 = variable name input

Registers destroyed: All

(continued)

FindAlphaUp *(continued)*

RAM used: OP2, OP3
upDownPtr — two byte pointer

Remarks: ProgObj, ProtProgObj and TempProgObj are grouped together.
ListObj and CListObj are grouped together.
NewEquObj and EquObj are grouped together.
See **FindAlphaDn**, **SrchVLstUp**, **SrchVLstDn.**

Example: Find all of the lists that are currently created, search alphabetically in ascending order.

```
FindLists:
             B_CALL     ZeroOP1
             LD         A,ListObj
             LD         (OP1),A       ; looking for a list
             LD         A,tVarLst     ; list designator token
             LD         (OP1+1),A     ;
             LD         A,0FFh        ; set name to FFh, so that the
                                      ; first list alphabetically is
                                      ; found
             LD         (OP1+1),A
FindLoop:
             B_CALL     FindAlphaUp   ; see if find another list name
             RET        C             ; return if no more list names
                                      ; not found yet
;
;  OP1 = next list name
;
             JR         FindLoop      ; find next using one just found
                                      ; as input
```

FindApp

Category: Memory

Description: Searches for an application that may be stored in Flash ROM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of application to search for

Outputs:

Registers: A = ROM page application starts on if found

Flags: CA = 0 if application exists
CA = 1 if application does not exist

Others: None

Registers destroyed: All

RAM used: appSearchPage (two-bytes)

Remarks:

Example:

FindAppNumPages

Category: Memory

Description: Finds the number of 16K pages an application uses in archive memory

Inputs:

Registers: A = first page of application

Flags: None

Others: None

Outputs:

Registers: A = first page of application
C = number of 16K pages the application uses

Flags: None

Others: None

Registers destroyed: BC, DE

Remarks: If an application was not found on the given page, C will equal 0.

Example:

```
        IN          A,(memPageAPort)    ; gets the current memory
                                        ; page for app. Make sure
                                        ; this is on the first page
                                        ; of a multi-page
                                        ; application.
        B_CALL      FindAppNumPages     ; finds the total number of
                                        ; pages the application
                                        ; uses in archive memory.
        LD          A,C                 ; A = number of pages
```

For multi-page apps, create a routine that will reside on the first page of the application that will return the memory page.
i.e., Get_First_Page:

```
        IN          A,(memPageAPort)    ; get the memory page of
                                        ; the first application
                                        ; page.
        RET
```

FindAppDn

Category: Memory

Description: Searches for the next application in Flash ROM whose name is alphabetically less than the name in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = the name to find an application less than

If searching for all of the application names in descending alphabetical order then this name is either the previous one found or the initial name used to start the search.

To initialize the search to find the last application name alphabetically, set (OP1 + 1) = 0FEh.

Outputs:

Registers: None

Flags: CA = 1 if no application with a lesser name exists. The previous found is the first alphabetically.

CA = 0 if an application less than OP1 was found.

Others: OP1 = application name found if one exists.

Registers destroyed: All

RAM used: OP2, OP3, appSearchPage (two-bytes)

Remarks: No information about what ROM page the application resides on is returned. To get this information a **FindApp** needs to be done.

Example: A loop that finds all of the application names in descending order.

```
             B_CALL     ZerroOP1     ; initialize OP1 for 1st search
             LD         A,0FEh
             LD         OP1+1),A     ; set OP1 = name > any valid
                                     ; name

loop:
             B_CALL     FindAppDn    ; look for next lesser
                                     ; alphabetically
             JR         NC, loop     ; jump if found one, go look for
                                     ; next one
;
             RET
```

FindAppUp

Category: Memory

Description: Searches for the next application in Flash ROM whose name is alphabetically greater than the name in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = the name to find an application greater than

If searching for all of the application names in ascending alphabetical order then this name is either the previous one found or the initial name used to start the search.

To initialize the search set OP1 = all 0's with a system call to **ZeroOP1**.

Outputs:

Registers: None

Flags: CA = 1 if no application with a greater name exists. The previous found is the last alphabetically.

CA = 0 if an application greater than OP1 was found

Others: OP1 = application name found if one exists

Registers destroyed: All

RAM used: OP2, OP3, appSearchPage (two-bytes)

Remarks: No information about what ROM page the application resides on is returned. To get this information a **FindApp** needs to be done.

Example: A loop that finds all of the application names in ascending order.

```
              B_CALL     ZerroOP1     ; initialize OP1 for 1st search
loop:
              B_CALL     FindAppUp    ; look for next higher
                                      ; alphabetically
              JR         NC, loop     ; jump if found one, go look for
                                      ; next one
;
              RET
```

FindSym

Category: Memory

Description: Searches the symbol table structure for a variable.

This search routine is used to find variables that are not programs, AppVar, or Groups. See **ChkFindSym**.

This is used to determine if a variable is created and also to return pointers to both its symbol table entry and data storage area.
This will also indicate whether or not the variable is located in RAM or has been archived in Flash ROM.

Inputs:

Registers: (OP1 + 1) to (OP1 + 6) = variable name
See documentation on variable naming conventions.

Flags: None

Others: None

Outputs:

Registers: CA flag = 1 if symbol was not found
= 0 if symbol was found

If symbol is found, additional outputs are:
ACC lower 5 bits = data type
ACC upper 3 bits = system flags about variable. Mask via “AND” with a value of 1Fh to obtain data type only.
B = 0 if variable is located in RAM else variable is archived
B = ROM page located on
If variable is archived then its data cannot be accessed directly, it must be unarchived first.
HL = pointer to the start of the variables symbol table entry
DE = pointer to the start of the variables data area if in RAM

Flags: None

Others: OP1 = variable name

Registers destroyed: All

Remarks: This will not find system variables that are preallocated in system RAM such as Xmin, Xmax etc. Use **RclSysTok** to retrieve their values.

This will not find applications.

(continued)

FindSym *(continued)*

Example:

```
; Look for List L1 in the symbol table.
; If it exists and is archived then unarchive it and relook it up.
; If it does not exist create it as a real list of 10 elements.
Relook:
             LD        HL,L1name
             B_CALL    Mov9ToOP1        ; OP1 = variable name
             B_CALL    FindSym          ; look up
             JR        NC, VarCreated   ; jump if it exists
;
             LD        HL,10            ; size to create at data
             B_CALL    CreateRList
             PUSH      HL
             PUSH      DE               ; save during move
             B_CALL    OP4ToOP1         ; OP1 = name
             POP       DE               ; restore
             POP       HL
             JR        Done
;
VarCreated:
             LD        A,B              ; check for archived
             OR        A                ; in RAM ?
             JR        Z, DONE          ; yes
;
             B_CALL    Arc_Unarc        ; unarchive if enough RAM
             JR        Relook           ; look up pointers again in
                                        ; RAM now
DONE:
             RET
;
L1name:
             DB        ListObj,tVarLst,tL1,0
```

FixTempCnt

Category: Memory

Description: Resets pTempCnt back to a input value, and delete all temps with name counters greater than or equal to that value.

Inputs:

Registers: DE = value to pTempCnt to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: (pTempCnt) = DE

All temps created with pTempCnt >= input DE are deleted. For example, if input DE = 5 then temps with counter value 5 or greater $0500 will be deleted. $0600...

Registers destroyed: All

RAM used: pTempCnt

Remarks: See the Temporary Variables section in Chapter 2. Also, see the **CleanAll** routine.

Example:

FlashToRam

Category: Memory

Description: Copies bytes from Flash to RAM.

Inputs:

Registers: A = page of source (Flash)
HL = offset of source (Flash)
DE = RAM location of destination
BC = number of bytes to copy

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks: Certain pages in Flash cannot be copied. This routine will wrap to the next page if the offset = 8000h. A will be incremented to the next page, and HL will be reset to 4000h, and the copying will go on.

Example:

InsertMem

Category: Memory

Description: Inserts RAM into an existing variable.

This routine will only insert the RAM — it stays uninitialized and if the variable inserting into has a size field, it is NOT UPDATED. Updating must be done by the application.

A check for enough free RAM must be done by the application. This routine assumes the RAM is available.

Inputs:

Registers: HL = number of bytes of RAM to insert, no check is made for enough free RAM.

DE = point of insertion address — this cannot be the first byte of the variable's data — if it is, its symbol table entry will not have the correct pointer to the data.

Flags: None

Others: None

Outputs:

Registers: DE = intact, the point of insertion address

Flags: None

Others: RAM inserted into variable.

Registers destroyed: All

Remarks: See **DelMem**.

(continued)

InsertMem *(continued)*

Example: Insert 10 bytes at the beginning of an Application Variable.

```
             LD          HL,10             ; number bytes to insert
             B_CALL      ErrNotEnoughMem   ; error if 10 bytes are not
                                           ; free
;
             LD          HL,AppVarName
             RST         rMov9ToOP1        ; OP1 = name of AppVar
             B_CALL      ChkFindSym        ; look up in symTable
             JR          NC, Created       ; jump if it exists
;
             B_JUMP      ErrUndefined      ; error if not there
;
;  DE = pointer to size bytes of AppVar
;
Created:
             PUSH        DE                ; save pointer to start of
                                           ; size bytes of data
;
             INC         DE
             INC         DE                ; move DE past size bytes
;
             LD          HL,10             ; number bytes to insert
             B_CALL      InsertMem         ; insert the memory
;
             POP         HL                ; HL = pointer to size bytes
             PUSH        HL                ; save
;
             B_CALL      LdHLInd           ; HL = size of AppVar,
                                           ; number bytes
             LD          BC,10
             ADD         HL,BC             ; increase by 10, amount
                                           ; inserted
;
             EX          DE,HL
             POP         HL                ; pointer to size bytes
                                           ; location
;
             LD          (HL),E
             INC         HL
             LD          (HL),D            ; write new size.
;
;
AppVarName:  DB          AppVarObj,'AVAR',0
```

LdHLInd

Category: Memory

Description: Loads register pair HL with the contents of memory pointed to by (HL).

Inputs:

Registers: HL = address.
Flags: None

Others: None

Outputs:

Registers: H = (HL+1)

L = (HL)

Flags: None

Others: None

Registers destroyed: A, HL

Remarks:

Example: Same as:

```
LdHLInd:
             LD          A, (HL)
             INC         HL
             LD          H, (HL)
             LD          L, A
             RET
```

LoadCIndPaged

Category: Memory

Description: Reads a byte of data from any ROM page. Main use is for applications to read data from variables that are archived, without having to unarchive them to RAM first.

Inputs:

Registers: B = ROM page to read byte from
HL = address of byte on the ROM page,
(4000h–7FFFh)

Flags: None

Others: None

Outputs:

Registers: C = byte of data from input ROM page and Offset

Flags: None

Others: None

Registers destroyed: C

Remarks: B, HL are not changed. See the **LoadDEIndPaged** routine. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example: Read the byte of data from ROM page 0Ch, address 4006h.

```
             LD          B,0ch           ; ROM page
             LD          HL,4006h        ; offset
;
             B_CALL      LoadCIndPaged   ; C = byte
             RET
```

LoadDEIndPaged

Category: Memory

Description: Read two consecutive bytes of data from any ROM page. The main use of this routine is for applications to read data from variables that are archived, without having to unarchive them to RAM first.

Inputs:

Registers: B = ROM page of first of two bytes to read
HL = address of byte on the ROM page,
(4000h–7FFFh)

Flags: None

Others: None

Outputs:

Registers: E = first byte read
D = second byte read

Flags: None

Others: None

Registers destroyed: DE, C

Remarks: B, HL are set to the address of the second byte read. If the second byte of data is not on the same ROM page as the first, the switch to the next ROM page is handled. See the **LoadCIndPaged** routine. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example: Read two bytes of data from ROM page 0Ch, address 4006h.

```
        LD          B,0ch             ; ROM page
        LD          HL,4006h          ; offset
;
        B_CALL      LoadDEIndPaged    ; D = byte @ (4007h),
                                      ; E = byte @(4006h)
        RET
```

MemChk

Category: Memory

Description: Returns the amount of RAM currently available.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: HL = amount of RAM available, in bytes

Flags: None

Others: None

Registers destroyed: BC, HL

Remarks: If a system editor is open, this will always return 0 bytes available. System edits use all of free RAM during the edit.

The amount returned may be inaccurate if there are any temporary variables that are marked as dirty but not yet deleted. There are two ways/options to solve this:

- The routine **CleanAll** can be used to remove all temporary variables. This is fine as long as an application is not using temporary variables. Temporary variables are returned by the parser if the result is not RealObj or CplxObj, make sure that none are still in use.
- Use the routine **EnoughMem** instead, it will delete only temps that are marked dirty.

Example: Delete all temporary variables and then check if there is at least 100 bytes available.

```
             B_CALL      CleanAll        ; delete all temporary
                                         ; variables
             B_CALL      MemChk          ; HL = amount of mem free
;
             LD          DE,100
             OR          A               ; CA = 0
             SBC         HL,DE           ; if CA = 1 then less than 100
                                         ; bytes are available
             JR          C,Not_100       ; jump if < 100
```

PagedGet

Category: Memory

Description: Used for reading data from the archive with the Caching technique. This routine will return the next byte and also refill the cache when it is emptied.

A call to the **SetupPagedPtr** routine must be done once before using this routine to retrieve data from the archive.

Inputs:

Registers: None

Flags: None

Others: These are initially set by the **SetupPagedPtr** routine and are updated each time a call is made to the **PagedGet** routine. Applications do not need to modify these RAM locations.

(pagedPN) = current Flash page.
(pagedGetPtr) = current Flash address.

Outputs:

Registers: ACC = byte read

Flags: None

Others: None

Registers destroyed: ACC

Remarks: Crossing ROM page boundaries is handled. See the **SetupPagedPtr**, **LoadCIndPaged**, and **LoadDEIndPaged** routines. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example:

```
             LD          B,PageToGet
             LD          DE,AddressToGet
             B_CALL      SetupPagedPtr     ; setup paged get
;
             B_CALL      PagedGet          ; ACC = byte from archive
             LD          E,A               ; E = byte
;
             B_CALL      PagedGet          ;
;
             LD          D,A               ; DE = 2 bytes read from
                                           ; archive
```

RclGDB2

Category: Memory

Description: Recalls graph database.

Inputs:

Registers: A = tVarGDB

Flags: None

Others: OP1 = data base name
(chkDelPtr1) contains data pointer

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks: Acts exactly as the user controlled RclGDB command: Restores graph mode stored in the GDB and replaces all equation variables with those stored in the GDB and all range values with those stored in the GDB.

Example:

```
                                            ; Restore GDB2 if it exists:
             B_CALL      ZeroOP1            ; zero out OP1
             LD          HL,GDB2Name        ; name -> OP1
             LD          DE,OP1
             LD          BC,03
             LDIR
             B_CALL      FindSym            ; find & point to symbol.
             RET         C                  ; abort if does not exist.
             B_CALL      RclGDB2            ; restore graph data base.
....
GDB2Name:
             DB          GDBObj,tVarGDB,tGDB2  (008h,061h,001h)
```

RclN

Category: Memory

Description: Recalls the contents of variable N if it exists.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: System error if N does not exist.

OP1 = contents of N if RealObj
OP1/OP2 = contents of N if CplxObj

Registers destroyed: All

RAM used: OP1 – OP2

Remarks:

Example:

RclVarSym

Category: Memory

Description: Recalls the contents of variable A – Z or THETA.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of variable to recall

Outputs:

Registers: None

Flags: None

Others: System error if variable does not exist.

If a variable other than A – Z or THETA, then nothing is done.

OP1 = contents of variable if RealObj
OP1/OP2 = contents of variable if CplxObj

Registers destroyed: All

RAM used: OP1 – OP2

Remarks:

Example:

RclX

Category:	Memory
Description:	Recalls the contents of variable X if it exists.
Inputs:	
Registers:	None
Flags:	None
Others:	None
Outputs:	
Registers:	None
Flags:	None
Others:	System error if X does not exist. OP1 = contents of X if RealObj OP1/OP2 = contents of X if CplxObj
Registers destroyed:	All
RAM used:	OP1 – OP2
Remarks:	
Example:	

RclY

Category:	Memory
Description:	Recalls the contents of variable Y if it exists.
Inputs:	
Registers:	None
Flags:	None
Others:	None
Outputs:	
Registers:	None
Flags:	None
Others:	System error if Y does not exist. OP1 = contents of Y if RealObj OP1/OP2 = contents of Y if CplxObj
Registers destroyed:	All
RAM used:	OP1 – OP2
Remarks:	
Example:	

RedimMat

Category: Memory

Description: Redimensions an existing matrix.

Inputs:

Registers: HL = new dimension of matrix wanted

Flags: None

Others: OP1 = name of matrix

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All, iMathPtr1, insDelPtr

RAM used: OP1, OP3

Remarks: If not enough room, then a memory error will occur.

The space is allocated/deallocated. The pointers are adjusted accordingly. All the new elements are set to 0. The old values of the elements that are not removed are kept. A Matrix cannot be modified if it is archived.

Example:

```
             B_CALL      ZeroOP1      ; zero out OP1
             LD          HL,MatrixA
             LD          DE,OP1
             LD          BC,3
             LDIR                     ; load matrix name into OP1
             B_CALL      ChkFindSym   ; find matrix variable name
             JR          C, skip      ; if not found, skip over work
             LD          A, B         ;
             OR          A            ; see if archived
             JR          NZ, skip     ; skip if variable archived
             LD          HL,0505h
                                      ; redimensionalize matrix to 5x5
             B_CALL      RedimMat
skip:
             RET
MatrixA:     DB          MatObj,tVarMat,MatA
```

SetupPagedPtr

Category: Memory

Description: Initializes the process of reading data from the archive using the caching method.

The **PagedGet** routine is used to read data from the archive after this initialization routine is called.

Inputs:

Registers: Start address of the first byte of data to be read
B = ROM page of the first byte
DE = address of first byte, on the ROM page
(4000h–7FFFh)

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: These outputs are inputs to the **PagedGet** routine. An application should not change these values directly.

pagedCount = 0 on first call
pagedPN = current Flash page
pagedGetPtr = current Flash address

Registers destroyed: None

Remarks: See the **PagedGet** routine. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example:

```
             LD         B,PageToGet
             LD         DE,AddressToGet
             B_CALL     SetupPagedPtr     ; setup paged get
;
             B_CALL     PagedGet          ; ACC = byte from archive
             LD         E,A               ; E = byte
;
             B_CALL     PagedGet          ;
;
             LD         D,A               ; DE = 2 bytes read from
                                          ; archive
```

SrchVLstDn, SrchVLstUp

Category: Memory

Description: Searches the I/O var list in the backward/forward direction, next lower alphabetically, and by type in the following order:

PROGRAM,ProtPtrg	05h,06h
DATABASE	08h
PICTURE	07h
LIST,Clist	01h,0Dh
MATRIX	02h
YVARS	03h
AppVars	15h
Group	17h
WINDOW	0Bh
ZSTO	0Ch
TABLE RANGE	0Dh
REAL	00h
Cplx	0Ch
String	04h
Apps	14h

Inputs:

Registers: OP1 = last name and type found in variable format

Flags: inGroup, (IY + groupFlags) should be reset
inDelete, (IY + ioDelFlag) should be reset

Others: (varClass) should be set to 9 to search through the entire list.

Outputs:

Registers: HL = pointer to symbol table entry if found

Flags: CA = 0 if found
CA = 1 if did not find anything

Others: OP1 = var format of next variable if found

Registers destroyed: All registers

Remarks: This calls **FindAlphaUp/FindAlphaDn** to find variables within each variable type.

Example:

StMatEl

Category: Memory

Description: Stores an element to a matrix. Convert matrix or element to complex if necessary.

Inputs:

Registers: BC = column number
DE = row number

Flags: None

Others: OP1 = existing matrix variable name
FPST = value to store (real or complex)

Outputs:

Registers: None

Flags: graphDraw set if graph reference flag was on.

Others: OP1 = value originally on FP stack
FPST was popped, value no longer on FPST
Value was stored to the matrix

Registers destroyed: All

Remarks:

Example:

StoAns

Category: Memory

Description: Stores OP1 to Ans variable.

Inputs:

Registers: None

Flags: None

Others: OP1[,OP2] = value if real [complex]
Otherwise OP1 = name of variable that contains the data to store into Ans

Outputs:

Registers: None

Flags: None

Others: Data stored if possible
OP1[,OP2] = original contents if real[complex]
else OP1 = Ans variable name

Registers destroyed: FPS, OP1, OP2, OP4

Remarks: If input was a parser temporary ($P) variable, it is marked dirty (to be deleted by memory management).
A memory error occurs if there is not enough room to store the value.
Ans is the same system variable that is found by pressing [2nd] [Ans] on the calculator keyboard.

Use **RclAns** to recall the contents of Ans.

Example:

StoGDB2

Category: Memory

Description: Stores the current graph mode settings and equations into a system graph database variable.

Inputs:

Registers: None

Flags: None

Others: OP1 = graph database name to store to

Outputs:

Registers: None

Flags: None

Others: GDB created or modified

Registers destroyed: All

RAM used: (ioData) buffer used to store name temporarily.

Remarks: This creates the graph database if it did not exist already. If it did exist, it is resized to fit the size of the variables to be stored.

Example:

StoN

Category: Memory

Description: Stores OP1 to sequence variable n.

Inputs:

Registers: None

Flags: None

Others: OP1 = a real number, positive integer

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: This does not store to variable N.
This will store to the system variable n used in Sequence graphing.
To recall, see **RclN**.

Example:

StoOther

Category: Memory

Description: General purpose routine that stores data to user created variables that are not of type ProgObj, GDBObj, GroupObj, AppObj or PictObj.

Also, this routine should not be used to store to system variables such as Xmin.

Inputs:

Registers: None

Flags: None

Others: OP1 = name and type of variable to store to.
(OP1) = data type, followed by the name.
FPST = data to store if not storing to CplxObj
FPS1/FPST = data to store if storing to CplxObj

If the variable storing to is RealObj or CplxObj, then the data storing CANNOT be another variable. The FPS must contain the literal data stored.

If the variable storing to is not RealObj or CplxObj, then the data storing MUST be another variable. This variable can either be user created or a temporary variable returned by the parser after executing an expression.

If the variable storing to is already created, then it must reside in RAM and not the archive.

Outputs:

Registers: None

Flags: Both the graph and the table can be marked dirty if the variable stored to was used in a graph equation.

Others: Error if the data is not the correct type to be stored to the variable — for example, store list data to a matrix.

Error if the variable storing to is archived.

Error if not enough memory.

If no errors:

If the variable storing to was not created on input, this routine will create it.

Data stored to the variable.

OP1/OP2 = data that was stored.

The data is removed from the FPS.

Registers destroyed: All

StoOther *(continued)*

Remarks: See the **StoSysTok** routine. See Chapter 2 for Error Handlers and Floating Point Stack.

Example: Store list L1 to list L3.

```
             LD          HL,L1name
             B_CALL      Mov9ToOP1      ; OP1 = L1 name
             B_CALL      PushRealO1     ; FPST = L1 name
;
             LD          A,tL3          ; token for L3
             LD          (OP1+2),A      ; change OP1 to L3 name
;
             B_CALL      StoOther       ; store L1 -> L3
             RET
;
L1name:
             DB          ListObj,tVarLst,tL1,0
```

StoR

Category: Memory

Description: Stores OP1[,OP2] -> user variable R.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: Note that there is not a RclR routine, but one can be made by:

```
             B_CALL      RName         ; set OP1 to R name
             B_CALL      RclVarSym     ; do recall
```

Example:

```
; This sets R to 1:
             B_CALL      OP1Set1
             B_CALL      StoR          ; INIT R = 1
             RET
```

StoSysTok

Category: Memory

Description: Stores a value in OP1 to system variable specified by token number in the accumulator.

Inputs:

Registers: A = system variable token number
OP1 = real number to save

Flags: None

Others: None

Outputs:

Registers: OP1 = contents of system variable

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
                                ; Store -3 into Xmin
        B_CALL      OP1Set3     ; register OP1 = floating point 3
        B_CALL      InvOP1S     ; negate FP number in OP1,
                                ; OP1 = -3
        LD          A,XMINt     ; ACC = Xmin variable token value
        B_CALL      StoSysTok   ; store OP1 to Xmin,
```

StoT

Category: Memory

Description: Stores OP1[,OP2] to user variable T.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: Note that there is not a RclT routine, but one can be made by:

```
B_CALL      TName        ; set OP1 to T name
B_CALL      RclVarSym    ; do recall
```

Example:

```
;
                         ; This sets T to 0. :B_CALL
                         ; OP1Set0
B_CALL      StoT         ; INIT T = 0
RET
```

StoTheta

Category: Memory

Description: Stores OP1[,OP2] to user variable Theta.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: Note that there is not a RclTheta routine, but one can be made by:

```
             B_CALL      ThetaName     ; set OP1 to Theta name
             B_CALL      RclVarSym     ; do recall
```

Example:

```
;
; This sets Theta to 2...
             B_CALL      OP1Set2
             B_CALL      StoTheta      ; INIT Theta = 2
             RET
```

StoX

Category: Memory

Description: Stores OP1[,OP2] to user variable X.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: See **RclX** to recall contents of X.

Example:

```
;
                                        ; This sets X to 2:
             B_CALL      OP1Set2
             B_CALL      StoX           ; INIT X = 2
             RET
```

StoY

Category: Memory

Description: Stores OP1[,OP2] to user variable Y.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: See **RclY** to recall contents of Y.

Example:

```
;
                                        ; This sets Y to 2:
        B_CALL      OP1Set2
        B_CALL      StoY                ; INIT Y = 2
        RET
```

13 System Routines — Parser

BinOPExec

Category:	Parser
Description:	Executes functions that have two arguments as inputs.
Inputs:	
Registers:	ACC = function to execute (see table below)
Flags:	None
Others:	OP1 = second argument FPST = first argument (Floating Point Stack Top), see example
Outputs:	
Registers:	None
Flags:	None
Others:	OP1 = result
Registers destroyed:	All
Remarks:	Checks for valid argument types are done. The values pushed onto the FPS are removed. This entry point should only be used if direct access to a particular function is not available. It can also be used in cases of mixed argument types. Like the example below where a real is added to a list. Valid arguments can be obtained from the *TI–83 Plus Guidebook*.

(continued)

BinOPExec *(continued)*

Example:

```
.5 + L1
                LD        HL,Point5
                RST       rMov9ToOP1     ; OP1 = .5
                B_CALL    PushOP1        ; OP1 -> FPST, or OP1/OP2 is
                                         ; complex number
;
                LD        HL,L1name
                RST       rMov9ToOP1     ; OP1 = L1 name
;
                LD        A,OPAdd        ; function is addition
                B_CALL    BinOPExec      ; OP1 = result of .5 + L1
L1name:         DB        RListobj,tVarLst,tL1,0,0
```

BinOPExec equates and functions

Equate	Function	Equate	Function	Equate	Function
OPBal	bal(	OPSum	sum(	OPProd	prod(
OPBinCdf	binomcdf(	OPBinPdf	binompdf(	OPIrr	irr(
OPFinNom	>Nom(	OPFinEff	>Eff	OPFinDbd	dbd(
OPRandNrm	randNorm(	OPstDev	stdDev(	OPVariance	variance(
OPPrn	îPrn(	OPIntr	îInt(	OPRandBin	randBin(
OPNormalPdf	normalpdf(	OPINormal	invNorm(	OPNormal	normalcdf(
OPPoiPdf	poissonpdf(	OPPoiCdf	poissoncdf(	OPGeoCdf	geometcdf(
OPGeOPdf	geometpdf(	OPChiPdf	xýpdf(	OPTpdf	tpdf(
OPAdd	+	OPSub	-	OPMult	*
OPDiv	/	OPPower	^	OPXroot	xûy
OPEq	=	OPRound2	round(	OPConst	Fill(
OPAug	augment(	OPMax	max(	OPMin	min(
OPLcm	lcm(	OPGcd	gcd(	tEvalF	u(beg,end
tMedian	median(	tMean	mean(	OPRandInt	randInt(
OPAnd	and	OPOr	or	OPXor	xor
OPNcr	nCr	OPNpr	nPr	OPLt	<
OPLe	<=	OPGt	>	OPGe	>=
OPRand1	randM(	OPInstr	inString(	OPPxtst	Pxl-Test(
OPRtOPr	R>Pr(	OPRtOPo	R>Pθ(	OPPtorx	P>Rx(
OPPtoRy	P>Ry(				

Note: For tEvalF there are really three inputs but execution still goes through the entry point for two arguments. The Equation name needs to be pushed onto the FPS first, then the second argument and the third in OP1. This is only valid in Sequential graph mode.

The second argument is the start value.
The third argument is the end value.
A list of results is returned.

FiveExec

Category: Parser

Description: Executes functions that have five arguments as input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = fifth argument
FPST = fourth argument (pushed onto FPS fourth)
FPS1 = third argument (pushed onto FPS third)
FPS2 = second argument (pushed onto FPS second)
FPS3 = first argument (pushed onto FPS first)

Outputs:

Registers: None

Flags: None

Others: OP1 = result

Registers destroyed: All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.

This entry point should only be used if direct access to a particular function is not available.
Valid arguments can be gotten from the *TI–83 Plus Guidebook*.

(*continued*)

FiveExec *(continued)*

Example:

```
                                        ; fin(Y1, X, 2, 4, .5);
             LD       HL,Y1name
             RST      rMov9ToOP1        ; OP1 = Y1 name
             B_CALL   PushOP1           ; save to FPST;
             B_CALL   XName             ; OP1 = X var name
             B_CALL   PushOP1           ; push onto FPST, Y1 -> FPS1;
             B_CALL   OP1Set2           ; OP1 = 2
             B_CALL   PushOP1           ; push onto FPST, Y1 -> FPS2,
                                        ; X -> FPS1;
             B_CALL   OP1Set4           ; OP1 = 4
             B_CALL   PushOP1           ; ->FPST, Y1->FPS3, X->FPS2,
                                        ; 2->FPS1, 4->FPST;
             LD       HL,point5
             RST      rMov9ToOP1        ; OP1 = .5;
             LD       A,OPFmin  1       ; function is fMin(
             B_CALL   FiveExec          ; OP1 = result
Y1Name:      DB       EquObj,tVarEqu,tY1,0,0
Point5:      DB       0,80h,50h,0,0,0,0,0,0
```

FiveExec equates and functions

Equate	Function	Equate	Function	Equate	Function
OPSeq	seq(	OPQuad	fnInt(	OPFmin	fmin(
OPFmax	fMax(				

FourExec

Category: Parser

Description: Executes functions that have four arguments as input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = fourth argument
FPST = third argument (pushed onto FPS third)
FPS1 = second argument (pushed onto FPS second)
FPS2 = first argument (pushed onto FPS first)

Outputs:

Registers: None

Flags: None

Others: OP1 = result

Registers destroyed: All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.

This entry point should only be used if direct access to a particular function is not available.
Valid arguments can be obtained from the *TI–83 Plus Guidebook*.

(continued)

FourExec *(continued)*

Example:

```
                                           ; nDeriv(Y1, X, 2, .5);
            LD          HL,Y1Name
            RST         rMov9ToOP1         ; OP1 = Y1 name
            B_CALL      PushOP1            ; save to FPST;
            B_CALL      XName              ; OP1 = X var name
            B_CALL      PushOP1            ; push onto FPST, Y1 -> FPS1;
            B_CALL      OP1Set2            ; OP1 = 2
            B_CALL      PushOP1            ; push onto FPST, Y1 -> FPS2,
                                           ; X -> FPS1;
            LD          HL,point5
            RST         rMov9ToOP1         ; OP1 = .5;
            LD          A,OPDeriv81        ; function is nDeriv
            B_CALL      FourExec           ; OP1 = result
Y1Name:     DB          EquObj,tVarEqu,tY1,0,0
Point5:     DB          0,80h,50h,0,0,0,0,0,0
```

FourExec equates and functions

Equate	Function	Equate	Function	Equate	Function
OPNpv	npv(	OPNormal	normalcdf(	OPMltRadd	*row+(
OPSeq	seq(	OPQuad	fnInt(	OPDeriv81	nDeriv(
OPSolve	solve(	OPFmin	fMin(	OPFmax	fMax(
OPDf	Fcdf(				

ParseInp

Category: Parser

Description: Executes an equation or program stored in a variable.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of equation or program to execute

Outputs:

Registers: None

Flags: None

Others: If executed an equation, then OP1 and Ans contain the result.

If executed a program, then no result is returned.

Errors will be generated during parsing — to avoid them from being displayed, install an error handler before parsing.

Registers destroyed: All

Remarks: See the Parsing Function, Temporary Variables section in Chapter 2 for further information.

(continued)

ParseInp *(continued)*

Example: Parse the graph equation y1 and store the answer in Y. Install an error handler around the parsing and the storing to catch any errors.
RET CA = 0 if OK, else RET CA = 1.

```
             LD          HL,y1Name
             RST         rMov9ToOP1    ; OP1 = y1 name
;
; if an error while parsing go to this label
             AppOnErr    ErrorHan      ; error handler installed,
                                       ; (macro)
;
             B_CALL      ParseInp      ; execute the equation
;
; returns if no error
;
             B_CALL      CkOP1Real     ; check if RealObj
             JR          Z,storit      ; jump if it is real
;
             AppOffErr                 ; remove the error handler
;
; come here if any error was detected
; error handler is removed when the error occurred
;
ErrorHan:
             B_CALL      CleanAll      ; clean any temp vars created by
                                       ; parser
             SCF                       ; CA = 1 signals failure
             RET
;
storit:
             B_CALL      StoY          ; store to Y, RET if no error,
                                       ; else ErrorHan
;
             AppOffErr                 ; remove error handler
;
             B_CALL      CleanAll      ; clean any temp vars created by
                                       ; parser
;
             CP          A             ; CA = 0 for no error
             RET

y1Name:
             DB          EquObj,tVarEqu,tY1,0,0
```

RclSysTok

Category: Parser

Description: Recalls a value in system variable specified by token number in the accumulator to OP1.

Inputs:

Registers: A = system variable token number

Flags: None

Others: None

Outputs:

Registers: OP1 = contents of system variable

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

```
LD          A,XMINt
B_CALL      RclSysTok      ; OP1 = contents of Xmin
```

ThreeExec

Category: Parser

Description: Executes functions that have three arguments as input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = third argument
FPST = second argument (pushed onto FPS second)
FPS1 = first argument (pushed onto FPS first)

Outputs:

Registers: None

Flags: None

Others: OP1 = result

Registers destroyed: All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.

This entry point should only be used if direct access to a particular function is not available.

Valid arguments can be obtained from the *TI–83 Plus Guidebook*.

(continued)

ThreeExec *(continued)*

Example:

```
                                              ; row +([A],1,2)
              LD           HL,MatAName
              RST          rMov9ToOP1         ; OP1 = [A] name
              B_CALL       PushOP1            ; save to FPST;
              B_CALL       OP1Set1            ; OP1 = 1
              B_CALL       PushOP1            ; push onto FPST, mat name
                                              ; moves to FPS1;
              B_CALL       OP1Set2            ; OP1 = 2;
              LD           A,OPRAdd           ; function is row +
              B_CALL       ThreeExec          ; OP1 = result, a temp Matrix
                                              ; variable
MatAName:     DB           MatObj,tVarMat,tMatA,0,0
```

ThreeExec equates and functions

Equate	Function	Equate	Function	Equate	Function
OPPrn	îPrn(	OPIntr	îInt(	OPBinpdf	binompdf(
OPBincdf	binomcdf(	OPIrr	irr(	OPNpv	npv(
OPSum	sum(	OPProd	prod(	OPNormalPdf	normalpdf(
OPRandNrm	randNorm(	OPRandBin	randBin(	OPRandInt	randInt(
OPINormal	invNorm(	OPInstr	inString(	OPNormal	normalcdf(
OPDt	tcdf(	OPFpdf	Fpdf(	Opchi	x²cdf(
OPSubstr	sub(	OPDeriv81	nDeriv(	tEvalF	Eval(
OPRadd	row+(	OPRswap	rowSwap(	OPRmlt	row*(
OPMltRadd	*row+(	OPSolve	solve(		

Note: For tEvalF there are really four inputs but execution still goes through the entry point for three arguments. The Equation name needs to be pushed onto the FPS first, then the second argument and then third, and then the fourth in OP1. This is only valid in Sequential graph mode.

The second argument is the start value.
The third argument is the end value.
The fourth argument is the step size.

A list of results is returned.

UnOPExec

Category: Parser

Description: Executes functions that have one argument as the input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = argument

Outputs:

Registers: None

Flags: None

Others: OP1 = result

Registers destroyed: All

Remarks: This entry point should only be used if direct access to a particular function is not available.

It is also useful to use this entry point when arguments are not simply real numbers. See example below.

Valid arguments can be obtained from the *TI–83 Plus Guidebook*.

(continued)

UnOPExec *(continued)*

Example:

```
                                      ; sin(L1)
          LD         HL,L1name
          RST        rMov9ToOP1       ; OP1 = L1 name;
          LD         A,OPSin          ; function is addition
          B_CALL     UnOPExec         ; OP1 = result of sin(L1) a
                                      ; temp list variable
L1name:   DB         RListObj,tVarLst,tL1,0,0
```

UnOPExec equates and functions

Equate	Function	Equate	Function	Equate	Function
OPLog	log(	OPTenX	10^X(	OPLn	ln(
OPEtoX	e^X(	OPNot	not(	OPSin	sin(
OPAsin	sin-1(	OPCos	cos(	Pacos	cos-1(
OPTan	tan(	OPAtan	tan-1(	OPSinh	sinh(
OPAsinh	sinh-1(	OPCosh	cosh(	OPAcosh	cosh-1(
OPTanh	tanh(	OPAtanh	tanh-1(	OPInverse	recipricol
OPDet	det(	OPSqroot	Sqrt	OPSquare	^2
Opnegate	(-)	OPIpart	iPart(	OPFpart	fPart(
OPIntgr	int(	tEvalF	y#(value	OPConj	conj(
OPFact	!	OPAbs	abs(	OPIdent	identity(
OPTranspose	mat transpose	OPSum	sum(	OPProd	prod(
OPMin	min(	OPMax	max(	OPTofrac	>Frac
OPReal	real(	OPImag	Imag(	OPAngle	angle(
OPExpr	expr(	OPRound2	round(	OPLength	length(
OPCube	^3	OPCbrt	³√	OPDim	dim(
OPRad	^r	OPDeg	°	tMean	mean(
tMedian	median(	OPRef	ref(	OPRref	rref(
OPCumSum	cumSum(	OPNormalPdf	normalPdf(	OPInormal	invNorm(
OPDeltalst	-List(	OPBal	bal(	OPStdev	stdDev(
OPVariance	variance(	OPRand	rand		

Note: For tEvalF there are really two inputs but execution still goes through the entry point for one argument. The Equation name needs to be pushed onto the FPS first, and the second argument in OP1.

This is valid in all graph modes.

The second argument is the value to evaluate at.

14 System Routines — Screen

ForceFullScreen

Category: Screen

Description: Switches the TI–83 Plus to Full Screen mode if currently In Horizontal or Vertical split mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Graph is dirtied if mode switched.

Example:

15 System Routines — Statistics

DelRes

Category: Statistics

Description: Invalidates the statistic results.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Statistic result variables marked as undefined.
RegEq variable is deleted.

Registers destroyed: All

Remarks: Note that this routine does not set the graphDraw flag even if the stat result variable is used in a graph equation. This is a known problem.

Example:

```
B_CALL      DelRes       ; invalidate stat results
```

OneVar

Category: Statistics

Description: Executes one-variable statistics.

Inputs:

Registers: ACC = number of arguments input

Flags: No_Del_Stat, (IY + more_flags) = 1 if:
Stat results that are not associated with one-variable stats are not to be deleted when this routine executes.
Also no Min's, Max's, or Quartiles will be computed.
Otherwise: previous statistic results are cleared.

Others: If ACC = 1 then OP1 = data list name.
If ACC = 2 then OP1 = frequency list name.
FPST = data list name.
Dimensions must match if two arguments.

Outputs:

Registers: None

Flags: statANSDISP, (IY+statFlags) = 1

Others: If no errors then one-variable stat output variables are updated.

Registers destroyed: All

Remarks: If the input lists have a formula associated with them this routine will not execute it and update the list values. This must be done by the calling routine.

See **Find_Parse_Formula**.

Example: Run one-variable stats on data list L1 and freq. list L2.

```
             LD          HL,L1name
             RST         rMov9ToOP1       ; OP1 = L1
             RST         rPushRealO1      ; data ->FPST
;
             LD          HL,L2name
             RST         rMov9ToOP1       ; OP1 = L2
;
             B_CALL      OneVar           ; execute 1-variable stats
;
             RET
L1name:      DB          ListObj,tVarLst,tL1,0,0
L2name:      DB          ListObj,tVarLst,tL2,0,0
```

Rcl_StatVar

Category: Statistics

Description: Recalls a statistic result variable to OP1.

Inputs:

Registers: ACC = stat variable to recall token value. These are listed in the TI83plus.inc file.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = stat variable value, floating-point number

Registers destroyed: All but the ACC.

Remarks: The statistic variables are validated by running a regression or one/two variable statistic commands.

This routine does not check that the statistic variables are valid. Recalling one when not valid may result in random values.

Example: Recall statistic result variable X mean, assume statistic have been computed.

```
LD          A,tXMean      ; token value for XMean
B_CALL      Rcl_StatVar   ; recall contents to OP1
```

TwoVarSet

Category: Statistics

Description: Executes two-variable statistics and regressions.

Inputs:

Registers: ACC = number of arguments input. Must be at least 2.

B = type of calculation

- 0 = LinReg (a+bx)
- 1 = ExpReg
- 2 = LnReg
- 3 = PwrReg
- 4 = LinReg (ax+b)
- 5 = QuadReg
- 6 = CubicReg
- 7 = QuartReg
- 8 = Med-Med
- 9 = 2-Var Stats
- 19h = LinRegTTest
- 1Ah = Logistic
- 1Bh = In use for ANOVA
- 1Ch = SinReg

Flags: No_Del_Stat, (IY + more_flags) = 1 if:
Stat results that are not associated with one-variable stats are not to be deleted when this routine executes.
Also no Min's, Max's, or Quartiles will be computed.
Otherwise: previous statistic results are cleared.

Others: If ACC = 2 then OP1 = Y - data list name.
FPST = X – data list name.

If ACC = 3 then OP1 = frequency list name.
FPST = Y - data list name.
FPS1 = X – data list name.

If ACC = 4 then OP1 = Name of equation to store RegEq to.
FPST = frequency list name.
FPS1 = Y – data list name.
FPS2 = X – data list name
List dimensions must match.

Outputs:

Registers: None

Flags: statANSDISP, (IY+statFlags) = 1

Others: If no errors then stat output variables are updated. Arguments are removed from Floating Point Stack.

Registers destroyed: All

Remarks: This B_CALL is not available on OS version 1.12 or below. The application should check the OS version before calling this routine. See **GetBaseVer**.

If the input lists have a formula associated with them this routine will not execute it and update the list values. This must be done by the calling routine.

See **Find_Parse_Formula**.

Example: Calculate LinReg(ax+b) on x-list L1 and y-list L2, and store the results in Y1.

```
           LD      HL,L1name
           RST     rMov9ToOP1                    ; OP1 = L1
           RST     rPushRealO1                   ; data ->FPST
;
           LD      HL,L2name
           RST     rMov9ToOP1                    ; OP1 = L2
           RST     RPushRealO1                   ; FPS1 = L1; FPST = L2
;
           LD      HL, Y1name
           RST     rMov9ToOP1                    ; OP1 = Y1

           LD      A, 3                          ; 3 arguments
           LD      B, 4                          ; calc. LinReg(ax+b)
           B_CALL  TwoVarSet                     ; execute stats
;
           RES     statANSDISP, (IY+statFlags)   ; don't show results
           RET
L1name:    DB      ListObj,tVarLst,tL1,0,0
L2name:    DB      ListObj,tVarLst,tL2,0,0
Y1name:    DB      EquObj, tVarEqu, ty1, 0, 0
```

16 System Routines — Utility

AppInit

Category: Utility

Description: Sets system monitor vectors.

This routine is used by advanced applications to override the system monitor vector table. This routine should only be used by applications, not ASM programs.

Inputs:

Registers: HL points to monitor vector table.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 contains the variable name Ans.

Registers destroyed: All

Remarks: A common use of **AppInit** is to override the system's putaway vector. This allows the application to save its state or clean up any flags before shutting down if the user presses 2nd + OFF or silent link activity is detected during a system B_CALL **GetKey.**

Monitor vector table format:

```
VecTab:     DW          CXMainPtr
            DW          CXPPutAwayPtr
            DW          CXPutAwayPtr
            DW          CXRedispPtr
            DW          CXErrorEPPtr
            DW          CXSizeWindPtr
            DB          AppFlagsByte
```

The application must set all of these pointers to a label somewhere in the application. If a vector is not used, it must point to a RET statement.

If an application uses **AppInit** to change the system monitor vectors, it must perform a B_CALL **ReloadAppEntryVecs** before exiting and also in the application's putaway routine.

See also **ReloadAppEntryVecs**.

Example: See Chapter 2: "Entering and Exiting an Application Properly" for example putaway code.

AnsName

Category: Utility

Description: Loads OP1 with the variable name Ans.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** OP1 contains the variable name Ans.

Registers destroyed: All

Remarks:

Example:

```
        B_CALL    AnsName      ; load OP1 with Ans variable
```

Chk_Batt_Low

Category: Utility

Description: Check for low battery. Return Z = 1 if battery is low.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: Z = 1 if battery is low.

Z = 0 if battery is not low.

Flags: None

Others: None

Registers destroyed: All

Remarks: An application should check the battery condition before attempting to archive a variable. There is a risk of corrupting the archive if the attempt fails due to low batteries.

Example: Archive variable whose name is in OP1 if batteries are not low:

```
              B_CALL     Chk_Batt_Low   ; check battery level
              RET        Z              ; ret if low batteries
;
              B_CALL     ChkFindSym
              RET        C              ; return if variable does not exist
              LD         A,B            ; get archived status
              OR         A              ; if non zero then it is archived
                                        ; already
              RET        NZ             ; ret if archived
              AppOnErr   errorHand      ; install error handler
;
              B_CALL     Arc_Unarc      ; archives the variable
;
              AppOffErr                 ; remove error handler
errorHand:
              RET
```

ConvDim00

Category: Utility

Description: Converts floating-point number in OP1 to a two-byte value and compares that value with an input two-byte value.

Inputs:

Registers:

HL = two-byte test value

Flags:

None

Others:

OP1 = floating-point value, must be a positive integer < 10,000

Outputs:

Registers: If no error on the input:
A = LSB hex value of OP1
DE = entire hex value of OP1

Flags: None

Others: None

Registers destroyed: All

Remarks:

Example: Test OP1 = positive integer < or = 400:

```
LD          HL,400d      ; test value
B_CALL      ConvDim00
```

CpHLDE

Category: Utility

Description: Non destructives compare of registers HL and DE.

Inputs:

Registers: HL = two-byte value
DE = two-byte value

Flags: None

Others: None

Outputs:

Registers: HL, DE intact

Flags: CA = 1 if DE > HL
Z = 1 if HL = DE
CA = 0 if HL > DE

Others: None

Registers destroyed: None

Remarks:

Example:

```
        B_CALL      CpHLDE
```

DisableApd

Category: Utility

Description: Turns off Auto Power Down feature.

Inputs:

- **Registers:** None
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** None
- **Flags:** None
- **Others:** apdAble, (IY + apdFlags) is reset

Registers destroyed: None

Remarks: Applications should re-enable APD before exiting. See **EnableApd**.

Example:

EnableApd

Category: Utility

Description: Turns on Auto Power Down.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: The TI–83 Plus will now power down if not used for approximately four minutes.

Example:

EOP1NotReal

Category: Utility

Description: Tests object in OP1 to be a real data type. If it is not, then jump to the system error DATA TYPE.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: None

Flags: None

Others: Error if not OP1 — it does not have the data type RealObj.

Registers destroyed: A

Remarks:

Example:

Equ_or_NewEqu

Category: Utility

Description: Sees if A = EquObj or NewEquObj type.

Inputs:

Registers: A = type, can have flags set

Flags: None

Others: None

Outputs:

Registers: A = type with flags reset

Flags: Z set if A = EquObj or NewEquObj type

Others: None

Registers destroyed: None

Remarks:

Example:

```
; see if ACC is EquObj or NewEquObj
Equ_or_NewEqu::
             AND         1Fh
             CP          EquObj
             RET         Z
             CP          NewEquObj
             RET
```

GetBaseVer

Category: Utility

Description: Returns current operating system version number.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: A = major version number
B = minor version number

Flags: None

Others: None

Registers destroyed: A, B

Remarks:

Example: For Operating system 1.00: A = 1, B = 0.

GetSysInfo

Category: Utility

Description: Return nine bytes of system information, including current speed.

Inputs:

Registers: HL = RAM location to save system information.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: (HL)…(HL+8) contain system information.

Registers destroyed: All

RAM used:

Remarks: This B_CALL is not available on TI-83 Plus version 1.12 and earlier. The calling routine needs to check the software version before performing this B_CALL. See **GetBaseVer.**

This routine returns nine bytes of data representing various aspects of system operation:

Btye	
00	Boot code revision # (Major)
01	Boot code revision # (Minor)
02	Hardware revision # (00 is TI-83 Plus, NZ if not)
03	Lsn = Current Speed
03	Bit 4 reset if TI-83 Plus; set if TI-83 Plus Silver Edition
04	Device code default
05	Reserved
06	Reserved
07	Reserved
08	Reserved

(continued)

Example: Determine if running fast or slow.

```
                B_CALL      GetBaseVer        ; OS version in A, B
                CP          2                 ; check major version
                JR          NC, above112      ; if 2.x, then > 1.12
;               CP          1
                JR          NZ, MustBeSlow    ; if 0.x, then < 1.12
                LD          A, B              ; major version = 1
                CP          13                ; check minor version
                JR          NC, above112      ; C if minor version < 13
MustBeSlow:
                XOR         A                 ; set Z to show slow
                JR          Done
Above112:                                     ; later than 1.12
                LD          HL, OP1
                B_CALL      GetSysInfo
                LD          A, OP1+3
                AND         0Fh
Done:
```

GetTokLen

Category: Utility

Description: Return the number of characters in a token's string.

Inputs:

Registers: DE = pointer to either a one or two byte token

Flags: None

Others: None

Outputs:

Registers: A = number of characters in the token's string
HL = address of string in Flash ROM.

Flags: None

Others: None

Registers destroyed: All

RAM used:

Remarks:

Example: Find the number of characters in the 'Sin(' token string.

```
             LD          DE,tSin     ; Sin( token
             B_CALL      GetTokLen   ; ACC = 4, the length of 'Sin('

;
```

Get_Tok_Strng

Category: Utility

Description: Copy a token's string to OP3 and return the number of characters in the string.

Inputs:

Registers: HL = pointer to either a one or two byte token

Flags: None

Others: None

Outputs:

Registers: A = number of characters in the token's string
BC = also contains the number of characters in the token's string
HL = address of OP3, location the string was copied to

Flags: None

Others: String copied to RAM, starting at OP3

Registers destroyed: All

RAM used: OP3 – OP3 + (length of string)

Remarks:

Example: Find the number of characters in the 'Sin(' token string.

```
             LD        A,tSin         ; Sin( token
             LD        (OP1),A
             LD        HL,OP1         ; pointer to token
;
             B_CALL    Get_Tok_Strng  ;
```

IsA2ByteTok

Category: Utility

Description: Determines if token in A is a one or two byte token.

Inputs:

Registers: A = First byte of token

Flags: None

Others: None

Outputs:

Registers: None

Flags: Z = 1 if A is the first byte of a two byte token

Z = 0 if A is not a two byte token.

Others: None

Registers destroyed: None

Remarks: The two byte token identifiers are: t2ByteTok, tVarStrng, tGFormat, tVarSys, tVarOut, tVarGBD, tVarPict, tVarEqu, tVarLst, and tVarMat.

Example:

JForceCmdNoChar

Category: Utility

Description: Exits the Application and returns to the home screen.

This should not be used to exit an application if the TI–83 Plus system monitor is closing the application due to link activity or turning off.

This routine will be the used in most applications to Close the application and return control to the TI–83 Plus system.

Before an application jumps to this entry point it must make certain the systems monitor vectors are set to the Application loader context.

See Entering and Exiting an Application Properly.

Inputs:

Registers: None

Flags: None

Others: Monitor vectors should be set to the Application loader.

Outputs:

Registers: None

Flags: None

Others: The home screen is given control.

Registers destroyed: All

Remarks: Only use a B_JUMP with this entry point.

This can be used by an application anytime — the return stack does not need to be at any certain level. This routine will set the stack level back to a safe level.

ASM PROGRAMS SHOULD NOT USE THIS ROUTINE TO EXIT BACK TO THE SYSTEM.

Example: Set the monitor vectors to the Application loader and exit the application and return control to the home screen.

```
Exit_App:
             B_CALL     ReloadAppEntryVecs  ; load the monitor vectors
                                            ; to App loader
;
             B_JUMP     JForceCmdNoChar     ; exit the app and
                                            ; initiate home screen
```

JForceGraphKey

Category: Utility

Description: Exits the Application and returns to the graph screen with a key to be executed in the graph screen.

This should not be used to exit an application if the TI–83 Plus system monitor is closing the application due to link activity or turning off.

This routine will be the used in most applications to Close the application and return control to the TI–83 Plus system.

Before an application jumps to this entry point it must make certain the systems monitor vectors are set to the Application loader context.

See Entering and Exiting an Application Properly.

Inputs:

Registers: ACC = key to execute in the graph screen

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Only use a B_JUMP with this entry point.

This can be use by an application anytime — the return stack does not need to be at any certain level. This routine will set the stack level back to a safe level.

ASM PROGRAMS SHOULD NOT USE THIS ROUTINE TO EXIT BACK TO THE SYSTEM.

Example: Set the monitor vectors to the Application loader and exit the application and enter trace mode.

```
Exit_App:
             B_CALL     ReloadAppEntryVecs  ; load the monitor vectors
                                            ; to App loader
;
             LD         A,kTrace
             B_JUMP     JForceGraphKey      ; exit the app enter trace
                                            ; mode
```

JForceGraphNoKey

Category: Utility

Description: Exits the Application and returns to the graph screen.

This should not be used to exit an application if the TI–83 Plus system monitor is closing the application due to link activity or turning off.

This routine will be the used in most applications to close the application and return control to the TI–83 Plus system.

Before an application jumps to this entry point it must make certain the systems monitor vectors are set to the Application loader context.

See Entering and Exiting an Application Properly.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Only use a B_JUMP with this entry point.

This can be use by an application anytime — the return stack does not need to be at any certain level. This routine will set the stack level back to a safe level.

ASM PROGRAMS SHOULD NOT USE THIS ROUTINE TO EXIT BACK TO THE SYSTEM.

Example: Set the monitor vectors to the Application loader and exit the application and give control to the graph context.

```
Exit_App:
             B_CALL      ReloadAppEntryVecs  ; load the monitor vectors
                                             ; to App loader
;
             LD          A,kTrace
             B_JUMP      JForceGraphNoKey    ; exit the app
```

MemClear

Category: Utility

Description: Clears a memory block (to 00h's).

Input:

Registers: BC = number of bytes in block
HL = address of first byte in memory block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block cleared

Registers destroyed: A, BC, DE, HL

Remarks: BC must be > 1

Example: TBD

MemSet

Category:	Utility
Description:	Sets a memory block to a given value.
Inputs:	
Registers:	A = value to set all bytes in memory block BC = number of bytes in block HL = address of first byte in memory block
Flags:	None
Others:	None
Outputs:	
Registers:	None
Flags:	None
Others:	Memory block set
Registers destroyed:	BC, DE, HL
Remarks:	BC must be > 1
Example:	TBD

Mov7B, Mov8B, Mov9B, Mov10B, Mov18B

Category: Utility

Description: Copies a short memory block where X = MovXB, where X is the number of bytes.

Inputs:

Registers: HL = start of source block
DE = start of destination block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Block starting at original HL copied to area starting at original DE.

Registers destroyed: BC, DE, HL

Remarks:

Example:

Mov9OP1OP2

Category: Utility

Description: Copies a block of 18 bytes of RAM/ROM to OP1/OP2, with the first nine-bytes to OP1 and the second nine-bytes to OP2.
This is most commonly used to copy a complex element of either a list or matrix to OP1/OP2, skipping the 10th and 11th bytes of OP1.

Inputs:

Registers: HL = pointer to start of 18 bytes to copy

Flags: None

Others: None

Outputs:

Registers: DE = DE + 18

Flags: None

Others: First nine-bytes OP1 and first nine-bytes of OP2 contain the 18 bytes copied.

Registers destroyed: All but ACC

Remarks:

Example: Copy the first element of complex list L1 to OP1/OP2:

```
             LD          HL,L1name
             RST         rMov9ToOP1     ; OP1 = L1 name
             B_CALL      FindSym        ; look up, DE = pointer to data
             EX          DE,HL          ; HL = pointer to data
             INC         HL
             INC         HL             ; HL = pointer to 1st element
;
             B_CALL      Mov9OP1OP2     ; OP1 = real part, OP2 = image
                                        ; part, of element 1
             RET
```

Mov9OP2Cp

Category: Utility

Description: Copies a floating-point number from RAM/ROM to OP2 and compares it to a floating-point number in OP1.

Inputs:

Registers: HL = pointer to floating point to copy to OP2

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Z = 1 if OP1 = OP2
Z = 0, CA = 1: OP1 < OP2
Z = 0, CA = 0: OP1   OP2

Others: OP1 = intact
OP2 = floating-point number copied

Registers destroyed: All

Remarks: Both OP1 and the float copied to OP2 are preserved.

Example: Copy the first element of real list L1 to OP2 and compare it to a floating-point number in OP1.

```
             LD          HL,L1name
             RST         rMov9ToOP1     ; OP1 = L1 name
             B_CALL      FindSym        ; look up, DE = pointer to data
             EX          DE,HL          ; HL = pointer to data
             INC         HL
             INC         HL             ; HL = pointer to 1st element
;
             B_CALL      Mov9OP2Cp      ; copy element to OP2 and
                                        ; compare to OP1
             RET
```

Mov9ToOP1

Category: Utility

Description: Copies nine-bytes of RAM/ROM to OP1.

Inputs:

Registers: HL = pointer to the nine-bytes to copy

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 contains the nine-bytes

Registers destroyed: All but ACC

Remarks:

Example:

```
B_CALL     Mov9ToOP1
```

Mov9ToOP2

Category: Utility

Description: Copies nine-bytes of RAM/ROM to OP2.

Inputs:

Registers: HL = pointer to the nine-bytes to copy

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 contains the nine-bytes

Registers destroyed: All but ACC

Remarks:

Example:

```
B_CALL     Mov9ToOP2
```

MovFrOP1

Category: Utility

Description: Copies OP1 (nine bytes) to another RAM location.

Inputs:

Registers: DE = pointer to destination of move

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: HL = OP1 + 9
DE = DE + 9
OP1 copied to (DE)

Registers destroyed: All but ACC

Remarks:

Example:

NZIf83Plus

Category: Utility

Description: Returns status if calculator is TI-83 Plus or not.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: NZ = status if TI-83 Plus

Z = status if TI-83 Plus Silver Edition

Others: None

Registers destroyed: None

Remarks: This B_CALL is not available on TI-83 Plus version 1.12 or earlier. The calling routine must check the software version before performing this B_CALL. This routine is not as intrusive as **GetSysInfo** if all you need to know is if the calculator is an earlier edition of TI-83 Plus.

See **GetBaseVer, GetSysInfo**

Example: Return NZ if running on TI-83 Plus

```
             B_CALL     GetBaseVer        ; OS version in A, B
             CP         1                 ; check major version
             JR         C, MustBe83Plus   ; if 0.x, then < 1.13
             JR         NZ, Above112      ; if 2.x, then > 1.12
             LD         A, B              ; major version = 1
             CP         13                ; check minor version
             JR         NC, above112      ; C if minor version < 13

MustBe83Plus:
             RET

Above112:                                 ; later than 1.12
             B_CALL     NZIf83Plus
             RET
```

OP1ExOP2, OP1ExOP3, OP1ExOP4, OP1ExOP5, OP1ExOP6, OP2ExOP4, OP2ExOP5, OP2ExOP6, OP5ExOP6

Category: Utility

Description: Exchanges 11-byte contents of OP(x) with OP(y).

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(X) = former contents of OP(Y)
OP(Y) = former contents of OP(X)

Registers destroyed: A, BC, DE, HL

Remarks: Combinations Available:

(y) (x)	OP1	OP2	OP3	OP4	OP5	OP6
OP1		X	X	X	X	X
OP2				X	X	X
OP3						
OP4						
OP5						X
OP6						

Example:

```
; Exchange contents of OP2 and OP4
B_CALL     OP2ExOP4
```

OP1ToOP2, OP1ToOP3, OP1ToOP4, OP1ToOP5, OP1ToOP6, OP2ToOP1, OP2ToOP3, OP2ToOP4, OP2ToOP5, OP2ToOP6, OP3ToOP1, OP3ToOP2, OP3ToOP4, OP3ToOP5, OP4ToOP1, OP4ToOP2, OP4ToOP3, OP4ToOP5, OP4ToOP6, OP5ToOP1, OP5ToOP2, OP5ToOP3, OP5ToOP4, OP5ToOP6, OP6ToOP1, OP6ToOP2, OP6ToOP5

Category: Utility

Description: Copies 11 bytes from OP(x) to OP(y).

Inputs:

Registers: None

Flags: None

Others: OP(x)

Outputs:

Registers: None

Flags: None

Others: OP(y) = former contents of OP(x)

Registers destroyed: BC, DE, HL

Remarks: Combinations Available:

Dest(y) Source(x)	OP1	OP2	OP3	OP4	OP5	OP6
OP1		X	X	X	X	X
OP2	X		X	X	X	X
OP3	X	X		X	X	
OP4	X	X	X		X	X
OP5	X	X	X	X		X
OP6	X	X			X	

Example:

```
B_CALL      OP1ToOP3
```

PosNo0Int

Category: Utility

Description: Checks if OP1 is a positive non-zero integer floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Z = 1 if OP1 = positive non 0 integer
Z = 0 if non integer or negative or 0

Others: None

Registers destroyed: ACC

Remarks:

Example:

PutAway

Category: Utility

Description: Force application to be put away.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Application is terminated.

Registers destroyed: ALL

Remarks: Applications should not use this routine for normal exit code. Applications should only use this entry point as part of putaway code in “Stand-Alone with Putaway Notification” mode. See Chapter 2: “Entering and Exiting an Application Properly”.

Example:

```
AppPutAway:
;
;
; Application gets itself ready for terminating by cleaning any system flags
; or saving any information it needs to.
;
             RES        plotLoc, (IY+plotFlags)    ; draw to display & buffer
             RES        textWrite, (IY+sGrFlags)   ; small font written to
                                                   ; display
; This next call resets the monitor control vectors back to the App Loader
;
             B_CALL     ReloadAppEntryVecs         ; App Loader in control of
                                                   ; monitor
;
             LD         (IY+textFlags),0           ; reset text flags
;
; This next call is done only if application used the Graph Backup Buffer
;
             B_CALL     SetTblGraphDraw
;
; Need to check if turning off or not, the following flag is set when
; turning off:
;
             BIT        MonAbandon,(IY+monFlags)   ; turning off ?
             JR         NZ, TurningOff             ; jump if yes
;
; if not turning off then force control back to the home screen
;
; note: this will terminate the link activity that caused the application
```

```
; to be terminated.
;
             LD          A, iall                    ; all interrupts on
             OUT         (intrptEnPort), A
             B_CALL      LCD_DRIVERON               ; turn on LCD
             SET         onRunning, (IY+onFlags)    ; on interrupt running
             EI                                     ; enable interrupts

             B_JUMP      JForceCmdNoChar            ; force to home screen
;
TurningOff:
             B_JUMP      Putaway                    ; force App loader to do its
                                                    ; put away
```

RclAns

Category: Utility

Description: Recalls answer to OP1[,OP2] or at least set up pointers to it.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1[,OP2] if real [or complex]

Registers destroyed: AF, BC, DE, HL,

Remarks: Entire code:

```
CALL      AnsName      ; see these routines for more
                       ; info
JP        RclVarSym    ; see these routines for more
                       ; info
```

AnsName puts the name of Ans into
OP1 = 00h,tAns,00h,00h,.....00h
= 00h,072h,00h,00h,.....00h

RclVarSym will recall the contents of the variable to OP1 if it is real, to OP1 and OP2 if the variable is complex and otherwise leaves the name as is in OP1 and returns HL as the symbol table pointer and DE as the data pointer as in **ChkFindSym**.

Example:

```
B_CALL    RclAns       ; This example presumes that
                       ; you already know that Ans is
                       ; a Real number.
LD        A,9          ; display up to 8 digits
B_CALL    DispOP1A     ;
```

ReloadAppEntryVecs

Category: Utility

Description: Sets the system monitor vector table to the Application loader context.

This routine is used by advanced applications that override the system monitor vector table. This routine should be called by the application just before exiting.

This routine should only be used by applications, not ASM programs.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Monitor system vectors are now set to the application loader.

Registers destroyed: All

Remarks:

Example: Assume we have an application that overrode the monitor vectors and our application is exiting because the user pressed the [Quit] key.

```
ChkForQuit:
             CP          kQuit                 ; quit key?
             JR          NZ,notQuit            ; jump if no
;
             B_CALL      ReloadAppEntryVecs    ; restore monitor to
                                               ; application loader
             B_JUMP      JForceCmdNoChar       ; switch to the home
                                               ; screen
;
```

SetExSpeed

Category: Utility

Description: Set execution speed to fast or slow.

Inputs:

Registers: A = 0 to set slow speed (6Mhz)

A = 1 to set 15Mhz

A = FF to set fastest future speed

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: Flag register modified

Remarks: This B_CALL is not available on TI-83 Plus version 1.12 or earlier. The calling routine must check the software version before performing this B_CALL. This routine is not as intrusive as **GetSysInfo** if all you need to know is if the calculator is an earlier edition of TI-83 Plus.

See **GetBaseVer, GetSysInfo**

This routine can be called on a TI-83 Plus unit running software version 1.13 and higher, but will not effect the operating speed of that unit.

On the TI-83 Plus Silver Edition, the operating system will set the speed back to fast once the application or assembly program returns, regardless of any settings changed. An exception to this is that the error handler will leave the speed setting intact in the event a GoTo is desired.

Some system routines such as the IO utilities may set slow speed for certain operations. These routines will restore the current speed upon completion. Other routines, such as **JforceCmdNoChar** force fast speed. Normally an application will not execute these routines except on completion.

(continued)

Example: Set fast speed if running on 1.13 or higher.

```
            B_CALL      GetBaseVer       ; OS version in A, B
            CP          2                ; check major version
            JR          NC, Above112     ; if 2.x, then > 1.12
            CP          1                ; if 0.x, then < 1.12
            JR          NZ, Below112     ; major version = 1
            LD          A, B             ;
            CP          13               ; check minor version
            JR          C, Below112      ; C if minor version < 13

Above112:
            LD          A, 0FFh          ; set fastest speed possible
            B_CALL      SetExSpeed
            JR          Done
Below112:                                ; earlier than 1.12
            ;
            ;
            ;
Done:
```

SetXXOP1

Category: Utility

Description: Sets OP1 equal to a floating-point integer between 0 and 99.

Inputs:

Registers: ACC = integer value to set OP1 equal to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point integer between 0 – 99

Registers destroyed: All

RAM used: OP1

Remarks: No error checking is done for invalid input.

Example: Set OP1 = 75.

```
LD          A,75
B_CALL      SetXXOP1       ; OP1 = floating point 75
```

SetXXOP2

Category: Utility

Description: Sets OP2 equal to a floating-point integer between 0 and 99.

Inputs:

Registers: ACC = integer value to set OP2 equal to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 = floating-point integer between 0 – 99

Registers destroyed: All

RAM used: OP2

Remarks: No error checking is done for invalid input.

Example: Set OP2 = 75.

```
LD          A,75
B_CALL      SetXXOP2       ; OP2 = floating point 75
```

SetXXXXOP2

Category: Utility

Description: Sets OP2 equal to a floating-point integer between 0 and 65535.

Inputs:

Registers: HL = integer value to set OP2 equal to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 = floating-point integer between 0 – 65535

Registers destroyed: All

RAM used: OP2

Remarks:

Example: Set OP2 = 7523.

```
LD          HL,7523
B_CALL      SetXXXXOP2     ; OP2 = floating point 7523
```

StoRand

Category: Utility

Description: Initializes random number seeds on OP1 value.

Inputs:

Registers: None

Flags: None

Others: OP1 = real number 0e0 ... < 1E9

Outputs:

Registers: None

Flags: None

Others: OP1 = same value as unmodified input.

Registers destroyed: All

RAM used: OP1, OP2, OP6

Remarks: Storing a 0 to the seed will reinitialize the random number generator to its original state from the factory.

The input value in OP1 must be a real number, but it does not have to fall within the specified range. If it does not, it will be modified (exponent reduced, sign changed, and truncated) to fit in the range.

Example:

StrCopy

Category: Utility

Description: Copy a null-terminated string in memory.

Inputs:

Registers: HL = starting address of source string

DE = starting address of destination

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers Destroyed: A, DE, HL

Remarks: This is like a C language StrCpy() function.

When complete:

? HL is left pointing to the null terminator of the source string.

? DE is left pointing to the null terminator of the destination string.

Example:

StrLength

Category: Utility

Description: Returns the length of a zero (0) terminated string residing in RAM.

Inputs:

- **Registers:** HL = pointer to start of zero terminated string, in RAM
- **Flags:** None
- **Others:** None

Outputs:

- **Registers:** BC = length of string, not including terminating 0
- **Flags:** None
- **Others:** None

Registers destroyed: BC

Remarks:

Example:

17 System Routines — Miscellaneous

ConvOP1

Category: Miscellaneous

Description: Converts a floating-point number in OP1 to a two-byte hexadecimal number in DE.

Inputs:

Registers: OP1 = floating-point number

Flags: None

Others: None

Outputs:

Registers: A = LSB hex value
DE = entire hex value
If OP1 exponent > 3 error

Flags: None

Others: None

Registers destroyed:

Remarks:

Example:

R Reference List — System Routines

A

B

C

F

K

L

M

N

O

P

R

S

T

U

V

X

Y

G Glossary

ACC	ACC stands for accumulator.
Address	A number given to a location in memory. You can access the location by using that number, like accessing a variable by using its name.
APD™	**A**utomatic **P**ower **D**own™.
API	**A**pplication **P**rogrammer's **I**nterface—the set of software services available to an application and the interface for using them.
Applet	A stand-alone application, usually in Flash ROM, with the associated security mechanisms in place. See ASAP.
Archive memory	Part of Flash ROM. You can store data, programs, or other variables to the user data archive, which cannot be edited or deleted inadvertently.
ASAP	**As**sembly **A**pplication **P**rogram—a RAM-resident application.
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange—a convention for encoding characters, numerals in a seven or eight-bit binary number.
Assembler	A program that converts source code into machine language that the processor can understand, similar to compilers used with high-level languages.
Assembly language	A low-level language used to program microprocessors directly. Z80 assembly language can be used on the TI–83 Plus to write programs that execute faster than programs written in TI–BASIC. See Chapter 3 for advantages and disadvantages.
Binary	A system of counting using 0's and 1's. The first seven digits and the decimal equivalents are: 0 — 0 1 — 1 10 — 2 11 — 3 100 — 4 101 — 5 110 — 6 111 — 7 See also Hexadecimal.
Bit	Short for binary digit — either 1 or 0. In computer processing and storage, a bit is the smallest unit of information handled by a computer and is represented physically by an element such as a single pulse sent through a circuit or a small spot on a magnetic disk capable of storing either a 1 or a 0. Considered singly, bits convey little information a human would consider meaningful. In groups of eight, however, bits become the familiar bytes used to represent all types of information, including the letters of the alphabet and the digits 0 through 9. (Microsoft Encarta '97)

Boot (code) A small amount of software that resides in ROM; therefore, it cannot be overwritten or erased. Boot code is required for the calculator to manage the installation of new base code.

Byte A unit of information consisting of 8 bits, the equivalent of a single character, such as a letter. 8 bits equal {0-255} and there are 256 letters in the extended ASCII character set. Standard ASCII uses a 7-bit value (0-127), thus there are 128 characters.

Calculator serial number An electronic serial number that resides in a calculator's Flash memory. It is used to uniquely identify that calculator.

Character A single letter, digit, or symbol. **Q** is a character. **4** is a character. **%** is a character. **123** and **yo** are not characters.

Compiled language A language that must be compiled before you can run the program. Examples include C/C++ and Pascal.

Compiler A compiler translates high-level language source code into machine code.

D-Bus A proprietary communication bus used between calculators, the Calculator-Based Laboratory™ (CBL™) System, the Calculator-Based Ranger™ (CBR™) and personal computers.

Decimal The standard (base 10) system of counting, as opposed to binary (base 2) or hexadecimal (base 16).

E-Bus Enhanced D-Bus.

Entry points Callable locations in the base code corresponding to pieces of code that exhibit some coherent functionality.

Execute To run a program or carry out a command.

Flash-D A PC program that is the integration of a PC downloader application with a calculator application. When the Flash-D program is executed on the PC, the calculator application is transferred to the calculator via a TI-GRAPH LINK™ cable.

Freeware Programs or databases that an individual may use without payment of money to the author. Commonly, the author will copyright the work as a way of legally insisting that no one change it prior to getting approval. Commonly, the author will issue a license defining the terms under which the copyrighted program may be used. With freeware, there is no charge for the license.

Garbage collection A procedure that automatically determines what memory a program is no longer using and recycles it for other use. This is also known as **automatic storage (or memory) reclamation**.

TI-GRAPH LINK™ An optional accessory that links a calculator to a personal computer to enable communication.

Group certificate Used to identify several calculators as a single **unit**. This allows the group of calculators, or **unit**, to be assigned a new program license using only one certificate (instead of requiring a new unique unit certificate for each

	calculator in the group). The group certificate must be used in conjunction with the unit certificate.
Hexadecimal	Base 16 system, which is often used in computing. Counting is as follows: {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.
High-level language	Any programming language that resembles English. This makes it easier for humans to understand. Unfortunately, a computer cannot understand it unless it is compiled into machine language. See also low-level language. Examples of high-level languages are C/C++, Pascal, FORTRAN, COBOL, Ada, etc.
IDE	**I**ntegrated **D**evelopment **E**nvironment.
Immediate	An addressing mode where the data value is contained within the instruction instead of being loaded from somewhere else. For example, in LD A, 17, **17** is an immediate value. In LD A, B, the value in **B** is not immediate, because it is not written into the code.
Interpreted language	A language that is changed from source code to machine language in real-time. Examples are BASIC (for the PC and the TI version, TI–BASIC) and JavaScript. Interpreted languages are often much simpler, which helps beginners get started and allows experienced programmers to write code quickly. Interpreted languages, however, are restricted in their capability, and they run slower.
Instruction	A command that tells the processor to do something, for example**, add two numbers** or **get some data from the memory**.
I/O port	An input/output interface from the calculator to the external world. It allows communication with other units, CBL™ and CBR™, and personal computers.
LCD port	An output port that drives LCD display device for use on overhead projectors. Available on the teacher's ViewScreen™ calculator only.
Low-level language	Any programming language that does not look like English but is still to be understandable by people. It uses **words** like **add** to replace machine language instructions like **110100**. See also high-level language.
Machine language	Any programming language that consists of 1's and 0's (called binary), which represents instructions. A typical machine instruction could be 110100, which means **add two numbers together**.
Mac Link	MacIntosh resident link software that can communicate with the calculator.
Marked Dirty	The graph is marked as needing to be updated. The next system routine that will affect the graph contents will cause the system to regraph all of the equations selected thereby making the graph clean.
Memory	Memory is where data is stored. On the TI–83 Plus, the main memory is the built-in 32K of RAM. This memory is composed of one-byte sections, each with a unique address.
Microprocessor	See processor.
Operating System (OS)	The software included with every new calculator. OS contains the features that are of interest to customers, as well as behind-the-scenes functionality

	that allows the calculator to operate and communicate. In our newer calculators, the OS is in Flash ROM, so the user can electronically upgrade it with OS.
Processor	A large computer chip that does most of the work in a computer or calculator. The processor in the TI–83 Plus is the Zilog Z80 chip.
Program	A program is a list of instructions written in sequential order for the processor to execute.
Program ID number	An ID number assigned to a particular software program. It is used during the program authentication process to match the program licenses in a unit/group certificate to the program being downloaded into the calculator.
Program license	A digital license purchased by a customer allowing the customer to authorize the download/execution of a particular software program to a specific calculator. The program licenses are assigned to and listed in the calculator unit/group certificates.
Register	A register is high-speed memory typically located directly on the processor. It is used to store data while the processor manipulates it. On the TI–83 Plus there are 14 registers.
Register pair	Two registers being used as if they were one, creating a 16-bit register. Larger numbers can be used in registered pairs than in single registers. The register pairs are AF, BC, DE, and HL. Register pairs are often used to hold addresses.
Run (Busy) Indicator	When the TI–83 Plus is calculating or graphing, a vertical moving line is displayed as a busy indicator in the top-right corner of the screen. When you pause a graph or a program, the busy indicator becomes a vertical moving dotted line.
SDK	Software Development Kit—a set of tools that allow developers to write software for specific platforms.
Shareware	Sometimes called **User Supported** or **Try Before You Buy** software. Shareware is not a particular kind of software, it is a way of marketing software. Users are permitted to try the software on their own computer systems (generally for a limited period of time) without any cost of obligation. Payment is required if the user has found the software to be useful or if the user wishes to continue using the software beyond the evaluation (trial) period. Payment of the registration fee to the author will bring the user a license to continue using the software. Most authors will include other materials in return for the registration fee—like printed manuals, technical support, bonus or additional software, or upgrades. Shareware is commercial software, fully protected by copyright laws. Like other business owners, shareware authors expect to earn money from making their software available. In addition, by paying, the user may then be entitled to additional functions, removal of time limiting or limits on use,

	removal of so-called **nag** screens, and other things as defined in the documentation provided by the program's author.
Signed application	An application that has been digitally signed by TI.
Silent link	Computer-initiated request—protocol version of communications between the computer and the calculator.
Software owner's account	An account set-up in the TI database listing all of the program licenses owned by a particular customer or group. The account also allows the software owner to assign a particular program to a specific calculator.
Source code	A text file containing the code, usually in a high-level or low-level programming language.
TASM	Table Assembler—a PC program that assembles source code for the Z80 and other processors. This has been one of the more popular tools for developing calculator ASM programs.
TI–BASIC	The programming language commonly used on the TI–83 Plus. It is the language that is used for PROGRAM variables. Its main drawback is that these programs run slower, since it is an interpreted language, rather than a compiled language.
TI signature	A digital signature placed on secured documents/files such as unit and group certificates, as well as software program images.
User Data Archive	Storage for user data in the Flash ROM. In some cases, the user can choose between the amount of Flash for applets versus user data.
Unique owner ID	An alphanumeric ID assigned to the owner of a software owner's account as a way of authorizing access to this account. Examples of the ID are mother's maiden name, social security number, birth date, etc.
Unit certificate	A digital certificate signed by TI that lists all of the program and group licenses issued to a specific calculator. The unit certificate also includes owner ID information and the calculator serial number.
Z80	This processor is used in the TI–83 Plus. Z80 assembler is the language used to program the Z80 chip.
ZDS	Zilog Development Studio—a tool used by developers to write software for Zilog products. This tool can be used to develop TI–83 Plus calculator applications and ASM programs.