AXE PARSER COMMANDS

VERSION 1.2.2

Axe Parser

Omega 1.2.2

This is a mere guide of the commands, you will not find a list of every possible use(s), this list is for Axe 1.2.2 only, as such some commands will not work with previous version as described with in, further you are warned that not every code behaves the same on different models of Z80, if you don't follow the syntax you will harm your calculator. For further information please visit, http://goo.gl/6W9iBX for the latest updates and info.

System	
Command	**Description**
_	Spaces are ignored in most situations. They mainly just help for code organization and readability.
:	The colon separates 2 statements in the same expression. Enter ends the expression, closing all unclosed parenthesis.
.	The period is a single line comment. Whatever follows will be ignored until the next newline.
...	Start or end of a multi-line comment ignoring everything in between. Must be the first characters on the line.
...If CONST	A conditional comment. Only starts a multi-line comment if the constant is zero.
...!If CONST	A conditional comment. Only starts a multi-line comment if the constant is non-zero.
...Else	If the previous conditional was a comment, the next comment block will not be a comment and vise-versa.
DiagnosticOn	Turns on the run indicator (marching ants). Program will display "done" after finishing.
DiagnosticOff	Turns off the run indicator. Program will not display "done" after finishing.
Full	Full speed mode is activated if supported, making most code run 2.5 times faster on newer calculators. Returns 0 if not supported.
Full^r	Suspends the memory delays on non-83+ calculators for the remainder of the program. This will not properly restore the original setting if Return^r is used. Do not execute in a loop!
Full^{rr}	Suspends the Horiz mode if it was set for the remainder of the program. Also clears the home screen upon execution and when the program returns. This will not properly restore the original setting if Return^r is used. Do not execute in a loop!
Normal	Full speed mode is deactivated.
Pause EXP	Pause for the given amount of time. A one second pause is a value of roughly 1800 at normal speed or 4500 in full speed mode.
getKey	Returns the last key pressed or zero if no keys were pressed. Its just like the BASIC getkey, but with different codes.
getKey^r	Pauses until a key or key combination is pressed and returns the key code. These are different codes than the normal getkeys.
getKeys	Calls the OS keypad scanning function. That's the getKey token followed by an s (or an S).
getKey(KEY)	Returns 1 if the key is held down this instant and 0 otherwise.
getKey(0)	Returns a non-zero number if any key is held down and 0 otherwise.
getKey(41)	Returns 1 if the ON key is held down this instant and 0 otherwise.
Asm(HEX)	Native assembly code written in hexadecimal is inserted at the current position.
prgmNAME	The code from the external program is parsed as if it completely replaced this command in the main program. (Similar to the C++ "include")
#Axiom(NAME) ***AsmComp()***	The Axiom assembly library becomes useable in the program. No quotes needed, just type the name of the appvar case sensitive.
#Icon(HEX) ***identity()***	Tells the parser to replace the default icon with the new icon. The icon must be 64 hex characters long.

#Realloc(PTR) ***real()***	Moves the variable buffer to another location in ram. If the field is left blank, the default location is restored.
#ExprOn ***ExprOn***	Requests that all following code be optimized for speed.
#ExprOff ***ExprOff***	Requests that all following code be optimized for size.

Screen and Buffer	
DispGraph **DispGraph(BUF)**	Draws the main buffer, or specified buffer, to the screen.
DispGraphClrDraw **DispGraphClrDraw(BUF)**	Same as DispGraph:ClrDraw, but as fast as just DispGraph.
DispGraphRecallPic **DispGraphRecallPic(BUF1,BUF2)**	Same as DispGraph:RecallPic, but faster.
DispGraph^{r} **DispGraph(BUF1,BUF2)^{r}**	Draws the main and back buffer, or specified buffers, to the screen to create 3 color grayscale.
DispGraphClrDraw^{r} **DispGraphClrDraw(BUF1,BUF2)^{r}**	Same as DispGraph^{r}:ClrDraw^{rr}, but as fast as just DispGraph^{r}.
DispGraph^{rr} **DispGraph(BUF1,BUF2)^{rr}**	Draws the main and back buffer, or specified buffers, to the screen to create 4 color grayscale.
DispGraphClrDraw^{rr} **DispGraphClrDraw(BUF1,BUF2)^{rr}**	Same as DispGraph^{rr}:ClrDraw^{rr}, but as fast as just DispGraph^{rr}.
ClrDraw **ClrDraw^{r}** **ClrDraw(BUF)**	Erases the main buffer, back buffer, or specified buffer to white pixels.
ClrDraw^{rr}	Erases both the front and back buffers.
DrawInv **DrawInv^{r}** **DrawInv(BUF)**	The colors on the main buffer, back buffer, or specified buffer are inverted.
StoreGDB	Takes a screenshot and saves it to the main buffer.
StorePic	Copies the main buffer to the back buffer.
RecallPic	Copies the back buffer to the main buffer.
Horizontal+	The main buffer, back buffer, or specified buffer is shifted right by 1 pixel. White pixels are shifted in.

Horizontal+r **Horizontal+(BUF)**	
Horizontal- **Horizontal-r** **Horizontal-(BUF)**	The main buffer, back buffer, or specified buffer is shifted left by 1 pixel. White pixels are shifted in.
Vertical+ **Vertical+r** **Vertical+(BUF)**	The main buffer, back buffer, or specified buffer is shifted down by 1 pixel. New pixels are not shifted in, that row remains the same.
Vertical- **Vertical-r** **Vertical-(BUF)**	The main buffer, back buffer, or specified buffer is shifted up by 1 pixel. New pixels are not shifted in, that row remains the same.
ZInterval EXP	Sets the LCD z-address, which sets a hardware vertical rotation of the LCD contents EXP rows upwards.
Shade()	Returns the LCD contrast value of the operating system before the program started.
Shade(EXP)	Sets the LCD contrast. 0 is lightest, 63 is darkest.

Control Blocks	
If EXP:code1:End	If EXP is true, code1 will be executed.
If EXP:code1:Else:code2:End	If EXP is true, then only code1 is executed. Otherwise, only code2 is executed.
ElseIf EXP	Can be used inside If blocks. If EXP is true, the block's code is executed and then skips to the end of the block.
!If EXP:code1:End	If EXP is false, code1 will be executed.
!If EXP:code1:Else:code2:End	If EXP is false, then only code1 is executed. Otherwise, only code2 is executed.
Else!If EXP	Can be used inside If blocks. If EXP is false, the block's code is executed and then skips to the end of the block.
While EXP:code1:End	The expression is checked first. If its true, code1 will be executed over and over until its false.
Repeat EXP:code1:End	The expression is checked first. If its false, code1 will be executed over and over until its true.
For(EXP):code1:End	Executes code1 exactly EXP times. 0 is not a valid number of iterations and will act as 65536. Goto cannot be used inside this structure.
For(EXP)r:code1:End	Executes code1 exactly EXP%256 times. 0 is not a valid number of iterations and will act as 256. Goto cannot be used inside this structure.
For(VAR,EXP1,EXP2):code1:End	The variable VAR is initialized with EXP1. If its greater than EXP2, the loop ends. Otherwise code1 is executed and VAR is incremented by 1.
DS<(VAR,MAX):code1:End	The variable VAR is decreased by 1. If its 0, code1 is executed and the variable resets back to MAX.

	Otherwise the code is skipped.
EndIf EXP	In loops, it will exit the loop if EXP is true. But it works just like a regular End otherwise.
End!If EXP	In loops, it will exit the loop if EXP is false. But it works just like a regular End otherwise.

Labels and Subroutines	
Lbl LBL	Creates a label at the current position.
Goto LBL	Jumps to the label.
Goto (EXP)	Jumps to a label defined by an expression.
sub(LBL,...)	Loads 0 to 6 arguments to the r_1 through r_6 variables respectively. Then the subroutine is called. Most subroutines should end with a Return.
sub(LBL^r,...)	Same as above except the argument variables it uses are saved before the subroutine is called and restored when it returns.
LBL(...)	Same as sub(but with a simpler-to-type format. Also takes 0 to 6 arguments.
(EXP)(...)	Same as above but with a subroutine defined by an expression.
Return	Returns from a subroutine. If not in a subroutine, the program will end.
Return^r	Emergency exits the program from within any number of nested calls.
ReturnIf EXP	Returns if EXP is true. Otherwise continues execution.
Return!If EXP	Returns if EXP is false. Otherwise continues execution.
$_L$LBL	Returns the address of the label.
λ(EXP) *log()*	Creates a subroutine that computes an expression in terms of r_1 through r_6. Returns the address of that subroutine.
Z-Test(EXP,LBL,...)	Checks the expression. If its 0, it will jump to the first label, if its 1, it will jump to the second, etc. If out of range, it will skip the command.

Basic Math	
CONST	Returns the constant.
VAR	Returns the variable. Uppercase A through Z, Θ, r_1 through r_6, and X_{1T} through Y_{6T} are variables as well as custom named variables.
VAR^r	Returns the low byte of the variable.
VAR^{rr}	Returns the big-endian value of the variable.
^{o}VAR	Returns the address of the variable in memory.
EXP→VAR	Stores the expression into the variable.

EXP→VAR'	Stores the expression into the low byte of the variable.
EXP→VAR''	Stores the expression into the variable in big-endian.
'CHAR'	The ASCII constant as an integer.
-EXP	Returns the negative of the expression. That's a negative sign, not a minus sign!
EXP1+EXP2	EXP2 is added to EXP1.
EXP1-EXP2	EXP2 is subtracted from EXP1.
EXP++	The variable or memory location is incremented by 1. Memory locations need curly brackets.
EXP--	The variable or memory location is decremented by 1. Memory locations need curly brackets.
EXP1*EXP2	EXP1 is multiplied by EXP2.
EXP^2	EXP is multiplied by itself.
EXP1^EXP2	EXP1 is taken modulo EXP2.
EXP1/EXP2	EXP1 is divided by EXP2.
EXP1=EXP2	Returns 1 if EXP1=EXP2 or 0 otherwise.
EXP1≠EXP2	Returns 1 if EXP1≠EXP2 or 0 otherwise.
EXP1>EXP2	Returns 1 if EXP1>EXP2 or 0 otherwise. This is an unsigned comparison.
EXP1≥EXP2	Returns 1 if EXP1≥EXP2 or 0 otherwise. This is an unsigned comparison.
EXP1<EXP2	Returns 1 if EXP1<EXP2 or 0 otherwise. This is an unsigned comparison.
EXP1≤EXP2	Returns 1 if EXP1≤EXP2 or 0 otherwise. This is an unsigned comparison.
EXP1 and EXP2	Returns the bitwise AND of the lower 8 bits of the expressions. For logical AND, see the ? command.
EXP1 or EXP2	Returns the bitwise OR of the lower 8 bits of the expressions. For logical OR, see the ?? command.
EXP1 xor EXP2	Returns the bitwise XOR of the lower 8 bits of the expressions.
not(EXP)	Returns the bitwise complement of the lower 8 bits.
√(EXP)	Returns the square root of the expression.
sin(EXP)	Returns the sine of the expression. One period is [0,256] and the value returned ranges from -127 to 127.
cos(EXP)	Returns the cosine of the expression. One period is [0,256] and the value returned ranges from -127 to 127.
tan^{-1}(DX,DY)	Returns the angle of a path that has moved DX to the right and DY up. One period is [0,256] and both DX and DY must be in the range [-512,512]
min(EXP1,EXP2)	Returns the minimum of the 2 expressions.
max(EXP1,EXP2)	Returns the maximum of the 2 expressions.
rand	Returns a random 16-bit number.
COND?EXP	If the condition is true, EXP is evaluated. Can be used as an and short circuit operator like && in C syntax.
COND?EXP1,EXP2	If the condition is true, EXP1 is evaluated. Otherwise, EXP2 is evaluated.
COND??EXP	If the condition is false, EXP is evaluated. Can be used as an or short circuit operator like || in C syntax.

COND??EXP1,EXP2	If the condition is false, EXP1 is evaluated. Otherwise, EXP2 is evaluated.

Advanced Math	
ɛHEX	The hexadecimal number as an integer. That prefix is the scientific notation "E".
πBIN	The binary number as an integer.
ᵀTOKEN	The 1 or 2 byte token as an integer. That prefix is the transpose symbol ᵀ.
INT.DEC	The non-integer decimal number as an 8.8 fixed point number. Maximum 3 decimal places allowed.
EXP1==EXP2	Returns 1 if EXP1=EXP2 or 0 otherwise. Same as the single equals operator.
EXP1≠≠EXP2	Returns 1 if EXP1≠EXP2 or 0 otherwise. Same as the single not-equals operator.
EXP1>EXP2	Returns 1 if EXP1>EXP2 or 0 otherwise. This is a signed comparison.
EXP1≥≥EXP2	Returns 1 if EXP1≥EXP2 or 0 otherwise. This is a signed comparison.
EXP1<<EXP2	Returns 1 if EXP1<EXP2 or 0 otherwise. This is a signed comparison.
EXP1≤≤EXP2	Returns 1 if EXP1≤EXP2 or 0 otherwise. This is a signed comparison.
EXP1*^EXP2	Computes the high order 16 bits of the unsigned multiplication of EXP1 and EXP2.
EXP1EXP2**	EXP1 is multiplied by EXP2, where both are signed 8.8 fixed point values.
EXP^{2r}	The signed 8.8 fixed point value EXP is multiplied by itself.
EXP1//EXP2	EXP1 is divided by EXP2, where both are signed values.
EXP1/*EXP2	EXP1 is divided by EXP2, where both are signed 8.8 fixed point values.
EXP^{-1}	Computes the reciprocal of the signed 8.8 fixed point value.
EXP1-EXP2	Returns the bitwise AND of the 16-bit expressions. This is a plot style token.
EXP1+EXP2	Returns the bitwise OR of the 16-bit expressions. This is a plot style token.
EXP1▫EXP2	Returns the bitwise XOR of the 16-bit expressions. This is a plot style token.
not(EXP)^{r}	Returns the bitwise complement of the 16-bit value.
EXP1*e*EXP2	Gets the EXP2-th bit of the lower 8 bits of EXP1. Unlike assembly, the leftmost bit (high order) is bit 0 and the rightmost bit (low order) is bit 7. That's the Euler's constant e.
EXP1*ee*EXP2	Gets the EXP2-th bit of the 16-bit value EXP1. Unlike assembly, the leftmost bit (high order) is bit 0 and the rightmost bit (low order) is bit 15. That's the Euler's constant e.
e^(EXP)	Returns 2 to the power of the expression (modular).
ln(EXP)	Returns the base 2 logarithm of the expression, or -1 if undefined.
abs(EXP)	Returns the absolute value of the expression.
√(EXP)^{r}	Returns the square root of the 8.8 fixed point expression.
Select(EXP1,EXP2)	Returns EXP1, saving its value while EXP2 is evaluated.

Drawing	
Pxl-On(X,Y) **Pxl-On(X,Y)r** **Pxl-On(X,Y,BUF)**	The pixel at (X,Y) becomes black on the main buffer, back buffer, or specified buffer respectively.
Pxl-Off(X,Y) **Pxl-Off(X,Y)r** **Pxl-Off(X,Y,BUF)**	The pixel at (X,Y) becomes white on the main buffer, back buffer, or specified buffer respectively.
Pxl-Change(X,Y) **Pxl-Change(X,Y)r** **Pxl-Change(X,Y,BUF)**	The pixel at (X,Y) inverts color on the main buffer, back buffer, or specified buffer respectively.
pxl-Test(X,Y) **pxl-Test(X,Y)r** **pxl-Test(X,Y,BUF)**	Returns 1 if pixel is black and 0 if pixel is white at (X,Y) on the main buffer, back buffer, or specified buffer respectively.
***[W]*HLine(Y)** ***[W]*HLine(Y)r** ***[W]*HLine(Y,BUF)** ***[W]*HLine(Y,X1,X2)** ***[W]*HLine(Y,X1,X2)r** ***[W]*HLine(Y,X1,X2,BUF)**	Draws a black, white, or inverted horizontal line from X1 to X2 (0 to 95 if unspecified) at the given Y coordinate on the main buffer, back buffer, or specified buffer respectively.
***[W]*VLine(X)** ***[W]*VLine(X)r** ***[W]*VLine(X,BUF)** ***[W]*VLine(X,Y1,Y2)** ***[W]*VLine(X,Y1,Y2)r** ***[W]*VLine(X,Y1,Y2,BUF)**	Draws a black, white, or inverted vorizontal line from Y1 to Y2 (0 to 95 if unspecified) at the given X coordinate on the main buffer, back buffer, or specified buffer respectively.
***[W]*Line(X1,Y1,X2,Y2)** ***[W]*Line(X1,Y1,X2,Y2)r** ***[W]*Line(X1,Y1,X2,Y2,BUF)**	Draws a black, white, or inverted line from point (X1,Y1) to (X2,Y2) on the main buffer, back buffer, or specified buffer respectively.
[W]*Rect(X,Y,W,H)** ***[W]*Rect(X,Y,W,H)r** ***[W]*Rect(X,Y,W,H,BUF)*ref()	Draws a filled black, white, or inverted rectangle with its upper left corner at (X,Y), a width of W, and a height of H on the main buffer, back buffer, or specified buffer respectively.
RectI(X,Y,W,H) **RectI(X,Y,W,H)r** **RectI(X,Y,W,H,BUF)*rref()***	Draws a filled inverted rectangle with its upper left corner at (X,Y), a width of W, and a height of H on the main buffer, back buffer, or specified buffer respectively.

***[W]*Circle(X,Y,R)** ***[W]*Circle(X,Y,R)"** ***[W]*Circle(X,Y,R,BUF)**	Draws a black, white, or inverted circle centered at (X,Y) and with a radius of R on the main buffer, back buffer, or specified buffer respectively.

Sprites	
Pt-On(X,Y,PIC) **Pt-On(X,Y,PIC)"** **Pt-On(X,Y,PIC,BUF)**	The 8x8 sprite pointed to is drawn using OR logic at (X,Y) to the main buffer, back buffer, or specified buffer respectively.
Pt-Off(X,Y,PIC) **Pt-Off(X,Y,PIC)"** **Pt-Off(X,Y,PIC,BUF)**	The 8x8 sprite pointed to is drawn using overwrite logic at (X,Y) to the main buffer, back buffer, or specified buffer respectively.
Pt-Change(X,Y,PIC) **Pt-Change(X,Y,PIC)"** **Pt-Change(X,Y,PIC,BUF)**	The 8x8 sprite pointed to is drawn using XOR logic at (X,Y) to the main buffer, back buffer, or specified buffer respectively.
Pt-And(X,Y,PIC) **Pt-And(X,Y,PIC)"** **Pt-And(X,Y,PIC,BUF)** *Plot3()*	The 8x8 sprite pointed to is drawn using AND logic at (X,Y) to the main buffer, back buffer, or specified buffer respectively.
pt-Get(X,Y) **pt-Get(X,Y)"** **pt-Get(X,Y,BUF)** **pt-Get(X,Y,BUF,TEMP)** *Plot2()*	Returns a temporary pointer to a copy of the 8x8 sprite at (X,Y) on the main buffer, back buffer, or specified buffer respectively. The copy will be placed at TEMP if specified.
Pt-Mask(X,Y,PIC) *Plot1()*	The 8x8 grayscale sprite (2 layers) pointed to is drawn to both buffers at (X,Y). Areas clear on both layers are transparent and the other combinations are 3-level grayscale.
Pt-Mask(X,Y,PIC)" **Pt-Mask(X,Y,PIC,BUF)"** *Plot1()*	The 8x8 masked sprite (2 layers) pointed to is drawn to the main buffer or specified buffer at (X,Y). Areas clear on both layers are transparent and the other combinations are white, invert, and black.
Bitmap(X,Y,BMP) **Bitmap(X,Y,BMP)"** **Bitmap(X,Y,BMP,BUF)** **Bitmap(X,Y,BMP,BUF,MODE)** *Tangent()*	Draws a bitmap to (X,Y) on the main buffer, back buffer, or specified buffer respectively. The bitmap data should have in order: width (1 byte), then height (1 byte), then the rows of the image padded with zeros to the nearest byte. Mode 0 is "Pt-On" logic and Mode 1 is "Pt-Change" logic. Mode 0 is used if unspecified.
rotC(PIC) *ShadeNorm()*	A copy of the 8x8 sprite pointed to is rotated clockwise 90 degrees. Returns a pointer to that new rotated sprite. Cannot be used recursively.
rotCC(PIC) *Shade_t()*	A copy of the 8x8 sprite pointed to is rotated counter-clockwise 90 degrees. Returns a pointer to that new rotated sprite. Cannot be used recursively.

flipV(PIC) *ShadeX2()*	A copy of the 8x8 sprite pointed to is flipped vertically. Returns a pointer to that new flipped sprite. Cannot be used recursively.
flipH(PIC) *ShadeF()*	A copy of the 8x8 sprite pointed to is flipped horizontally. Returns a pointer to that new flipped sprite.

Text	
ClrHome	Erases the screen and text shadow and moves the cursor to the upper left corner.
Disp PTR	The string that is pointed to is displayed at the cursor position. The cursor moves with the string. If it reaches the end of the screen, it will loop around to the next line.
Disp EXP▶Dec	The number is displayed as a decimal at the cursor position. The cursor is then advanced 5 spaces.
Disp EXP▶Char▸*Frac*	The ASCII character is displayed at the cursor position. The cursor is advanced 1 space. A new line is added if it hits the edge.
Disp EXP▶Tok▸*DMS*	The 1- or 2-byte token pointed to is displayed at the cursor position. The cursor is advanced. A new line is added if it hits the edge.
Disp *i*	The cursor moves to the next line down. This is the imaginary, not lowercase 'i'.
Output(X)	The cursor moves to the cursor position (X/256,X%256).
Output(X,Y)	The cursor moves to the cursor position (X,Y).
Output(X,Y,...)	The cursor moves to the cursor position (X,Y) and whatever follows is displayed at that position.
Text EXP	The text pointed to is drawn at the current pen location. See Fix command for drawing details.
Text EXP▶Dec	The number is drawn as a decimal at the current pen location. See Fix command for drawing details.
Text EXP▶Char▸*Frac*	The ASCII character is drawn at the current pen location. See Fix command for drawing details.
Text PTR▶Tok▸*DMS*	The 1- or 2-byte token pointed to is drawn at the current pen location. See Fix command for drawing details.
Text(X)	The text pen moves to the position (X%256,X/256).
Text(X,Y)	The text pen moves to the position (X,Y).
Text(X,Y,...)	The text pen moves to the position (X,Y). Whatever comes next becomes the Text command.
EXP▶Hex▸*Rect*	Converts the number to hexadecimal and returns the pointer to that string.
Fix 0	Small size font. Calculator should exit in this mode if changed!
Fix 1	Large size font.
Fix 2	Normal colored font. Calculator should exit in this mode if changed!
Fix 3	Inverted font.
Fix 4	Text is drawn directly to the screen. Calculator should exit in this mode if changed!
Fix 5	Text is drawn to the buffer.
Fix 6	Automatic scrolling on last line of display. Calculator should exit in this mode if changed!

Fix 7	No scrolling on last line of display
Fix 8	Lowercase alpha is turned off.
Fix 9	Lowercase alpha is turned on.
input	Prompts for an input string just like BASIC then returns a pointer to the string structure. Don't forget, its a string of tokens, not characters.

Data and Storage	
"STRING"	Adds the string to program memory, but without the ending character.
[HEX]	Adds the hex to the program memory.
[OSVAR]	Absorbs the picture, appvar, program, or string from the calculator into the program. Only the source needs the var, not the executable.
[PICVAR"]	Absorbs the tile map picture from the calculator into the program. 12 tiles across, moving down. Only the source needs the pic, not the executable.
Data(CONST,...) *▲List()*	Adds the bytes to program memory. Numbers ending with " are added as 2 byte numbers.
Buff(SIZE) **Buff(SIZE,CONST)** *det()*	Creates a buffer that is size bytes large, filled with the byte constant, or with zero if unspecified.
DATA→NAME	Saves the data's pointer to the static variable. Also terminates current string if applicable.
CONST→NAME	Saves the constant's value to the static variable.
CONST→→NAME	Saves the constant's value to the static variable only if it hasn't already been defined. Ignored otherwise.
NAME	Returns a pointer to the start of the data.
See below	The list tokens point to "safe" areas of memory you are free to use.
L_1	768 bytes (saveSScreen) Volatility: **LOW**
L_2	531 bytes (statVars) Volatility: **LOW** (Some shells, including MirageOS, use this for custom interrupt and other data; it is advised to use LnReg or set up a custom Axe interrupt with fnInt() before using this area)
L_3	768 bytes (appBackUpScreen) Volatility: **MED** (Saving to back-buffer will corrupt)
L_4	256 bytes (tempSwapArea) Volatility: **MED** (Corrupt after archiving/unarchiving anything)
L_5	128 bytes (textShadow) Volatility: **MED** ("Disp","Output", and "ClrHome" will corrupt)
L_6	768 bytes (plotSScreen) Volatility: **HIGH** (Any buffer drawing will corrupt)
{EXP}	Returns the single byte the expression points to. It will be in the range 0 to 255.
{EXP}"	Returns the two-byte value the expression points to.
{EXP}""	Returns the big-endian two-byte value the expression points to but in reverse order.
EXP1→{EXP2}	The single byte of EXP1 is stored to where EXP2 points.
EXP1→{EXP2}"	The two-byte value of EXP1 is stored to where EXP2 points.

EXP1→{EXP2}ʳʳ	The big-endian two-byte value of EXP1 is stored to where EXP2 points.
signed{EXP} *int()*	Returns the single byte the expression points to. It will be in the range -128 to 127.
nib{EXP} *iPart()*	Returns the EXPth nibble in RAM. Use this to access external data in RAM. Since there are twice as many nibbles as bytes, make sure pointers are multiplied by 2.
nib{EXP}ʳ *iPart()*	Returns the EXPth nibble in RAM, or ROM if compiled as an application. Use this to access internal data. Since there are twice as many nibbles as bytes, make sure pointers are multiplied by 2.
EXP1→nib{EXP2} *iPart()*	Writes the nibble EXP1 to the EXP2th nibble in RAM. Since there are twice as many nibbles as bytes, make sure pointers are multiplied by 2.

Data Processing	
Fill(PTR,SIZE)	The byte already at PTR is copied to all the bytes after it until SIZE bytes have been filled with that value. 0 is not a valid SIZE.
Fill(PTR,SIZE,BYTE)	SIZE bytes of data starting at PTR are filled with a value. SIZE must be greater than 1.
Copy(BUF) *conj()*	Copies the 768 byte buffer to the main buffer. Same as Copy(BUF,L_6,768)
Copy(BUF1,BUF2) *conj()*	Copies the 768 byte buffer BUF1 to BUF2. Same as Copy(BUF1,BUF2,768)
Copy(PTR1,PTR2,SIZE) *conj()*	SIZE bytes starting from PTR1 are copied to PTR2 onwards. 0 is not a valid SIZE.
Copy(PTR1,PTR2,SIZE)ʳ *conj()*	SIZE bytes ending at PTR1 are copied to PTR2 moving backwards. 0 is not a valid SIZE.
Exch(PTR1,PTR2,SIZE) *expr()*	SIZE bytes starting from PTR1 are exchanged with SIZE bytes starting at PTR2. 0 is not a valid SIZE.
length(PTR)	Returns the number of bytes from the pointer to the next zero data element.
inData(BYTE,PTR) *inString()*	Searches for BYTE in the zero-terminated data. If found, it returns the position it was found in (starting at 1). If not found, 0 is returned. The byte 0 will never be found.
inData(BYTE,PTR,SIZE) *inString()*	Searches for BYTE in the data with the given size. If found, it returns the position it was found in (starting at 1). If not found, 0 is returned.
strGet(PTR,N) *stdDev()*	Returns the pointer to the Nth string (starting from zero) from a list of zero-terminated strings.
Equ▶String(STR1,STR2)	Checks if 2 null-terminated strings are equal. Returns 0 if equal and non-zero otherwise.
SortD(PTR,SIZE)	Sorts up to 256 bytes of data from largest to smallest starting at the pointed address.
cumSum(PTR,SIZE)	Calculates the 2-byte checksum of some data starting at the pointer for SIZE bytes.

External Variables	
Ans	Returns the OS's Ans variable as an integer. This is unrelated to any Axe value. Throws an error if out of range.
EXP→Ans	Stores the expression into the OS's Ans variable as an integer. This is unrelated to any Axe value.
GetCalc(PTR)	Finds the object who's name is pointed to and returns a pointer to the start of its data, or zero if it was archived

	or not found.
GetCalc(PTR,FILE)	Attempts to create a file of the OS variable who's name is pointed to so it can be read from archive. Returns 0 if the variable was not found, and non-zero otherwise.
GetCalc(PTR,SIZE)	Creates an OS variable who's name is pointed to in RAM and makes it SIZE bytes large. Returns a pointer to the start of data, or zero if there was not enough RAM. Overwrites existing variable, even if it was in archive.
UnArchive PTR	Tries to unarchive the object who's name is pointed to. Returns 1 if it could unarchive and 0 otherwise. Gives a memory error if not enough RAM.
Archive PTR	Tries to archive the object who's name is pointed to. Returns 1 if it could archive and 0 otherwise. Gives a memory error if not enough Flash Memory.
DelVar PTR	Deletes the OS variable who's name is pointed to even if in archive. Nothing happens if the variable does not exist.
float{PTR} *fPart()*	Converts the float at the pointed address to an integer. Floats are 9 bytes large.
EXP→float{PTR} *fPart()*	Converts the expression into a float and then stores it at the pointed address. Floats are 9 bytes large.

Interrupts	
fnInt(LBL,FREQ)	Turns the subroutine into an interrupt and then turns interrupts on. The frequency can be (fastest) 0, 2, 4, or 6 (slowest). L_2 is used for interrupt data so do not use L_2 for storage when using interrupts.
FnOn	Turns on interrupts.
FnOff	Turns off interrupts.
Stop	Stops execution until the next interrupt occurs. Interrupts must be on or else the calculator will freeze.
LnReg	Returns the calculator to regular interrupt mode.
LnReg'	Returns the calculator to regular interrupt mode and repairs the damage done by setting up a custom interrupt with fnInt(). It is highly suggested to use this before exiting the program if using a custom interrupt.

Link Port	
port *ClrTable*	Returns the status of the link port as a number 0-3.
EXP→port *ClrTable*	Sets the link port to a given status with a number 0-3. Must exit program with status 0 if changed!
Freq(WAVE,TIME) *SinReg*	Sound is played out of the link port. Wave is inversely proportional to frequency and Time must be much greater than Wave to hear anything.
Send(BYTE,TIME)	Tries to send the byte across the linkport. It will keep trying until the other calculator receives the byte or time runs out. Returns 1 if the byte was sent successfully or 0 if it timed-out. Time is in the order of microseconds.
Get *Get(*	Checks if the sender is trying to send anything. Returns the byte if it was received or -1 if nothing was sent. No waiting is done.