

Note that this is not complete or well organised. There are still over 50 calls to attend to, but most of the calls are finished and all but a few of the unfinished calls have an input/output mentioned, even if there is no mention of what the call does. If you have questions, feel free to email me or PM me at Omnimaga or UnitedTI (I am Thunderbolt there).

The header to Grammer ASM programs is:

first two bytes are the normal BB6D

the next two bytes are 55C9

also, .org should be 9D91h instead of 9D93h

so here is a sample to copy and paste (I use Brass):

```
.nolist
#define      TASM
#define      bcall(xxxx)      rst 28h \ .dw xxxx
#define      equ .equ
#include     "Grammer.inc"
.org        9D91h
.list
        .db $BB,6Dh,55h,$C9
Start:
```

To run the program on calc, do not use the Asm( token. If Grammer is installed, its parser hook will detect that it is a Grammer program and will load it properly. The program data (everything after the header) is copied to 9D95h even if it is archived.

If Grammer is installed, regular assembly programs can be run from the homescreen from RAM or archive that don't have the Grammer header and still use Grammer calls. If the program is an ION, MirageOS, or DoorsCS program, a custom error is thrown, instead.

I do not suggest releasing any no-stub programs that use Grammer calls.

```
ld hl,GramName
rst 10h
in a,(6)
push af
bcall(4C4Eh)
pop hl
ret c
push hl
out (6),a
call _CompatCall      ;to load the Grammer RAM
;=====
;Insert your program code here and use ExitProg to restore the flashpage
;=====
ExitProg:
    pop af
    out (6),a
    ret
```

Project.....Grammer Documentation  
Program.....Grammer (APP)  
Author.....Zeda Elnara (ThunderBolt)  
E-mail.....xedaelnara@gmail.com  
Size.....1-Page App  
Language.....English  
Programming.....Assembly  
Version.....2.24.12.11  
Last Update.....24 December 2011

## Call Info

### Introduction

In an attempt to better document Grammer and its calls, this document will address at least the basic information of the calls available in the jump table. This includes the address of the jump, the inputs and outputs, notes, comments and any other information that may be useful. My hope and intent is to open the floor to assembly programmers for creating new games and ideas by allowing the use of premade routines from Grammer.

### Terminology

#### **\_ProgramStart**

equ 4083h

Inputs:

Outputs:

Destroys: All

Notes: A program calls this to allow the Grammer parser to step in

#### **\_CompatCall**

equ 4086h

Inputs: The name of the program is in OP1

Outputs: z flag if the program has the proper header

A is the flashpage the data is on

BC is the size of the program

HL points to the data

(TempWord3) is A

(TempWord4) is BC

(TempWord5) is HL

Loads Grammer RAM

Destroys:

Notes: Use this to verify that a program is a Grammer program (or just to get the info of the var)

#### **\_SelectedProg**

equ 4089h

Inputs: Outputs from \_CompatCall

Outputs: Executes the program with the input data

Destroys: All

Notes: Call this if `_CompatCall` returns `z`

### **`_ExecOP1`**

equ 408Ch

Inputs: OP1 contains the name of the var (in RAM) to execute

Outputs:

Destroys: all

Notes: Executes the program named in OP1 as a Grammer program

### **`_ParserNext`**

equ 408Fh

Inputs: (progPtr) contains the start address to parse at  
BC is Ans

Outputs: BC contains the result of the parsing  
HL, (progPtr) contains the end address of the parsing

Destroys: A, DE

Notes: Use this to parse Grammer code that ends with Stop or End

### **`_ParseArg`**

equ 4092h

Inputs: (progPtr) points to the code to execute  
BC is Ans

Outputs: BC contains the result of the parsing  
HL, (progPtr) contains the end address of the parsing

Destroys: A, DE

Notes: Use this to parse an argument

### **`_ReadByte`**

equ 4095h

Inputs: HL points to a pointer

Outputs: BC contains the byte value that the pointer points to  
The 2-byte word at (hl) is incremented  
DE points to the byte read  
HL is not changed

Destroys:

Notes: HL points to the word that points to the byte to read. The word is then incremented. This lets you store the location of a variable to RAM and then read each byte incrementally.

### **`_Input`**

equ 4098h

Inputs: BC is the buffer size minus 1  
HL is the location of the buffer

Outputs: BC points to the string

Destroys: A, DE, HL

Notes: This is the Input routine used by Grammer

## **\_PCycle**

equ 409Bh

Inputs: BC points to the particle buffer

Outputs: BC is the number of particles

Destroys: A, DE, HL

Notes: This will recalculate the position and redraw the particles in the particle buffer.

## **\_PCycleDefault**

equ 409Eh

Inputs: (PBufPtr) points to the particle buffer

Outputs: BC is the number of particles in the buffer

Destroys: A, DE, HL

Notes: See \_PCycle

## **\_PBufInfoDef**

equ 40A1h

Inputs: (PBufPtr) points to the particle buffer

Outputs: BC is the size of the particle buffer

DE is the number of particles

HL points to the particle data

Destroys:

Notes: This is used to obtain information about a particle buffer

## **\_PBufInfo**

equ 40A4h

Inputs: HL points to the particle buffer

Outputs: BC is the size of the particle buffer

DE is the number of particles

HL points to the particle data

Destroys:

Notes: This is used to obtain information about a particle buffer

## **\_NextParticle**

equ 40A7h

Inputs: Outputs from PBufInfo

Outputs: HL points to where the new particle will go  
nz if there is enough room for ther particle

Destroys: Assume everything else for now

Notes:

## **\_AddParticle**

equ 40AAh

Inputs: BC=ParticlePlot

HL=ParticleLoc

Outputs: The particle is added and drawn.

Destroys:

Notes: To just add it, use "ld (hl),c \ inc hl \ ld (hl),b"

### **\_ShiftGraphBuf**

equ 40ADh

Inputs: A is the direction:

Down : bit 0 set

Left : bit 1 set

Right : bit 2 set

Up : bit 3 set

C is the number of shifts

Outputs:

Destroys: All registers

Notes: If you scan the arrow keys and invert the bits (using cpl),  
it will be the proper input for this routine.

### **\_ShiftGraphUpA**

equ 40B0h

Inputs: A is the number of shifts

Outputs:

Destroys: A,BC,DE,HL

Notes:

### **\_ShiftGraphRightA**

equ 40B3h

Inputs: A is the number of shifts

Outputs:

Destroys: A,BC,DE,HL

Notes:

### **\_ShiftGraphLeftA**

equ 40B6h

Inputs: A is the number of shifts

Outputs:

Destroys: A,BC,DE,HL

Notes:

### **\_ShiftGraphDownA**

equ 40B9h

Inputs: A is the number of shifts

Outputs:

Destroys: A,BC,DE,HL

Notes:

### **\_ZeroMemF**

equ 40BCh

Inputs: BC is the number of bytes minus 1 to zero  
HL points to the data to fill

Outputs: A=BC=0

DE is the original HL, plus BC+1

HL is the original HL, plus BC

Notes: This will fill the bytes with 0

### **\_SetMemF**

equ 40BFh

Inputs: A is the byte to fill with  
BC is the number of bytes minus 1 to fill  
HL points to the data to fill

Outputs: BC is 0  
DE is the original HL, plus BC+1  
HL is the original HL, plus BC

Destroys:

Notes: This will fill the bytes with the value in A

### **\_ZeroMemE**

equ 40C2h

Inputs: BC is the number of bytes minus 1 to fill  
HL points to the end of the data to fill

Outputs: A is 0  
BC is 0  
DE points to the byte before the start of the data  
HL points to the start of the data

Destroys:

Notes: This fills the bytes with 0

### **\_SetMemE**

equ 40C5h

Inputs: A is the byte to fill with  
BC is the number of bytes minus 1 to fill  
HL points to the end of the data to fill

Outputs: BC is 0  
DE points to the byte before the start of the data  
HL points to the start of the data

Destroys:

Notes: This fills the bytes with the value in A

### **\_nCr**

equ 40C8h

Inputs: HL is n  
DE is r

Outputs: BC is the result  
interrupts off  
a is 0  
de is "n"  
hl is the result  
a' is not changed  
bc' is "r"+1  
de' is an intermediate calculation  
hl' is "r" or the compliment, whichever is smaller

Destroys:

Notes: This performs "n choose r"

**\_SetSpeed** equ 40CBh

Inputs: C is 0=6MHz, 1=15MHz, 2=Toggle speed

Outputs: C is the previous speed setting

Destroys:

Notes:

**\_ZeroMem** equ 40CEh

Inputs: HL=Location  
BC=size

Outputs: A is 0

Destroys:

Notes: Similar to \_ZeroMems. This also detects if BC is 0.

**\_ClrDraw** equ 40D1h

Inputs:

Outputs: A is 0  
Clears the graph buffer  
Sets the text coordinates to 0

Destroys: BC,DE,HL

Notes:

**\_ClrHome** equ 40D4h

Inputs:

Outputs: A is 80h  
Clears the homescreen buffer  
Sets the (CurRow) and (CurCol) to 0

Destroys: BC,DE,HL

Notes:

**\_MaxMin** equ 40D7h

Inputs: Bit 0 of A is 1 if you want the max of BC and HL  
Bit 0 of A is 0 if you want the min of BC and HL  
HL,BC

Outputs: BC contains the max or min of the two values

Destroys:

Notes:

## **\_SetSmallMem**

equ 40DAh

Inputs: A is the value  
B is the number of bytes (0 is 256)  
HL is where to start

Outputs: B is 0  
HL is incremented B times

Destroys:

Notes: This is used to set a section of memory to the same value.

## **\_InvSmallMem**

equ 40DDh

Inputs: B is the number of bytes  
HL is where to start

Outputs: B is 0  
HL is incremented B times

Destroys:

Notes: This is used to set a section of memory to the same value.

## **\_Rand**

equ 40E0h

Inputs:

Outputs: Loads random values to the registers A,B,C,D,E,H,L  
(randSeed)

Destroys:

Notes:

## **\_expr**

equ 40E3h

Inputs: BC is the location of the code to interpret

Outputs: BC is result

Destroys:

Notes: This will execute Grammer code to an EOL

## **\_lcmBC\_DE**

equ 40E6h

Inputs: BC,DE

Outputs: HL and BC are the least commom multiple of the inputs

Destroys: Make no assumptions.

Notes: This is used to compute the least common multiple

## **\_gcdHL\_BC**

equ 40E9h

Inputs: HL,BC

Outputs: BC is the greatest common divisor of HL and BC

Destroys: Probably the rest of the registers

Notes: This computes the GCD



## **\_Pause**

equ 40ECh

Inputs: BC is the hundredths of seconds to pause

Outputs: BC is whatever DE was at input.

Destroys: DE,HL,A

Notes: If BC is 174, this will pause for 1.74 seconds.

## **\_PlotPixel**

equ 40EFh

Inputs: BC is (x,y)

D is the method:

0=PixelTest

1=PixelOn

2=PixelOff

3=PixelInvert

bit 4 at iy+34 is the result of an out of bounds pixel test

Outputs: A is the value of the byte

BC is the pixel test value before drawing the pixel

D is 0

E is the mask used

HL points to the byte written to in the buffer

Destroys:

Notes:

## **\_Call**

equ 40F2h

Inputs: BC points to the code to execute

Outputs: BC is the result of the code

HL points to the next byte

Destroys: A,DE

Notes: This will call a routine that ends with End.

## **\_PutSM**

equ 40F5h

Inputs: HL points to the string to display

BC is the size

Outputs: BC is 0

HL points to the byte after the string

Destroys: A,DE

Notes: This will display a string on the current buffer using the given fontset and style.

## **\_PutSprite**

equ 40F8h

Inputs: A is the Method

0=Overwrite

1=AND

2=XOR

3=OR

4=DataSwap

5=Erase

BC is (x,y)

DE points to the sprite data

H is the height

L is the width (not used yet)

Outputs:

Destroys: All

Notes: This draws a sprite to pixel coordinates

## **\_BreakProgram**

equ 40FBh

Inputs: (progPtr) is the location of the byte about to be parsed

(SPSave) is the address to use to reset the stack

Outputs:

Destroys:

Notes: This is supposed to exit the program. You should not need to change (SPSave)

## **\_ErrorJump**

equ 40FEh

Inputs: (parseError) points to Grammer code as an error handler,  
otherwise this is 0.

(progPtr) points to the error

(8595h) contains the Grammer error value

Outputs:

Destroys:

Notes: This will throw an error.

## **\_PutTile**

equ 4101h

Inputs: A is the Method  
0=Overwrite  
1=AND  
2=XOR  
3=OR  
4=Swap  
5=Erase  
B is the width in bytes  
C is the height in pixels  
DE points to the sprite data  
HL points to the buffer location to draw to

Outputs: A is 0  
B is not changed  
C is 12-B  
HL is A\*12 larger (next sprite down?)  
DE points to the next byte after the sprite data

Destroys: a'

Notes: Sprite data is set up in rows, then columns. The input HL is the offset into the draw buffer to start drawing at.

## **\_CheckKey**

equ 4104h

Inputs: A is the key to check

Outputs: BC is 1 if the key is being pressed, else it is 0  
z if the key is pressed, nz if the key is not pressed

Destroys:

Notes:

## **\_GetKey**

equ 4107h

Inputs:

Outputs: A is the key press (0 to 56)  
BC is the key press, also  
D is the last key group tested  
E is the same as A with a mask of %11111000  
HL is not changed

Destroys:

Notes:

### **\_ParseFullArg**

equ 410Ah

Inputs: (progPtr) points to the next byte to parse. The parsing stops at a colon, comma, newline, or sto token.  
BC is the input Ans

Outputs: A is the byte that ended the argument  
BC is the result  
(progPtr) points to the byte before the next argument  
HL is the same as (prgPtr)

Destroys: DE

Notes: Use this to compute an argument. If there is an argument immediately following, use `_ParseNextFullArg` after this.

### **\_ParseNextFullArg**

equ 410Dh

Inputs: (progPtr) points to the byte before the argument to parse. The parsing stops at a colon, comma, newline, or sto token.  
BC is the input Ans

Outputs: A is the byte that ended the argument  
BC is the result  
(progPtr) points to the byte before the next argument  
HL is the same as (prgPtr)

Destroys: DE

Notes: Use this to compute an argument.

### **\_ParseCondition**

equ 4110h

Inputs: (progPtr) points to the next byte to parse. The parsing stops at a colon, comma, newline, or sto token.  
BC is the input Ans

Outputs: A is 3Fh  
BC is the result  
(progPtr) points to the byte before the next argument  
HL is the same as (prgPtr)

Destroys: DE

Notes: This will parse a line of code (it parses until a newline).

### **\_Sine**

equ 4113h

Inputs: A is the value to find the sine of (-127 to 127)

Outputs: A' is twice the input  
BC is the result (-127 to 127)

Destroys: DE

Notes: add 40h to the input to compute the cosine.

## **\_EndOfCommand**

equ 4116h

Inputs: (progPtr) points to a location

Outputs: (progPtr) and hl point to the byte ending the argument  
HL is the same  
A is the ending byte

Destroys:

Notes:

## **\_EndOfArg**

equ 4119h

Inputs: A is the byte to test

Outputs: z if A is a byte that will end an argument

Destroys:

Notes:

## **\_EndOfArgNotSto**

equ 411Ch

Inputs: A is the byte to test

Outputs: z if A is a byte that will end an argument

Destroys:

Notes: This will not test if A is a Sto token (04h)

## **\_FindEndToken**

equ 411Fh Input: HL is the address, Output: HL points to the  
byte after the proper End

## **\_TokensToASCII**

equ 4122h Input: HL=StringLoc, Output: BC=Size, DE points to  
string

## **\_GetGrammerStr**

equ 4125h Input: HL=StartLoc, Output: BC=Size

## **\_pVarPointer**

equ 4128h Input: A=pVar, HL=NextByte, Output: HL points to  
var

## **\_EndOfLine**

equ 412Bh Input: HL=Start, Output: HL is tyhe next line

## **\_EndOfNumber**

equ 412Eh Input: HL points to the number, Output: HL points  
to the end

## **\_EndOfHexNum**

equ 4131h Input: HL points to start, Output: HL points to  
the end

## **\_IsHexTok**

equ 4134h Input: DE points to byte, Output: DE is  
incremented, nc if hex token

## **\_ConvRStr**

equ 4137h Input: HL points to the number (or byte before),  
Output: HL points to the byte after the number, BC  
is the result

## **\_HL\_Times\_BC**

equ 413Ah Input: HL,BC, Output: DEHL is result, A is 0

## **\_DE\_Times\_BC**

equ 413Dh Input: DE,BC, Output: DEHL is result, A is 0

## **\_DE\_Div\_BC**

equ 4140h Input: DE,BC, Output: HL=quotient, DE is remainder

## **\_HL\_Div\_BC**

equ 4143h Input: HL,BC, Output: HL=quotient, DE is remainder

<b><u>_HL_Times_A</u></b>	equ 4146h Input: HL,A, Output: DE=original HL, HL=Product, B is 0
<b><u>_DE_Times_A</u></b>	equ 4149h Input: DE,A, Output: HL=Product, B is 0
<b><u>_IsHLAtEOF</u></b>	equ 414Ch Input: HL, Output: DE is (progEnd), c if HL is not at or past the EOF
<b><u>_SearchString</u></b>	equ 414Fh Input: BC=Size, DE=SearchStr, HL=Start, Output: cf if there was a match, HL points to match
<b><u>_CheckStatus</u></b>	equ 4152h Output: z if ON is pressed, c if 15MHz mode set
<b><u>_GraphToLCD</u></b>	equ 4155h
<b><u>_BufferToLCD</u></b>	equ 4158h Input: HL points to the buffer
<b><u>_DrawRectToGraph</u></b>	equ 415Bh Input: A=Type, B=Height, C=Y-Coord, D=Width, E=X-Coord
<b><u>_PutSS</u></b>	equ 415Eh Input: HL points to sting, D is the byte to end at, (textRow), (textCol)
<b><u>_GPutSS</u></b>	equ 4161h Input: HL points to the zero terminated string, b=Col, c=Row
<b><u>_GPutS</u></b>	equ 4164h Input: HL points to the zero terminated string, (textRow), (textCol)
<b><u>_PutSC</u></b>	equ 4167h Input: A=char, (textRow), (textCol), Output: Updated coords
<b><u>_PutFS</u></b>	equ 416Ah Input: A=char, (textRow), (textCol)
<b><u>_SqrHL</u></b>	equ 416Dh Input: HL, Output: E
<b><u>_Circle</u></b>	equ 4170h Input: DE=center(x,y), c=radius, a=method
<b><u>_SetMem</u></b>	equ 4173h Input: A is the fill byte, HL is the location, BC is the size
<b><u>_ConvNumBase</u></b>	equ 4176h Input: C is the base, HL is the number, Output: A=BC=#ofDigits, DE=0, HL=pointer to number string
<b><u>_PrimeTest</u></b>	equ 4179h Input: HL, Output: BC=LowestFactor, HL is the result after dividing by BC, cf if prime
<b><u>_HL_Div_C</u></b>	equ 417Ch Input: HL,C Output: A is the remainder, B is 0, HL is the quotient
<b><u>_Is_2_Byte</u></b>	equ 417Fh Input: A, Output: z if 2-byte
<b><u>_Is_Var_Name</u></b>	equ 4182h Input: A, Output: z if start of a var name
<b><u>_DrawLine</u></b>	equ 4185h Input: A=Method, BC=(x1,y1), DE=(y2,x2)
<b><u>_InCHLMem1</u></b>	equ 4188h Output: HL is incremented, if it
<b><u>_Conv_OP1</u></b>	equ 418Bh Output: HL is incremented by 9, DE is the value, A is the 8-bit value
<b><u>_ConvDecAtHL</u></b>	equ 418Eh Input: Hl points to the FP number to convert, Output: Same as above

<b><u>_GetPixelLoc</u></b>	equ 4191h	Input: bc=(x,y), Output: HL points to the byte, A is the mask, c if on screen	
<b><u>_HexTok</u></b>	equ 4194h	Input: HL=Output, DE=Input (hex), BC=Output size	
<b><u>_OP1NameLength</u></b>	equ 4197h	Output: BC is the size of the program name in OP1	
<b><u>_NameLength</u></b>	equ 419Ah	Input: HL points to the zero terminated string, Output: same as above	
<b><u>_IsOP1GrammerProg</u></b>	equ 419Dh	Output: z if Grammer prog, A=flashpage of data, BC is the size, HL points to the data	
<b><u>_ReadArc</u></b>	equ 41A0h	Input: A=page, HL=Input, DE=output, BC=size	
<b><u>_CallError</u></b>	equ 41A3h	Input: A is the error code	
<b><u>_ONErr</u></b>	equ 41A6h		
<b><u>_MemErr</u></b>	equ 41A9h		
<b><u>_C_Div_L</u></b>	equ 41ACH	Output: A is the remainder, BC is the quotient	
<b><u>_TileMap1</u></b>	equ 41AFh	Input: DE=MapLoc, BC=TileLoc, HL=MapWidth, A=TileLogic, (TempWord2)=MapX, (TempWord3)=MapY	
<b><u>_HL_SDiv_BC</u></b>	equ 41B2h	Output: HL=Quotient, DE=remainder	
<b><u>_VPutC</u></b>	equ 41B5h		
<b><u>_RoundHL_Div_C</u></b>	equ 41B8h	Output: HL is the quotient, nc means it rounded up	
<b><u>_SearchLine</u></b>	equ 41BBh	Input: HL=SearchStart, BC=number of bytes to search, DE=LineNumber, A is the linebyte, Output: BC is the number of bytes left, DE points to the line, HL is the length of the line	
<b><u>_SendByte</u></b>			equ 41BEh
	Inputs: HL is the Time E is the byte to send		
	Outputs:		
	Destroys:		
	Notes:		
<b><u>_GetByte</u></b>			equ 41C1h
	Inputs:		
	Outputs: L is the byte recieved H is 0		
	Destroys:		
	Notes:		
<b><u>_PutIM</u></b>			equ 41C4h
	Inputs: The string to display follows the call		
	Outputs: All registers are preserved		
	Destroys:		

Notes: The first two bytes after the call is the string size (little endian), then the string data. For example:

```
call _ClrDraw
call _PutIM
.dw 5
.db "Hello"
jp GraphToLCD
```

## **\_GPutSI**

equ 41C7h

Inputs: The zero-terminated string to display follows the call

Outputs: All registers are preserved

Destroys:

Notes: An example of using this:

```
call _ClrDraw
call _GPutSI
.db "Hello World!",0
jp GraphToLCD
```

## **\_DrawRectToGraphI**

equ 41CAh

Inputs: The 5 bytes following the call are the inputs in this order:

```
X coordinate
Y coordinate
Height
Width
Method
```

This uses the same drawing methods as [\\_DrawRectToGraph](#)

Outputs: All registers are preserved

Destroys:

Notes: This is useful if you need to draw a menu or something where the rectangles have fixed sizes and position. It saves at least 3 bytes per rectangle to use this.

## **\_ParseFullArgI**

equ 41CDh

Inputs: BC is Ans

The code to execute directly follows the call.

Outputs: BC is the result

All other registers are preserved

Destroys:

Notes: This will parse some Grammer code following the call.

## **\_CallI**

equ 41D0h



Inputs: BC is the input Ans  
The code following (up until an End token)

Outputs: BC is the result  
All other registers are preserved

Destroys:

Notes: This will parse the data following the call as Grammer code until an End token is reached.

### **\_DEHL\_Mul\_32Stack**

equ 41D3h

Inputs: DEHL  
Two pushes to the stack. The last push is the lower 16-bits.

Outputs: AF is the return address  
HLDEBC is the lower 48-bit result  
4 bytes at TempWord1 contain the upper 32-bits of the result  
4 bytes at TempWord3 contain the input stack values  
The stack contains two pops to perform:  
First pop is the bits 33 to 48  
Second pop is the bits 49 to 64

Destroys:

Notes: This routine will multiply two 32-bit values and routine a 64-bit routine. To multiply 21030332h\*54010320h:

```
ld hl,2103h \ push hl
ld hl,0332h \ push hl
call _DEHL_Mul_32Stack
pop af \ pop af
```

Now, technically, AFHLDEBC contains the 64-bit result where C is the lowest 8-bits.

## **\_CopyZStr**

equ 41D6h

Inputs: The string to copy is zero-terminated and follows the call  
DE is the location to copy to

Outputs: All registers preserved, the string is copied.

Destroys:

Notes:

## **\_CreateZVar**

equ 41D9h

Inputs: The name of the var directly follows the call and is zero-terminated  
BC is the size of the var to create

Outputs: AF is preserved  
BC is the size of the var  
DE points to the var data  
HL points to the SymEntry

Destroys:

Notes: This will delete a preexisting var

## **\_ChkFindVar**

equ 41DCh

Inputs: OP1 contains the name of the program, protected program,  
temporary program, appvar, or group to search for.

Outputs: A is the type  
B is the flashpage  
C is the name length  
DE points to the size bytes  
HL points to the symentry

Destroys:

Notes:

## **\_ChkFindVarAtDE**

equ 41DFh

Inputs: DE points to the name of the program, protected program,  
temporary program, appvar, or group to search for.

Outputs: A is the type  
B is the flashpage  
C is the name length  
DE points to the size bytes  
HL points to the symentry

Destroys:

Notes: The name must include the type byte followed by the a string  
that is either zero-terminated or 8 bytes long

## **\_SearchVarBC**

equ 41E2h

Inputs: BC contains the name of the var to search for, excluding programs, appvars, or any other user named variable. C is the type and B is the number. For example, 01AAh searches for Str2.

Outputs: A is the type  
B is the flashpage  
C is 2 (the name length)  
DE points to the size bytes  
HL points to the symentry

Destroys:

Notes:

## **\_SearchVarAtHL**

equ 41E5h

## **\_SearchVarAtDE**

equ 41E8h

## **\_FindSym**

equ 41EBh

## **\_FindVar**

equ 41EEh

Inputs: OP1 contains the name of the var to search for

Outputs: A is the type  
B is the flashpage  
C is the name length  
DE points to the size bytes  
HL points to the symentry

Destroys:

Notes: This combines \_FindSym and \_ChkFindVar to cover all vars