```
Project.........SpriteLib
Program.........βαtʟib
Author..........Zeda Elnara (ThunderBolt)
E-mail..........xedaelnara@gmail.com
Size............Big :P
Language........English
Programming.....Assembly
Version.........4.00.00 (beta)
Last Update.....01 January 2011
```

# βαtʟib Calls

This is a guide to a bunch of the calls in βαtʟib that can be used in opcodes. I have no clue how far βαtʟib will go, so these might come in handy down the road (in case βαtʟib functions as a shell, too). For now, if you use a regular program instead of the commands 22 or 23, you should start your program with this:

```
.nolist
#include "Zeda.inc"
#include "ti83plus.inc"
.list
.org    $9D93
    .dw $6DBB
SetUp:
    in a,(6)            ;DB06
    ld (SavePage),a     ;320080      ;This is an equate in zeda.inc
    ld hl,BatLibName    ;21****
    rst 20h             ;E7
    bcall(_FindApp)     ;EF4E4C
    ret c               ;D8
    out (6),a           ;D306
    call BatLibInstall  ;CDEC41     ;This loads all the RAM stuff
;
;Before you exit use this code (else it crashes):
    ld a,(SavePage)     ;3A0080
    out (6),a           ;D306
BatLibName:
    .db $14,$BC,$BB,$74,$DC,$D7,$62,$20,$20
```

## DE_Div_BC

4099h

**Description:** Divides DE by BC

**Inputs:**

**Registers:** DE, BC

**RAM:**

**Outputs:**

**Registers:** HL is the quotient
DE is remainder
BC is unchanged
A is 0

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is really just  ex de,hl before HL_Div_BC

## HL_Div_BC

409Ch

**Description:** Divides HL by BC

**Inputs:**

**Registers:** HL, BC

**RAM:**

**Outputs:**

**Registers:** HL is the quotient
DE is remainder
BC is unchanged
A is 0

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## DE_Times_BC

**Description:** Multiplies DE by BC

**Inputs:**

**Registers:** DE,BC

**RAM:**

**Outputs:**

**Registers:** A is 0
DE is 0
BC is unchanged
HL is the product

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## HL_Times_BC

**Description:** Multiplies HL by BC

**Inputs:**

**Registers:** HL,BC

**RAM:**

**Outputs:**

**Registers:** A is 0
DE is 0
BC is unchanged
HL is the product

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## HL_Div_10

**Description:** Divides HL by 10

**Inputs:**

**Registers:** HL

**RAM:**

**Outputs:**

**Registers:** A is the remainder
BC is 10
DE is not changed
HL is the quotient

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used in the SetXXXX routines

## HL_Div_C

**Description:** Divides HL by C

**Inputs:**

**Registers:** HL,C

**RAM:**

**Outputs:**

**Registers:** A is the remainder
B is 0
C is not changed
DE is not changed
HL is the quotient

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** HL_Div_10 uses this call.

## RoundHL_Div_10

410Bh

**Description:** This gives a rounded value for HL/10

**Inputs:**

**Registers:** HL

**RAM:**

**Outputs:**

**Registers:** A is the signed remainder
BC is 5
DE is unchanged
HL is the rounded value

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This isn't used in βαtʟ*i*b, yet, but I figured it might be useful to somebody.

## RoundHL_Div_C

**Description:** This gives a rounded value for HL/C

**Inputs:**

**Registers:** HL

**RAM:**

**Outputs:**

**Registers:** A is the signed remainder
BC is half the original C
DE is unchanged
HL is the rounded value

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## HL_Times_A

**Description:** Multiplies HL times A

**Inputs:**

**Registers:** HL,A

**RAM:**

**Outputs:**

**Registers:** HL is the product
DE is the old value of HL
B is 0
C is unchanged
A is unchanged

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is really just an EB before DE_Times_A

## DE_Times_A

**Description:** Multiplies DE times A

**Inputs:**

**Registers:** DE,A

**RAM:**

**Outputs:**

**Registers:** HL is the product
B is 0
A is unchanged
C is unchanged
DE is unchanged

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

# MP_Divide_E

**Description:** This is a multiprecision routine. It divides a number by E.

**Inputs:**

**Registers:** HL points to the number
C is the length in bytes of the number (up to 32)
E is the value to divide it by

**RAM:**

**Outputs:**

**Registers:** A is the remainder
D is the remainder
E is not changed
BC is the length of the quotient
HL points to the quotient
c flag is reset
z is set if the quotient is 0

**RAM:** The original number is replaced by the quotient

**Destroys:**

**Registers:**

**RAM:**

**Notes:** HL does not necessarily remain the same. If the quotient is smaller in bytes than the original number, HL points to the start of the number (as opposed to the leading zero's). BC is also adjusted to reflect the new length. For example, to find the base 10 value of a very large number:

```
 ld hl,Number    ;This points to the size byte of the number
 ld de,Output    ;This is the output location (last byte)
 ld c,(hl)       ;The number has a leading size byte
 inc hl          ;Now it points to the number
Convert:
  push de
  ld e,10
  Call MP_Div_E  ;Divides by 10...
  pop de
  ld (de),a
  dec de
  jr nz,-5       ;keeps dividing until size is 0
 ret
```

## TileMap

40B7h

**Description:** This makes a simple tile map with 8x8 tiles.

**Inputs:**

**Registers:** HL points to the start of sprite data
DE points to the tile map data

**RAM:**

**Outputs:**

**Registers:** A is 1
B is 0
C is unchanged
HL is plotSScreen+12 or 934Ch
DE points to the byte after the tilemap data

**RAM:**

**Destroys:**

**Registers:**

**RAM:** TempWord1
TempWord2

**Notes:** The sprte data is an array of 8x8 sprites. Tile map data is read in columns starting at the upper left.
This does not update the screen.

## PutSprite8x8

411Dh

**Description:** Draws an 8x8 sprite without updating the screen.

**Inputs:**

**Registers:** DE points to the sprite data
HL points to where the sprite is drawn (usually somewhere in plotSScreen)

**RAM:** PutSprite8x8+4 is the type of logic to use

**Outputs:**

**Registers:** A is 1
BC is 12
DE points to the byte after the sprite data (possibly the next sprite)
HL points to location below the sprite

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This used by the TileMap routine

## PutSpriteXxY

**Description:** Draws a variable size sprite without updating the screen.

**Inputs:**

**Registers:** A is the height
B is the width (in bytes)
DE points to the sprite data
HL points to the output location

**RAM:** PutSpriteXxY+9 is the logic to use

**Outputs:**

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

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## SetSpriteLogic

**Description:** Changes a byte in PutSprite8x8 and PutSpriteXxY to make them draw a sprite using logic.

**Inputs:**

**Registers:**

**RAM:** ArgLoc points to an FP number

**Outputs:**

**Registers:** HL is the updated ArgLoc value
DE is the converted 16-bit value of the FP number
A is the mod 4 value of the FP number

**RAM:** ArgLoc points to the next FP number if it is an array
ArgCount is decremented if it is not 0
If the FP number is:
- 0, the logic is changed to AND
- 1, the logic is changed to OR
- 2, the logic is changed to XOR
- 3, the logic is changed to Overwrite

**Destroys:**

**Registers:** BC

**RAM:**

**Notes:** If you already have an A value, call SetSpriteLogic+3 and it will not use LoadIncrement.

## Coordinates

40E4h

**Description:** This uses two inputs in an FP array (like a list) to calculate the offset for a sprite.

**Inputs:**

**Registers:**

**RAM:** ArgLoc points to two FP numbers where the first is the X offset (0~11) and the second is the Y offset (0~63)

**Outputs:**

**Registers:** HL is the offset to use.
DE is the value of the first FP number
A is the value of the second FP number

**RAM:** ArgLoc and ArgCount are updated

**Destroys:**

**Registers:** BC

**RAM:**

**Notes:** Add 9340h to HL to get the offset into plotSScreen or if you are using saveSScreen as a buffer, add 86ECh. Then you can use a PutSprite routine ^_^

## DPutSprite

40E7h

**Description:** This draws a sprite directly to the LCD instead of a buffer

**Inputs:**

**Registers:** B is the height of the sprite in pixels
C is the width of the sprite in bytes
DE points to the sprite data
H is the X-coordinate
L is the Y-Coordinate

**RAM:**

**Outputs:**

**Registers:** A is 0
B is unchanged
C is 0
DE points to the byte after the sprite data
H is the old value of H plus the old value of B
L is the old value of L plus C

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** The sprite data is set up in columns instead or rows, like the other commands, so if you use ConvDSprite, you can convert regular sprite data to this format.
Because this draws directly to the LCD, updating the graph buffer overwrites the sprite. This should be used for moving sprites or other temporary sprites.

## ConvDSprite

40EAh

**Description:** This converts regular Sprite data to the kind that is compatible with PutDSprite

**Inputs:**

**Registers:** B is the width of the sprite
C is the height of the sprite
DE points to  the sprite data

**RAM:**

**Outputs:**

**Registers:** DE points to the converted data
All other registers are left intact

**RAM:** The data is converted to a spot in saveSScreen

**Destroys:**

**Registers:**

**RAM:**

**Notes:** You can use this directly before using PutDSprite to convert the data since none of the registers are modified except DE (which points to the correct data).

## LCDToBuffer

**Description:** Copies the contents of the LCD to a 768 byte area of RAM

**Inputs:**

**Registers:** HL points to the buffer.

**RAM:**

**Outputs:**

**Registers:** A is 0
BC is 11h
DE is unaffected
HL points to the byte after the buffer
z flag is set
c flag is reset
p/v flag reset

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## BufferToLCD

**Description:** This copies 768 bytes of data to the LCD screen.

**Inputs:**

**Registers:** HL points to the buffer

**RAM:**

**Outputs:**

**Registers:** BC is 11h
DE is unaffected
HL points to the byte after the buffer
z flag is set
c flag is reset
p/v flag reset

**RAM:**

**Destroys:**

**Registers:** A-I think it is 63h, but I am not sure if it is always 63h

**RAM:**

**Notes:**

## DrawRectPattern

**Description:** This draws a rectangle of some format

**Inputs:**

**Registers:** A is the type of rectangle to draw

```
0 =White
1 =Black
2 =XOR
3 =Black border
4 =White border
5 =XOR border
6 =Black border, white inside
7 =Black border, XOR inside
8 =White border, black inside
9 =White border, XOR inside
10=Shift Up
11=Shift Down
```

B is the height
C is the Y pixel coordinate
D is the width in pixels
E is is the X pixel coordinate

**RAM:**

**Outputs:**

**Registers:** BC is 0

**RAM:** Draws the rectangle on plotSScreen.

**Destroys:**

**Registers:** A is the value of the last byte loaded to plotSScreen.
HL points to the last byte in the last row in plotSScreen where the rectangle was drawn.
DE is either saveSScreen+12 or saveSScreen+24, depending on the routine used

**RAM:** The first 24 bytes of saveSScreen is used for storing the pattern

**Notes:** Use these instead of the OS bcalls. The reasons are:
**Speed**-this is faster
**Accuracy**-These always work like they are supposed to, unlike the bcalls which can turn off pixels around the borders
**Safety**-This uses width and height and a coordinate. The bcalls use two coordinates where one must not be less than other (else a crash occurs). For this call, B+C cannot be greater than 64 and D+E cannot be greater than 96
**Options**- This provides more rectangle options than the bcalls :P

## PutSprite6x8

**Description:** Draws a sprite directly on the LCD, 6 pixels wide, 8 pixels tall

**Inputs:**

**Registers:** B is the column to start drawing at (0~23)
C is the row to start drawing at (0~63)
HL points to the sprite data

**RAM:**

**Outputs:**

**Registers:** A is 1
B is 0
C is not changed
DE is not changed
HL points to location after the sprite data

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This can be used in custom font routines... :D

## Data Info

## GetString

4120h

**Description:** This is used to determine the length of a string in RAM ending in 3Fh (or newline)

**Inputs:**

**Registers:** HL points to the byte before the string

**RAM:**

**Outputs:**

**Registers:**

**RAM:** DE points to the string
BC is the size of the string
HL points to the end of the string

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used in ReCode and is likely to change...

## GetStringArc

4123h

**Description:** This returns the stats of a string

**Inputs:**

**Registers:** A is the flash page the string starts on
E is the byte the string ends with
HL points to the start of the string

**RAM:**

**Outputs:**

**Registers:** HL points to the start of the next string
DE is the start of the string
BC is the length of the string
A is the ending page or 0 if in RAM

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** If E is 3Fh (newline) and HL points to the data start of a program, the result will be the stats of the first line and HL will point to the start of the second line.

## End_Of_Number

40EDh

**Description:** This is used to set up for ConvRStr

**Inputs:**

**Registers:** HL points to a string of base 10 digits

**RAM:**

**Outputs:**

**Registers:** HL points to the byte after the last base 10 number
A is the value of the byte at (HL)

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used by ReCode to point to the end of a number for ConvRStr
(It is used in the call ConvRStr)

## CorrectOffset

40F9h

**Description:** When you are about to read from archive and HL points to an unknown offset, you can use this to adjust the page and address to read from.

**Inputs:**

**Registers:** A is the start page
HL is the offset into that page

**RAM:**

**Outputs:**

**Registers:** HL and A are adjusted if HL does not point to an address from 4000h to 7FFFh and A is not 0 (pointing to RAM).
BC is not changed

DE is not changed

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used when the user defines an offset into data that may be in archive.

## LocateStr1

40FCh

**Description:** This finds Str1 and returns key data about it

**Inputs:**

**Registers:**

**RAM:**

**Outputs:**

**Registers:** A is 4
BC is the size of the string
DE points to the data (not the size bytes)
HL points to the SymEntry

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used often in the program version of βατιib. There really isn't a use for it here anymore... Oh well, it's only 16 bytes.

## LocateStr

40F3h

**Description:** This finds the next string to parse and returns info.

**Inputs:**

**Registers:**

**RAM:**

**Outputs:**

**Registers:** A is 4
BC is the size of the string
DE points to the data (not the size bytes)
HL points to the SymEntry

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used often in βατιib.

## FindVarNameArc

4129h

**Description:** This is used to get crucial data about a variable

**Inputs:**

**Registers:**

**RAM:** ArgLoc points to nine bytes containing the name of the string with the var name in it. Following this is an FP number that is the type of the var.

**Outputs:**

**Registers:** A is the flash page the data is on (or 0 if in RAM)
HL points to the variable data
DE points to the SymEntry
BC is the size of the variable

**RAM:** ArgLoc and ArgCount are adjusted

**Destroys:**

**Registers:**

**RAM:** OP1

**Notes:** βαtιib uses this very often. Whenever you see a syntax use sum(xx,“VarName”,Type), guess which call is used?

## GetStatsArc

412Ch

**Description:** This is used to get info about a variable that is in RAM or archive

**Inputs:**

**Registers:** A is the page the data is on or 0 if in RAM
DE points to the data
BC is the length of the name

**RAM:**

**Outputs:**

**Registers:** A is the flash page the data is on (or 0 if in RAM)
HL points to the variable data
DE points to whatever HL was before the call
BC is the size of the variable

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is actually an offset into FindVarNameArc. Here is an example of code to use before calling this:

```
push bc                ;C5       This is the length of the var name
bcall(_CheckFindSym)   ;EFF142   Gets general info about the var
ld a,b                 ;78       This is the flash page
pop bc                 ;C1       Erm... Length o' the string in BC
```

Some var types do not work or do not work properly (like matrices and lists). If the list has a token name, like L1, then add one to the length and you will be set :D

## FindVar

40FFh

**Description:** This works exactly like FindVarSym

**Inputs:**

**Registers:**

**RAM:** OP1 is the name of a variable that is **not** a named variable (like a program)

**Outputs:**

**Registers:** A is the variable type
B is the flash page
DE points to the size bytes
HL points to the SymEntry
*The c flag is set if the variable is not found, reset if it is

**RAM:**

**Destroys:**

**Registers:** C

**RAM:**

**Notes:** I actually wouldn't use this. I use D7 instead. I included it only because it used only three extra bytes

## FindVarSym(DE)

4102h

**Description:** This finds a variable who's name starts at (DE)

**Inputs:**

**Registers:** DE points to the name of a variable that is **not** a named variable.

**RAM:**

**Outputs:**

**Registers:** A is the variable type
B is the flash page
DE points to the size bytes
HL points to the SymEntry
*The c flag is set if the variable is not found, reset if it is

**RAM:**

**Destroys:**

**Registers:** C

**RAM:**

**Notes:** This can be pretty useful in case the name is stored in Str1 or something. So, if Str1 was "Pic1" and DE pointed to the data in Str1, it would search for Pic1.

## FindVarSym(HL)

4105h

**Description:** This finds a variable who's name starts at (HL)

**Inputs:**

**Registers:** HL points to the name of a variable that is **not** a named variable.

**RAM:**

**Outputs:**

**Registers:** A is the variable type
B is the flash page
DE points to the size bytes
HL points to the SymEntry
*The c flag is set if the variable is not found, reset if it is

**RAM:**

**Destroys:**

**Registers:** C

**RAM:**

**Notes:** See FindVarSym(DE)

## SearchVarBC

4108h

**Description:** Searches for a simple variable that is **not** a named var

**Inputs:**

**Registers:** BC is the name of the variable

**RAM:**

**Outputs:**

**Registers:** A is the variable type
B is the flash page
DE points to the size bytes
HL points to the SymEntry
*The c flag is set if the variable is not found, reset if it is

**RAM:**

**Destroys:**

**Registers:** C

**RAM:**

**Notes:** As an example, to find Pic1, BC would be 0060h

Convert

## ConvRStr

40F0h

**Description:** Converts a string of real numbers to hex

**Inputs:**

**Registers:** HL points to either the byte before the string or the byte starting the string.

**RAM:**

**Outputs:**

**Registers:** BC is the converted value
DE points to the start of the number
HL points to the byte after the real number
A is the byte at (HL)

**RAM:** AnsWord1 is the converted value

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is used in ReCode to convert numbers

## SetXXXXOP1

40A5h

**Description:** Converts HL to a floating point value and stores it to OP1

**Inputs:**

**Registers:** HL

**RAM:**

**Outputs:**

**Registers:** HL is 7
DE points to OP1+9
BC is 0
A is the number of digits in the floating point result plus 80h

**RAM:**

**Destroys:**

**Registers:**

**RAM:** TempWord2
TempWord3
TempWord4

**Notes:** The result is always an even number of digits, so there may be a leading 0. It doesn't harm anything.

## SetXXXXDE

40A8h

**Description:** Converts HL to a floating point value and stores it to RAM pointed to by DE

**Inputs:**

**Registers:** HL,DE

**RAM:**

**Outputs:**

**Registers:** HL is 7
DE incremented by 9
BC is 0
A is the number of digits in the floating point result plus 80h

**RAM:**

**Destroys:**

**Registers:**

**RAM:** TempWord2
TempWord3
TempWord4

**Notes:** If DE points to a list element, after using this call, it will point to the next list element.

## LoadIncrement

40D2h

**Description:** Using pointers, this converts the next floating point value and sets up pointers to the following FP number.

**Inputs:**

**Registers:**

**RAM:** ArgCount is the number of pieces of data. (not really needed)
ArgLoc points to a floating point number or array

**Outputs:**

**Registers:** DE is the converted value of the FP number
A is equal to E
HL points to the next FP number (if it is an array like a list)
TempWord2 contains the value of DE

**RAM:** ArgLoc is the same as HL
ArgCount is decremented unless it is 0

**Destroys:**

**Registers:** BC

**RAM:**

**Notes:** In βατʟ*i*b, this is used to obtain the converted value of the next element in the input list. It is used very often.

## ConvertSupplement

40D5h

**Description:** This is a call used by the ConvDecAtHL routine.

**Inputs:**

**Registers:** A is a value
HL is a value

**RAM:** TempWord4 is a value

**Outputs:**

**Registers:** A is 10
B is 0
C is unchanged
DE is unchanged
HL is multiplied by ten

**RAM:** The original A and HL are multiplied and added to TempWord4

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## Conv_OP1

40D8h

**Description:** This is meant to be a much need alternative to ConvOP1

**Inputs:**

**Registers:**

**RAM:** OP1 contains the FP number to convert

**Outputs:**

**Registers:** A is the 8-bit converted value
DE is the 16-bit converted value
HL is OP2+1, making it ready to convert OP2

**RAM:** TempWord2 is the 16-bit converted value

**Destroys:**

**Registers:** BC

**RAM:**

**Notes:** The advantage here is that it is not limited to 9999. The number can be just about anything and it will return the mod 65536 value.
See the next routine if this one makes you happy...

## ConvDecAtHL

40DBh

**Description:** This is used to convert a FP number in RAM

**Inputs:**

**Registers:** HL points to where the FP number starts, plus 1. So converting OP1 would need HL to be OP1+1

**RAM:**

**Outputs:**

**Registers:** A is the 8-bit converted value
DE is the 16-bit converted value
HL is incremented by 9, pointing to the next FP number if it is an array/list

**RAM:** TempWord2 is the 16-bit converted value

**Destroys:**

**Registers:** BC

**RAM:**

**Notes:** If the B_Call ConvOP1 could handle numbers larger than 9999, the equivalent would be:

```
B_Call Mov9ToOP1     ;EF7A41
push HL              ;E5
B_Call ConvOP1       ;EFEF4A
pop HL               ;E1
```

But of course, even that destroys OP1 if not anything else, so in other words, if you can use this call, use it!

## SwapBytes

40F6h

**Description:** Swaps the bytes from one location to another

**Inputs:**

**Registers:** HL points to one address
DE points to another address
BC is the number of bytes to swap

**RAM:**

**Outputs:**

**Registers:** A is the value at the last byte pointed to by (HL-1)
BC is 0
DE points to the byte after the swapped data
HL points to the byte after the swapped data

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## ReadArc

4126h

**Description:** This is like LDIR except A is used to better refine where to read from.

**Inputs:**

**Registers:** A is  the flash page to read from (or 0 if in RAM)
HL points to the data to read.
DE points to where the data is copied
BC is the number of bytes to read

**RAM:**

**Outputs:**

**Registers:**

**RAM:** A is the current loaded flash page
B is the last flash page read from
C is 0
DE points to the byte after where the data was copied
HL points to the byte after where the data was read

**Destroys:**

**Registers:**

**RAM:**

**Notes:**

## GetTokAtHL

40C0h

**Description:** Used to convert hex to tokens

**Inputs:**

**Registers:** HL points to the byte to convert
DE points to where the data gets converted to

**RAM:**

**Outputs:**

**Registers:** DE is decremented by 1
A is the token equivalent of the last nibble at (hl)

**RAM:** The byte at (DE) contains the converted value of the LSN at (HL)

**Destroys:**

**Registers:**

**RAM:** Practically destroys the byte at (HL)

**Notes:** Use this again to convert the MSN at (HL)

## GetHexAtDE

40C3h

**Description:** Used in HexTok

**Inputs:**

**Registers:** DE points to the data to be read
HL is where the nibbles are screwed with. Read the note below.

**RAM:**

**Outputs:**

**Registers:** DE is decremented by 1

**RAM:**

**Destroys:**

**Registers:** A

**RAM:**

**Notes:** I suggest not using it because it is honestly a little complicated. To give an idea, it makes the last nibble in A a value from 0~F, depending on the token character, and then uses RLD. RLD is just too weird to explain (It likes to mess with nibbles at (HL) and A). It took about an hour to come up with this idea and get it to work properly :D

## TokHex

40C6h

**Description:** Converts a string of tokens to hex

**Inputs:**

**Registers:** HL poins to the END of the data to be converted
DE points to the END of where to convert the data
BC is the size of the data to be converted

**RAM:**

**Outputs:**

**Registers:** A is 0
BC is 0
DE points to the start of where the data was read
HL points to the start of where the data was copied

**RAM:**

**Destroys:**

**Registers:**

**RAM:** The RAM were the string was

**Notes:**

## HexTok

40C9h

**Description:** This converts a string of hex to tokens

**Inputs:**

**Registers:** HL points to where the tokenized dat is put
DE points to where the data to tokenize is
BC is the size of the tokenized data

**RAM:**

**Outputs:**

**Registers:** BC is 0
DE points to the end of the data that was read
HL points to the end of where the data was copied

**RAM:**

**Destroys:**

**Registers:** A

**RAM:** Practiacally demolishes the original data

**Notes:** This is helpful for making data.

## ASCIIToken

40DEh

**Description:** Converts a string of lowercase letters, uppercase letters, and numbers to tokens.

**Inputs:**

**Registers:** HL points to the data to convert
BC is the length of the ASCII string
DE points to where the data gets converted

**RAM:**

**Outputs:**

**Registers:** A is 0
HL is the size of the new string
BC is also the size of the new string
DE points to the end of the converted data

**RAM:**

**Destroys:**

**Registers:**

**RAM:**

**Notes:** This is not yet used by βαtι*i*b even though it is included. I will likely change this, so do not use it yet.

## TokenToASCII

40BAh

**Description:** This converts a string of tokens that use the space, numbers, and upper- or lower-case letters to ASCII

**Inputs:**

**Registers:** B is the string length
DE points to where it gets converted to
HL points to the string to convert

**RAM:**

**Outputs:**

**Registers:** HL points to the end of the original string
DE points to the end of where it was converted
BC is the size of the new string

**RAM:**

**Destroys:**

**Registers:** A

**RAM:**

**Notes:** βαtιib uses this so that programs or appvars with lowercase letters can be accessed.

## COPxtoOPy

**Description:** OPx and OPy represent OP registers. These routines copy the first 9 bytes of OPx to Opy.
**This applies to COPxtoDE and HLtoOPy

**Inputs:**

**Registers:**

**RAM:**

**Outputs:**

**Registers:** HL points to OPx+9
DE points to OPy+9
BC is 0
A is not affected

**RAM:** The first 9 bytes of OPx are copied to OPy.

**Destroys:**

**Registers:**

**RAM:**

**Notes:** If you can use these instead of a B_Call, use 'em.

## Copy9b

41B0h

**Description:** This copies 9 bytes from HL at DE

**Inputs:**

**Registers:** HL points to the 9 bytes to copy
DE points to where the data gets copied

**RAM:**

**Outputs:**

**Registers:** 9 is added to HL
9 is added to DE
BC is 0
A is not affected

**RAM:** The nine bytes of data at HL are copied to the location at DE

**Destroys:**

**Registers:**

**RAM:**

**Notes:** Calling this saves two bytes of code...

## MakeAnsList

40ABh

**Description:** Makes Ans a list with DE elements

**Inputs:**

**Registers:** DE

**RAM:**

**Outputs:**

**Registers:** HL points to SymEntry
DE points to data start (not size bytes)

**RAM:** OP4 is the var name (Ans)

**Destroys:**

**Registers:** A, BC

**RAM:** OP1, OP2

**Notes:**

## MakeAnsStrX

40BDh

**Description:** Makes Ans a string.

**Inputs:**

**Registers:** BC is the length of the string to be made.

**RAM:**

**Outputs:**

**Registers:** HL points to the SymEntry
DE points to the data start (not the size bytes)

**RAM:**

**Destroys:**

**Registers:** A,BC

**RAM:**

**Notes:**

## PlayNote(HL)

4111h

**Description:** This plays a note based on the value at (HL)

**Inputs:**

**Registers:** HL points to the note

**RAM:**

**Outputs:**

**Registers:** A is 0
E is the value of the note played
D is not changed
HL is incremented by 1

**RAM:**

**Destroys:**

**Registers:** BC (actually, I am just feeling too lazy to analyse at the moment)

**RAM:**

**Notes:** Headphones need to be plugged into the serial port to hear the noise.
Values at (HL) should be a value from 01h to 7Fh. Values above 80h do not create a noise (for the sake of a pause) and 00h will cause an infinite loop. Well, actually, the loop might not be infinite, but it will probably last for a very long time (meaning an hour or two).

## PlayNoteA

4114h

**Description:** Plays the note value of A

**Inputs:**

**Registers:** A is the note to play

**RAM:**

**Outputs:**

**Registers:** Same as PlayNote(HL)

**RAM:**

**Destroys:**

**Registers:** BC?

**RAM:**

**Notes:** See PlayNote(HL)

## Miscellaneous

### HardExit1

40B4h

**Description:** This is kind of an emergency exit usually used when an argument isn't there to avoid a crash. For example, if Str2 is an argument, but Str2 doesn't exist, calling this exits the program.

**Inputs:**

**Registers:**

**RAM:** SPReset
OP2 is a number or the name of a temp var

**Outputs:**

**Registers:** SP points to a safe location

**RAM:**

**Destroys:**

**Registers:**

**RAM:** OP2 is copied to OP1

**Notes:** SPReset should be defined at the very beginning of the program to use this. βαtιib defines this, so using this will work.

## BatLibInstall

41ECh

**Description:** This installs the BatLib parser hook and any calls that need to be in RAM are loaded at 9A01h (to remain compatible with Celtic 3).

**Inputs:**

**Registers:**

**RAM:**

**Outputs:**

**Registers:** A is the flash page the hook is on
B is the flash page as well
C is 0
DE points to the byte after where all the calls were stored (in RAM). This should be the last byte of AppBackUpScreen.
HL points to the hook

**RAM:** 9A01h to 9B71h contains calls that need to be in RAM
800Fh through 8017h is data used to store hook data for transfering hooks.

**Destroys:**

**Registers:**

**RAM:**

**Notes:** -If you use 800Fh to 8017h, either restore the data or set 8011h, 8014h, 8017h to 0
-If Celtic 3 is installed, all data on AppBackUpScreen is used
-9B71h is the flashpage the app is on (in case you need to juggle some flash pages)
-For the most part, the calls copied to RAM deal with reading archive which is why they cannot be on the page they are swapping out :D