Project.....SpriteLib
Program.....βatιib
Author.....Zeda Elnara (ThunderBolt)
E-mail.....xedaelnara@gmail.com
Size.....7213 bytes
Language.....English
Programming.....Assembly
Version.....4.00.00.beta
Last Update.....15 January 2011

Intro

SpriteLib has at last made it. It started as a BASIC program called Font Editor that involved some manual hook juggling with Celtic 3 and Omnicalc. Then, slowly, it evolved into a simple assembly program that still relied on Omnicalc. Then it turned into SpriteLib, the full fledged, independent assembly program. And now, after two years, it is an Application.

Getting Started

-Send βatιib to your calculator
-Select βatιib from the Apps menu. A pretty screen should pop up. Just press clear to exit the menu.
-Use numbers 1, 2, or 3 to view hooks. You can use up/down, too, but the keys register pretty fast :P
*At the moment the menu is not complete, but hopefully I will let the user manage installed hooks in the future (like massive parser chaining :P)

OR

-Send βatιib and prgmZINSTALL to your calculator
-Run 3:Asm(prgmZINSTALL

Now feel free to play with βatιib's many commands :D

Information

If this is your first time using an assembly library, you should read this section.

If you have used assembly libraries like Celtic 3, xLIB, or Omnicalc, you can skim through this section.

Syntaxes

All functions run using the **sum(** functions. For example, **sum(0)** executes command 0 (DisableFont). Using sum(the normal way still works.

Most commands use multiple arguments. For example, GetVar needs the name of the var and type using this format: **sum(21**,"VarName",Type**)**. If you want to store the contents of prgmSPIDER to Ans as a string, you can do: **sum(21**,"SPIDER",5**)**. For a list of types, go here.

The terms 'token', 'hex', and 'ASCII' come up several times. An example of a **token** is "sum(" or "A" or any of the commands or variables in the TI-OS. Each of these tokens is represented by

hexadecimal to the calc. For example, "A" is 41 and "sum(" is B6. ASCII is all individual characters. These also have a hex representation. For example, "sum(" is made up of 4 characters. The ASCII char for 5F is "_" and the token for 5F is "prgm"

Blahblahblah, bored yet? I am... ;p

Binary and Hex

*You will probably be better off using the internet...

In our number system, we count to nine and then we go to two digit numbers.In hex, you count to 15 before going to two digit numbers. After counting 0~9, you use letters A~F, so 10 is really A and 14 is E. Now to really understand hex, you must learn about binary. In binary, you count to 1 before going to two digit numbers. So 0 is 0b and 1 is 1b, but 2 is 10b. So how does this help with Hex? Count to 17 and I'll show you:

```
Dec|Binary|Hex
+--+------+---
  0|     0|0
  1|     1|1
  2|    10|2
  3|    11|3
  4|   100|4
  5|   101|5
  6|   110|6
  7|   111|7
  8|  1000|8
  9|  1001|9
 10|  1010|A
 11|  1011|B
 12|  1100|C
 13|  1101|D
 14|  1110|E
 15|  1111|F
 16| 10000| 10
 17| 10001| 11
```

If you split the binary into groups of 4 digits, you can find the hex. So take 17:
1 0001

The first group equals 1 and the second group does as well. So, the hex for 17 is 11h. For 217, here is the binary:
1101 1001

Looking at the chart, 1101b is D and 1001b is 9. So, 217=D9h.
==========

If you want a mathematical approach to this, here you go. If you break down the number 217, you get:

100x2 plus 10x1 plus 1x7

In other words, you get:
7 times 10^0 (for the ones place)
1 times 10^1 (for the tens place)
2 times 10^2 (for the hundreds place)

In binary, 217=11011001. That can be rewritten as this:
1 times 2^0 (or 1x1) =1
0 times 2^1 (or 0x2) =0
0 times 2^2 (or 0x4) =0
1 times 2^3 (or 1x8) =8
1 times 2^4 (or 1x16)=16
0 times 2^5 (or 0x32)=0
1 times 2^6 (or 1x64)=64
1 times 2^7 (or 1x128) =128

If you add 1+8+16+64+128, you get 217.

In hexadecimal, 217=D9. That can be rewritten as:
9 times 16^0 (or 9x1)=9
D times 16^1 (or 13x16)=208

Add 9 and 208 and you get 217.
==========

Now that that is over, how is this information used in this program? For sprites (see the "Sprites" section), data is stored with pixels. A darkened pixel can be represented with a 1 and a light pixel can be represented with a 0. Convert this data to hex and then compress it and you can compress the data by 1/8. For example, here is a circle:

```
00111100  3C    1111
01000010  42   1    1
10000001  81  1      1
10000001  81  1      1
10000001  81  1      1
10000001  81  1      1
01000010  42   1    1
00111100  3C    1111
```

The data in hex is 3C4281818181423C

Another use is for when you store and recall data. 233 uses 3 bytes if stored that way and it uses 9 bytes if stored to a var. If you store it as a byte (a byte is two hex digits) you use just that-- a byte.
==========

Again, you will probably have more luck checking the internet for a tutorial if you don't understand binary and hex.

Sprites

So, what is a sprite? A sprite is like a picture that uses pixels. All these letters on this screen are sprites as is the cursor and all the icons on your desktop. In games, a sprite is typically animated or designed to represent an object. As an example, if you have ever played Pokémon, your character is represented by a sprite. Since the calculator uses only black and white pixels, sprites can be represented as a bunch of ones and zeros. Convert these ones and zeroS to hex and then to bytes and you will have succesfully created sprite data.

For this program, sprite data is formed in rows. If a sprite is two bytes wide (16 pixels), you would find the sprite data going left and right and then down. When you have converted the sprite to hex, use HexToken to convert it to bytes.

To animate a sprite, you can use the coordinates. For example, in a program you can make it so that pressing right adds 1 to the X coordinate.
There are several methods of displaying sprites that are useful in different situations.

```
============/
OR Logic   /
==========/
```

OR logic means that the sprite is displayed without turning pixels off. In other words, if only 1 bit is 1, the result is 1. For example:

```
00101001  Data already on the screen
00111100  Sprite data
00111101  Result.
```

The result is only 0 (pixel off) if both bits are 0.

```
============/
AND Logic  /
==========/
```

AND logic means that the both bits need to be 1 in order for result to be 1. If even 1 bit is 0, the result is 0. Think of it like multiplication of the bits. 1x1 is the only way to get 1. Using the same example:

```
00101001  Data already on the screen
00111100  Sprite data
00101000  Result.
```

The result is only 1 (pixel on) if both bits are 1.

```
============/
XOR Logic  /
==========/
```

XOR logic returns a 1 if both bits are different. Using XOR two times results in no change to the original data. Par exemple:

00101001 Data already on the screen
00111100 Sprite data
00010101 Result.

Now do it again:

00010101 New data on the screen
00111100 Sprite data
00101001 Result.

As you can see, the result is identical to the original data. If both bits are the same, the result is 0.
==========/
Overwrite /
==========/
Overwrite does just that-it overwrites the data ignoring old values. With the DPutSprite command, the sprite kind of uses the overwrite method, but at the same time it doesn't. The screen data for the graph, when it is displayed, is stored in two places: the graph buffer and the actual screen. The LCD screen has its own memory seperate from the calculator, so editing this memory will change what appears on the sreen without changing the buffer data. Updating the buffer will restore the screen.

Data Editing

This program uses data in the form of bytes for many of its inputs. There are also a few commands that directly edit parts of the memory (as defined by the user). This data is typically not in the form of data you are used to, but it uses a small fraction of the memory and is much, much faster to use. When reading something like program data, do not assume each token is one byte (a token is something like "Goto " or "→" et cetera) because some are in fact two bytes. If you are reading sprite data, you should know how many bytes to read (ie. 8 bytes for an 8x8 sprite) as well as the offset. Data editing features really should be used only by those who understand what they are doing. Bad inputs could result in very undesired results. Reading data should not be a problem, but writing should be done with care.

The HexToken and TokenHex commands are two commands that I have found particularly useful. HexToken will convert a string of Hex to its Tokens, so you can use this to do things like making hacked vars, illegal (meaning not normally possible) strings, et cetera. TokenHex does the exact opposite by splitting up the tokens into its hex. Hacked vars? Illegal strings? What is this?

Hacked Vars:
If you want more than just 10 strings to work with, you can actually type the hex values of a hacked string. String tokens (as well as most variables) use 2 bytes. Strings start with AA and are followed with the string number. Str1 is AA00, Str2 is AA01, et cetera. After AA09 (Str0), the vars don't have real names, so Str32 is actually represented by a lowercase "a" (which I find aesthetically pleasing in a program). Here is a chart of token types:

```
-------------
Type      Token
-------------
Matrix    5Ch
List      5Dh
Y-Var     5Eh
Pic       60h
GDB       61h
Stat      62h
Zoom      63h
Command   7Eh
Strings   AAh
Chars     BBh
More      EFh

Illegal Strings:
 Illegal strings are strings that contain the newline token, the → token, or a
quotation mark. Here are the token equates:

2A   "
04   →
3F   Newline
```

Features

Hooks: βatʟib has a few key features that make it neat. βatʟib uses a parser hook just like many other libraries (like Omnicalc, xLIB, and Celtic 3), but what makes βatʟib special is its ability to chain its parser with any other parser. So, say you want to still use Omnicalc functions. All you need to do is first install Omnicalc and then run βatʟib. Now you can use both Omnicalc and βatʟib functions! That being said, it is possible that another parser might use some of the same RAM as βatʟib and that might cause some problems.

Stringing: βatʟib also has the ability to string functions. What this means is that instead of doing this:

```
:sum(1
:sum(2
:sum(3,0,sum(5,"01020304050607
```

You can string all of the arguments together all at once, saving memory and speed:

```
:sum(1,2,3,0,sum(5,"01020304050607
```

That turns off the run indicator, enables the font hook, and then changes the font data for "0"

Control: βatʟib provides BASIC programmers with a lot of control over their calc and with this comes some danger. For example, command 17 gives access to editing the 32kb of memory. 8kb of that is a bunch of system information (like the contents of the graph screen).

Memory: I have tried to make commands that let the user do a lot with a little bit of data. For example, many of the sprite routines use tokenized data as opposed to hex data so the input strings are half the size. Also, most of the functions that read from memory can handle archived data (I still need to update a few commands...), so this can free up some memory.

Errors: When detected, a string is output to tell the user what kind of error occured. For example, if a variable isn't found, ".BAD NAME" or if the height of a sprite is too big, ".BAD HEIGHT" is returned. See the Error section for more info.

Hmm, I think it is time to have fun.... !!!

βatʟ *i* b Commands

*Offset is always 0 for the first byte/sprite/et cetera unless otherwise noted

00-Disable Font sum(0)

Turns off all fonts

01-Indicator Off sum(1)

Turns the indicator off

02-ProgFont sum(2)

This enables the βatʟ*i*b program font.

03-SetData sum(3,Char#,"Data")

Char# is a value from 0 to 15
"Data" contains the data for the char

The data in *"Data"* is written as char data for the program font. Every seven bytes is a char. If the string is larger than seven bytes, the next character or characters are edited, too.

04-LoadData sum(4,StartChar,#)

StartChar is a value from 0 to 15 that tells which char to start reading data at.
is how many chars to read

The data is stored as a string in Ans. As an example, using **sum(4,0,16)** will copy all 16 chars to Ans.

05-HexToken sum(5,"Hex")

"Hex" is a string of hex digits to be compressed into tokens

As an example, if *"Hex"* is "31300441" then the output will be "31→A" in Ans
*As a "pro" tip, **sum(5,Str1** will actually modify Str1 and output Str1 as Ans. In other words, **sum(5,Str1** is the same as **sum(5,Str1→Str1**

06-TokenHex sum(6,"String")

"String" is the string to convert to hex

This is pretty much the opposite of HexToken.
*The same "pro" tip aplies here as well.

07-SetMap sum(7,"MapData")

"MapData" contains the map data

The result of GetMap is the proper data to use. This is a fast way to output a 64 byte, 16x8 tilemap.

08-GetMap sum(8)

This converts the data on the home screen to a 64 byte tilemap for use with SetMap. The data is stored in Ans as a string.

09-GetTile sum(9,Y,X)

Y is a value from 0 to 7 that is the Y coordinate on the home screen
X is a value from 0 to 15 that is the X coordinate on the home screen

This should be used to get the tile number in the tilemap. The value is 0 to 15 (unless, of course, you aren't using a tilemap :D).

10-GetKeyGroup sum(10,GroupValue)

GroupValue is a number from 1 to 127. This determines which key groups to test for. To determine the key group, select the values from the chart and add them together:

Value:	1	2	4	8	16	32	64	128	
Group 1	Down	Left	Right	Up					
Group 2	Enter	+	-	*	/	^	Clear		
Group 4	(-)	3	6	9	)	tan	Vars		
Group 8	.	2	5	8	(	cos	prgm	stat	
Group 16	0	1	4	7	,	sin	apps	XTON	
Group 32		Sto	ln	log	x^2	x^{-1}	math	Alpha	
Group 64	Graph	Trace	Zoom	Window	Y=	2nd	Mode	Del	

So for example, if I wanted to test for the arrows, I would use Group 1 so **sum(10,1)**. When down is pressed, the result value is 1. If down and up are pressed, 8+1=9 is returned.

If you want to test group 64 and 1, keep in mind that pressing either Graph or Down returns 1.

This waits for a key in the group or groups to be pressed

11-GetBytes sum(11,Offset,"VarName",Type,#Bytes)

Offset is how far into the variable to start reading at starting at 0
"VarName" is the name of the variable to read from
Type is the variable type. See this chart for data types.
#Bytes is how many bytes to read

If the appvar "TILEMAPS" had a bunch of 64 byte tilemaps and I wanted to get the data for the third one, then the first would be at offset 0, the second at offset 64, the third at offset 128. Appvars are type 21 and 64 bytes are to be read: **sum(11,128,"TILEMAPS",21,64**

12-StoBytes sum(12,Offset,"VarName",Type,"Data")

Offset is how far into the variable to start reading at starting at 0
"VarName" is the name of the variable to read from
Type is the variable type. See this chart for data types.
"Data" is a string of data to copy starting at the offset

Okee, so say I want to use the current homescreen as a tilemap and I wanted to save it. If Appvar "TILEMAPS" has the tilemap data and I want to save it as the third tilemap:
sum(12,128,"TILEMAPS",21,sum(8

13-TileMap sum(13,Logic,"SpriteData","MapData")

Logic is the type of sprite logic to use:
0=AND
1=XOR
2=OR
3=Overwrite
"SpriteData" is a string of tile data. Every eight bytes is a tile.
"MapData" is a 96 byte string of data that is the tilemap

Unfortunately, I have a tough time explaining how tilemaps work, so sorry in advance.

Tilemap data in βatιb is made going down first then right. Each byte is a tile and tilemaps are 8x12, so the first 8 bytes are the first column.

Each tile is 8x8 and each byte in the tilemap tells which sprite to display. So, if a tile was 02h, then the third sprite is displayed at the current position.

14-VarEditByte sum(14,"VarName",Type,Offset,Value)

"VarName" is the name of the var
Type is the type of the var
Offset is the byte to edit
Value is the value to replace the byte with

Ans is the old value of the byte

15-VarReadByte sum(15,"VarName",Type,Offset)

"VarName" is the name of the var
Type is the type of the var
Offset is the byte to read

Ans is the value of the byte at the offset

16-VarDrawSprite sum(16,Logic,"VarName",Type,Offset,X,Y,Width,Height)

Logic is the method of drawing the sprite
"VarName" is the name of the var
Type is the type of the var
Offset is the byte to edit
X is a number from 0 to 11 that is the X-coordinate at which the sprite is drawn
Y is a number from 0 to 7 that is the Y-coordinate at which the sprite is drawn
Width is a number from 1 to 12. This is the width of the sprite in bytes.
Height is a number from 1 to 64. This is the height of the sprite.

*If you let the sprite go off the screen, you risk crashing your calc.

17-MemEdit sum(17,Offset,"Data")

Offset is the offset into memory. 0 is the start of RAM.
"Data" is a string of data to copy to the memory location.

The data is swapped, so Ans will contain what was at that address.
At the moment, βatιb is using the first 22 bytes of RAM for some data as well as OP5, OP6, and the remaining bytes of appBackUpScreen that Celtic 3 isn't using.
For everybody who went "huh?" I will give these offsets:
1772-768 bytes of memory. This gets cleared when the calc APD's. If you use command 22 or 23, 29, and others the bytes of code get copied here, too.
2618-531 bytes of memory used by the StatVars. Do not use stat vars if you store data here.
4928-768 bytes that is the graph screen :D

18-MemRead sum(18,Offset,Size)

Offset is the offset into RAM (just like before)
Size is how many bytes to read

Ans contains the data read.

19-EditByte sum(19,Offset,Value)

Offset is the offset into RAM
Value is a value from 0 to 255 that is the value to write

Ans contains the previous value

20-ReadByte sum(20,Offset)

Offset is the offset into RAM

Ans is the value of the byte read

21-GetVar sum(21,"VarName",Type)

"VarName" is the name of the var
Type is the var type

Ans is a string containing the contents of the variable. This does not work properly for lists, matrices, or real numbers. For all others (like strings or programs), this will work even if the variable is archived.

22-ASMHex sum(22,"Hex")

"Hex" is a string of hex data that is to be executed as an assembly opcode.

As an example, **sum(22,**"EF4045") will clear the LCD. If you are like me and you prefer to program in hex, this is for you: Further down in this document is a list of most of the calls in βatιb (maybe all, I'm not sure :P). Across from the name is the address for the calls. The address actually points to a table, so they will remain compatible with future versions... Hopefully.
The code is executed from address 86ECh

23-ASMToken sum(23,"TokenHex")

"TokenHex" is a string of tokenized hex data.

As an example, to clear the LCD, do **sum(23,sum(5,**"EF4045

24-DPutSprite sum(24,Width,Height,Y,X,"Data")

Width is a value from 1 to 12 that is the width of the sprite in bytes.
Height is a value from 1 to 64 that is the height of the sprite in bytes.
Y is a value from 0 to 63 that is the Y coordinate of the sprite
X is a value from 0 to 11 that is the X coordinate of the sprite
"Data" is a string of data for the sprite

This draws a sprite directly to the LCD without drawing to a buffer, so it has several "side effects."
-Updating the LCD removes the sprite. This makes this function ideal for sprites that have to move as it does not affect the actual graph screen.
-Sprites will wrap around if they go off the edge of the screen.

As an example, this is my prettyful tree I designed for an RPG (it is 16x16):
sum(24,2,16,0,0,sum(5,"03C00D7010A82054402A4016802B8015802B405630AC0E70018001800
3C007E0

I used the sum(5 to tokenize it. Issa pretty tree!? ^-^

25-VarType sum(25,NewType,"VarName",Type)

NewType this is the variable type you want to change the var to.
"VarName" is the name of the var
Type is the current type of the variable

This is where this chart comes in handy. This can be used to change any variable to another type, however, there are some things to be noted:
-Using the "red" types is discouraged. These data types aren't nice
-Only change a type to a type of the same color. These have similar data structures.
-Blue data types (token variables like string, pic, et cetera) will be changed back to their normal type when they are used.
-Green data types remain changed unless you are changing it when it is archived.
-Changing the type of an archived var is not permanent.
***As a suggestion, do not change a group to a program or appvar and then unarchive it. It won't be a happy group D:

So, as an example, to make Pic1 show up in the string menu:
sum(25,4,"Pic1",7

26-BatteryLevel sum(26)

On OSes 2.30 and up, this returns a value from 0 to 4 indicating the battery level. 0 is low, 4 is high, everything in between is inbetween.

On lower OSes, 0 or 4 is returned. 0 is low, 4 is high.

27-IncContrast sum(27)

This increases the contrast one point unless it is maxed

28-DecContrast sum(28)

This decreases the contrast one point unless it is maxed

29-Rectangle sum(29,X,Width,Y,Height,Type)

X is the X pixel coordinate
Width is the width in pixels of the rectangle
Y is the Y pixel coordinate
Height is the height of the rectangle in pixels.
Type 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
```

***
X+*Width* should not be greater than 96
Y+*Height* should not be greater than 64

Hehe, this is the first routine that uses pixel width and pixel x coordinates ^_^ and the best part is that it doesn't use a single bcall! The whole thing uses a little more than 683 bytes

30-ScreenToGraph sum(30)

This copies the contents of the current screen (what is on the LCD) to the graph screen buffer.

31-DispChar sum(31,Y,X,Char)

Y is a value from 0 to 7 that is the Y coordinate
X is a value from 0 to 15 that is the X coordinate
Char is a number from 0 to 255 that is the ASCII character to display

The coordinates are like output coordinates, minus 1.

32-SetContrast sum(32,Contrast)

Contrast is a value from 0 to 39. 39 sets the darkest contrast and 0 sets the lightest. 24 is about normal.

33-FlagWrite sum(33,Value,Flag)

Value is the value to write to the flag group
Flag is the flag group to edit

Ans contains the old value of the flag group
***See this section on Flag Editing for more info

34-FlagRead sum(34,Flag)

Flag is the flag group to read

Ans contains the value of the flag group

35-GetSprite sum(35,X,Y,Height,Width)

X is a value from 0 to 11 that is the X coordinate location of the sprite.
Y is a value from 0 to 63 that is the Y coordinate location of the sprite.
Height is the height of the sprite. Use 1 to 64
Width is the width of the sprite. Use 1 to 12

This is a rather useful command because it saves you from needing to convert the sprite to hex! This command returns the tokenized string in Ans and the string is the correct format for βatι*i*b. If you use **sum(6** on the data, it is the proper format for the Celtic 3 command **identity(5**.

36-PicHandle sum(36,Function,Pic#)

Pic# is a number from 0 to 255 allowing for hacked pictures. 0 is Pic1.
Function is defined as follows:

- 0-RecallPic: Copies the pic to the graph screen
- 1-StorePic: Copies the current screen to the picture
- 2-DeletePic
- 3-Unarchive
- 4-Archive
- 5-Toggle Archived

37-OutputASCII sum(37,Y,X,"ASCII")

Y is a value from 0 to 7 that is the Y coordinate
X is a value from 0 to 15 that is the X coordinate
"ASCII" is a string that is read and output as ASCII data

This is like BASIC's output function, except the coordinates are one less and the string is read as ASCII, not tokens

38-SubList sum(38,Size,Offset,"Name")

Size is how many elements to read. 1 reads 1 element. Do not use 0.
Offset is which element to start reading at. 0 is the start of the list.
"Name" is the name of the list. For user defined lists, include the little ʟ in the name.

As an example, if ʟEIGHT was {1,1,2,3,5,8,13,21} and I wanted to read the {8,13,21} portion, I would use:
Size=3
Offset=5
Name=ʟEIGHT
sum(3,5,"ʟEIGHT")
This works even if the list is in archive :D Yays! \(^-^)/

39-Z-Address sum(39,z)

z is a value from 0 to 63 that determines how far up to shift the screen.

As an example, if you rotate the screen up 8 pixels, the top 8 pixels appear on the bottom :D
This has been included in EnLib and EnPro (called ScrollScreen), but I very specifically omitted it from βatιb. I figured it was cool, but who was going to use it? Then, one day I saw a request to add z-addressing to βatιb. When I found out what that was, I started laughing. I finally added it ^_^

40-BASIC ReCode sum(40)

This starts a ReCode block.
THIS FUNCTION DOES NOT WORK YET. There have been too many changes that I haven't kept up to date with *blushes*.

41-GetStats sum(41,"VarName",Type)

"VarName" is the name of the variable
Type is the variable type.

This returns a three element list with the information {Size,Type,Flash}:
- Size is how many bytes of data the variable has
- Type is the variable type. This is useful if you are not sure, specifically if a program is a regular or protected program or if a number or list is real or complex.
- Flash is 0 if the var is in RAM. Otherwise, it is the flash page it is on.

This works with strings, programs, and appvars. Others may crash.

All values are returned with an even amount of digits (unless it is 0), so you will see results like {0123,05,0}, but the leading 0 is harmless. 0123 is still 123 and it does not use any extra memory.

If the var does not exist, the type is returned in Ans instead of a list...

42-AnsType sum(42)

This returns the type that Ans is. Use this with something like **If** so that Ans isn't modified. So for example, if you try to use SubList on L2:

```
sum(38,2,3,"L2
If 1≠sum(42
Then
Pause "GAHH!NO L2?!?!
Stop
End
Pause "PHEW,L2 EXISTS!
```

43-Get2Key sum(43)

Returns a number from 0 to 3247 representing the key press. If you are familiar with the xLIB/Celtic 3 real(8 command (GetKey), then pressing 1 key returns the same values. Pressing multiple keys... that returns something else. You will probably need to test it yourself.

For two keys: HighestKey*57+NextHighestKey

44-PlayData sum(44,Duration,"Data")

Duration is the length of each note. 256 is fast, 65535 is really slow
"Data" is a string of tokenized sound data

*For sound data, 80h and above are pauses. Do not use 00h. Ever. 01h~7Fh make noise.
*You will need headphones to hear the noise. I got two pairs of headphones, both with an adapter at Wal-Mart. Both are designed to work with cell phones (2.5mm), but one uses the adapter to plug into a cellphone (or in my case, a TI-84+SE) and the other uses an adapter to plug into a regular headphone jack. I also got an adapter at Radio Shack ^_^

Anywho, as an example:
sum(44,4096,**sum(5,**"323225251A1A11110A0A0505
Hmm...sounds familiar... maybe if I ever get around to Pokémon Amber I can included this noise somewhere...

45-GetChar sum(45,Y,X)

Y is a value from 0 to 7 that is the homescreen Y coordinate
X is a value from 0 to 15 that is the homescreen X coordinate

Ans is the ASCII value of the char at location (Y,X)

46-PortEdit sum(46,Port,Value)

Port is the port number to edit
Value is the value to write to the port

Ans is a two element list where element one is the read value before writing to the port and element 2 is the value after writing to the port.
THIS COMMAND DOES NOT WORK YET!!! It only modifies port 0 for some odd reason, so always make sure *Value* is 0!
Gahh! The code itself works perfectly fine... so long as it is in RAM for some reason "(If it isn't in RAM, it only uses port 0.

47-PortRead sum(47,Port)

Port is the port to read.

Ans is the value read from the port
*This works.

48-ScreenShot sum(48)

This installs a key hook. When you are in the TI-OS, like in a menu or in the program editor, press [2nd][.] (the imaginary "i") and the currently displayed image will be copied to the graph screen.

49-SpeedyKeys sum(49,Pause,Delay)

Pause is the delay before repeats start. Use 2 to 50. 50 is default
Delay is the delay between key repeats. Use 1 through 10 and less than *Pause*.

This installs a keyhook. **Note** that only one key hook works at a time. Setting a keyhook overwrites the previous one.

As an example, I like to use **sum(49**,11,3. Try scrolling through the catalog menu...

50-Uninstall sum(50)

This uninstalls the βatι*i*b Parser Hook (which executes all of these sum(commands). Any font hooks or key hooks remain active, though.

51-DisableKeyHooks sum(51)

This disables any active key hooks.

52-HexSprite sum(52,"Hex",Height,X,Y,Logic)

"Hex" is the hex data for the sprite.
Height is the height of the sprite in pixels.
X is the X coordinate of the sprite. Use 0 to 11
Y is the Y coordinate of the sprite. Use 0 to 63
Logic is the method of drawing the sprite:

- 0-Overwrite
- 1-AND
- 2-XOR
- 3-OR

There is no width input because it uses the length of the data divided by the height to find the width (area/width=height). (it rounds up if there isn't enough data :P)

53-TokenSprite sum(53,"Data",Height,X,Y,Logic)

"Data" is the tokenized data for the sprite.
Height is the height of the sprite in pixels.
X is the X coordinate of the sprite. Use 0 to 11
Y is the Y coordinate of the sprite. Use 0 to 63
Logic is the method of drawing the sprite:

- 0-Overwrite
- 1-AND
- 2-XOR
- 3-OR

54-DBRead sum(54,LineByte,Line,"VarName",Type)

LineByte is the byte value that represents the start of a line of data.
Line is the line number to read. The first line is 1.
"VarName" is the name of the database var
Type is the type of the var

If you make the value of *LineByte* 63, you will effectively make the Celtic 3 LineRead command :P (63 is the value for the BASIC newline token).

I made this command in case you choose to use a different value. At the moment, for reading lines in BASIC programs, I would use Celtic 3. This command cannot read the last line properly and Line 0 returns .BAD NUMBER as opposed to the number of lines in the file (which Celtic 3 does).

THIS COMMAND IS NOT COMPLETE. There is a good chance that the syntax will change in a future version.

***As a random use, if you happen to know that there are 4 programs that start with ":RAH!" and you happen to have Celtic 3 installed, too:

```
:det(9,":RAH!
:sum(54,41,x,"Ans",4
```

Will get the xth program name.

***As another random use, if you use Celtic 3 to convert a list to a string, you can use a value of 43 to read the element number :P

55-SetFontHook sum(55,FSType,Type,Offset,"VarName")

FSType FSType is the fontset type

0-Experimental 6x8 font. Currently, this is not the best of fonts for navigating the OS as it is a little buggy in menus (It doesn't like to display all the programs/apps). 8 bytes per character

1-5x7 font. This uses the same data as the previous font. It is meant as a temporary work around while I work out the kinks in the 6x8 routine.

2-5x7 font. This is made to be compatible with the Omnicalc fontset format. Use an offset of 11 for Omnicalc fonts.

3-ProgFont2. This is the 6x8 version of ProgFont (**sum(2)**). This modifies hexadecimal characters during program execution. The fontset is organized in 8 byte increments starting with the character replacement for '0'.

4-ProgFont3. This is an experimental 8x8 font that modifies hexadecimal characters during program execution. This uses the same format as the 6x8 fonts.

5-Experimental 8x8 font. Since this font allows for only 12 chars per line, some things will go off the screen

Type is the type of the var
Offset is the offset into the var where the font data is
"VarName" is the name of the var with the font data

NOTE: When using an 8x8 font, there are only 12 chars to a line (not 16), so you wont see the last 4 chars.

56-Draw sum(56,Func,[Arg1,...)

Func is the drawing function to use:

0-PixelTest
Arg1 is the Y pixel coordinate
Arg2 is the X pixel coordinate
This returns "0" if the pixel is off

1-PixelOff
Arg1 is the Y pixel coordinate
Arg2 is the X pixel coordinate
This returns "0" if the pixel was off before turning it off

2-PixelOn
Arg1 is the Y pixel coordinate
Arg2 is the X pixel coordinate
This returns "0" if the pixel was off before turning it on

3-PixelInvert
Arg1 is the Y pixel coordinate
Arg2 is the X pixel coordinate
This returns "0" if the pixel was off before inverting it

4-FillBufOff
This clears the graph screen without forcing a redraw

5-FillBufOn
This turns the graph screen black

6-FillBufInvert
This inverts the graph screen

7-PixelHorizOff
Arg1 is the Y pixel coordinate
This clears a horizontal line of pixels

8-PixelHorizOn
Arg1 is the Y pixel coordinate
This turns a horizontal line of pixels on

9-PixelHorizInvert
Arg1 is the Y pixel coordinate
This inverts a horizontal line of pixels

10-PixelVertOff
Arg1 is the X pixel coordinate
This clears a vertical line of pixels

11-PixelVertOn
Arg1 is the X pixel coordinate
This turns a vertical line of pixels on

12-PixelVertInvert
Arg1 is the X pixel coordinate
This inverts a vertical line of pixels

When it says "returns 0 if the pixel was off" it does not necessarily return 1 if the pixel was on. For functions one through three, they serve double duty by doing a pixel test followed by changing the pixel. If you do not want to do a pixel test, use the OS routines to save a few bytes :P

57-GetVersion sum(57)

Returns a string telling which SpriteLib version is installed...
I wouldn't rely on this...
It requires me to remember to change some bytes every update...
I cannot even remember to update the readme half the time...
:P

58-ShiftScreen sum(58,NumShifts)

NumShifts is the number of times to shift the graph screen

**Currently only shifts right
THIS COMMAND IS NOT FINISHED. I still need to add a "direction" argument

59-BaseX sum(59,"Hex",Base,Size)

"Hex" is a hexadecimal string of digits (up to 62 digits)
Base is the base to convert the hex to
Size is the size of the string to return (in case a specific number of digits is required)

THIS COMMAND IS NOT FINISHED. It will eventually handle inputs in other bases as well as real input/output as an option. Also, the size argument does nothing at the moment

??-DispGraphBuffer sum(??)

The displays the graph screen. This is used to update the screen after using a sprite or tilemap command.

This is executed when a number is used beyond the number of commands used. So, if 51 was the last command, using anything from 52 to 255 would execute this command.
For compatability reasons, use 99 to execute this command. If for some crazy reason I come up with more than 99 commands, I will make sure that this is command 99 and some other command is executed last.

BASIC ReCode

Erm, BASIC ReCode is still under development, so many of the commands and syntaxes are very likely to change. That being said, if you include a ReCode block in your programs, don't expect it to work properly in future versions. For now, though, you can use it to get a feel for how it works.

Stop

```
Stop
```

This ends a ReCode block.

Pause

```
Pause xx
```

xx is a value from 0 to 65535 determining how long to pause

If xx is a value of 100, it pauses for a second on a TI-84+ SE
0 is the equivalent of 65536, so unless you want to pause for 655.36 seconds, try not to use 0...

Ans

```
Ans
```

This will store the last converted value into the BASIC Ans variable.

**This will likely change in future versions of ReCode. I only made it so that I could make sure the math functions worked properly :D

Disp

```
Disp Y,X,<<String>>
```

Y is a value from 0 to 7 representing the Y home screen coordinate
X is a value from 0 to 15 representing the X home screen coordinate
<<String>> is an ASCII string **without** quotation marks.

Because it is an ASCII string, certain chars do not match up. For example, a space will be displayed as) instead. Also, because the string only ends at a newline, '?' cannot be used.

I'm thinking I should make Output(convert the chars...

Line(

line(T,L,U,R,D

T is the type of rectangle to draw
0-White rectangle
1-Black rectangle
2-Inverted rectangle
3-Rectangle Border (Blegh)
4-White rectangle border (Blegh)
5-Rectangle border, white inside
L is the left-most coordinate
U is the upper coordinate
R is the right-most coordinate
D is the lower coordinate

This draws a rectangle on the screen. I think this command will change a little in future versions.
THIS DOES NOT WORK CURRENTLY. This command is under reconstruction and may take a while before I finish.

AsmPrgm

AsmPrgm<<opcode>>

<<opcode>> is an assembly opcode to execute. The opcode can be up to 767 bytes long and does not need a C9. It is executed from address 86ECh.

This might change in the future to allow for an opcode that spans multiple lines and allows for defining addresses and labels. Heck, I might even make it allow for straight mnemonics.

Math Functions

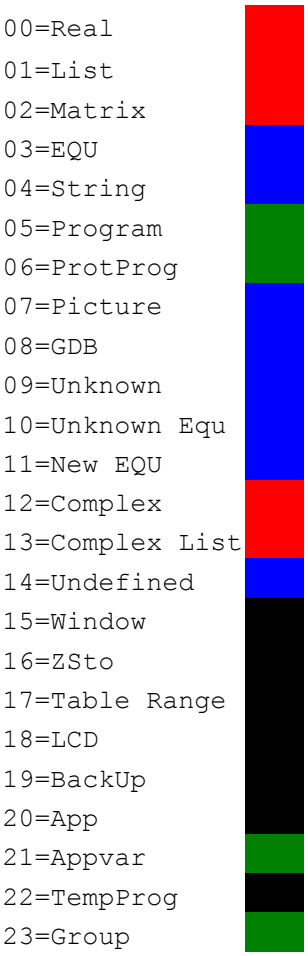
+-*/

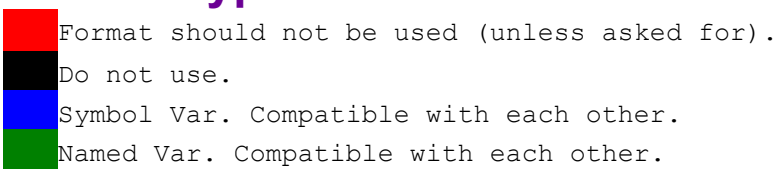
+ adds the value following with the previously computed value
- subtracts the following value from the previously computed value
*** multiplies the following value with the previously computed value
/ divides the following value into the previously computed value

At the moment functions are computed right to left. I was told that Axe did it the opposite direction, so I might change it in a future version since so many people are used to the Axe method. As an example:
100+3*36/**15-6**
100+3***36/9**
100+**3*4**
100+12
112

Info

Data Types





Flag Editing

Flags are bits of data. Literally bits of data. A flag is either off or on (0 or 1). Since there are 8 bits to a byte, there are as many as 8 flags per byte. Each flag has something to do with a system setting. For example, there are flags that determine if the Axes are on or off, there is a flag that tells if the on button was pressed, and there is a flag that determines if lowercase letters are activated. Info on the flags can be found in Flags.text. The info was found in a ti83plus.inc I found online (annotated by Brandon Wilson). So anyway, example time. Let's take a look at flag group 4:

```
grfDBFlags              equ 4h
grfDot                  equ 0           ;0=line, 1=dot
grfSimul                equ 1           ;0=sequential, 1=simultaneous
grfGrid                 equ 2           ;0=no grid, 1=grid
grfPolar                equ 3           ;0=rectangular, 1=polar coordinates
grfNoCoord              equ 4           ;0=display coordinates, 1=off
grfNoAxis               equ 5           ;0=axis, 1=no axis
grfLabel                equ 6           ;0=off, 1=axis label
```

"equ 4h" tells us that this is flag group 4. all the other "equ " things tell which bit corresponds to

which flags. So if I wanted to do Connected, Sequential, GridOff, Rectangular, CoordOff, AxesOff, and LabelOff, then I would set the bits to 0100000. Convert that to decimal and you get 32. So, using **sum(33**,32,4 you can set seven different modes. The best part is that this also returns in Ans what the previous flag settings were. Save that value and you can restore the settings at the end of the program without using a GDB variable.

Errors

```
.BAD NUMBER
     This is a general error given when an input value is out of range.
.BAD INPUT
     This is usually output when a non-number argument is expected and not found
.BAD NAME
     Indicates that the var was not found or is of a bad type.
.NO DATA
     This error occurs to prevent 0-sized outputs (like empty lists/strings)
.TOO BIG
     Erm, not used yet...
.LOW MEM
     This indicates that there was not enough memory for the output
.BAD X
     This is output if an X-coordinate is bad (usually).
.BAD Y
     This is output if an Y-coordinate is bad (usually).
.WIDTH
     This is output if the width argument is 0 or too big.
.HEIGHT
     This is output if the height argument is 0 or too big.
```