Project.........SpriteLib
Program.........βαtL𝑖b
Author..........Zeda Elnara
E-mail..........xedaelnara@gmail.com
Size............1-Page App (16002 bytes of 16384 used)
Language........English
Programming.....Assembly
Version.........4.01.123.55
Last Update.....21 June 2012

Intro

So what is BatLib? BatLib is an application designed to give even more functions to BASIC users. There are tons of functions designed for beginners and all the way up to advanced programmers. With over 100 functions **and** a new, fast programming language, you are bound to find something neat. Among the functions you will find: "new, fast programming language" has been modified and is lacking documentation.

- -Custom fonts
- -Sprite routines
- -Key routines (such as for testing multiple keys)
- -Memory editing (for advanced users)
- -Drawing (including a Rectangle routine with 12 fill methods)
- -System Flag editing (for advanced users)
- -Reading directly from archive
- -Easy use of hacked variables
- -New manipulations of lists and matrices (such as SubMatrix)
- -Sound routines
- -BASIC ReCode (a fast, interpreted programming language) lacking documentation.
- -Key hooks (such as SpeedyKeys)
- -Drawing directly to pictures instead of just the graph screen
- -New text and string functions

If you cannot find something cool, feel free to ask!

Getting Started

-First, you will want either BatLib or BatLibG. See the BatLibG readme for differences.
-Send BatLib to your calculator
-Select BatLib 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
*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)

OR

-Send BatLib and prgmZINSTALL to your calculator
-Run 2:Asm(prgmZINSTALL

See the **Main Menu** section for more info
Now feel free to play with BatLib's many commands

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 probably skim through this section.

Syntaxes

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

-Many times variable names are needed as arguments. For example, GetVar returns the contents of a variable as a string and uses the arguments **dim(**21,"VarName"). The names require a prefix byte to tell what var type to read. So to copy the contents of the appvar SPIDER, you use dim(21,"USPIDER") where U is a prefix for appvars. For a list of prefixes, go here.

-The terms 'token', 'hex', and 'ASCII' come up several times. An example of a **token** is "sin(" 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, "dim(" is made up of 4 characters. The ASCII char for 5F is "_" and the token for 5F is "prgm"

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 covered, how is this information used in BatLib? 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 or list. 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.

========/

Erase /

======/

This will erase the sprite data from the screen. For example:

```
00101001  Data already on the screen
00111100  Sprite data
00000001  Result.
```

Anywhere with a 1 in the sprite makes a zero on the screen. Anywhere with a zero in the sprite is ignored. There are several ways to get this effect using bit logic:

- OR the sprite to the screen, then XOR
- Invert the sprite data, then AND to the screen (method used)

=======/

Mask /

=====/

This draws a two-layered sprite with the mask first, then the sprite data. The mask data is put on the screen with AND logic and the sprite data is put on after that with OR logic. The way BatLib uses this is by actually interleaving the two sets of data. For example, if red is the mask and white is sprite:

```
1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0  =C33C
1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0  =8142
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1  =0081
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1  =0081
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1  =0081
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1  =0081
1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0  =8142
1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0  =C33C
```

What this does is draws a circle where everything inside is overwritten.

============/
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 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

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

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

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

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

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

Control: BatLib 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). If you aren't careful, you might clear your RAM with some of these commands.

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.

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, ".HEIGHT" is returned. See the Error section for more info.

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

BatLib Commands

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

00-Disable Font dim(0)

Turns off all font hooks

01-Indicator Off dim(1)

Turns the indicator off. In versions after 4.00.95.51, this disables the Done message, too. After major revision 1, (so 4.01.xx.yy and later), the decimal after the input is an argument:

- 0 turns off the run indicator and disables done
- 1 disables done
- 2 Indicator off
- 3 Indicator on
- 4 Done on

02-ProgFont dim(2)

This enables the BatLib program font.

03-SetData dim(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 dim(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 **dim(4,0,16)** will copy all 16 chars to Ans.

05-HexToken dim(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, **dim(5,Str1** will actually modify Str1 and output Str1 as Ans. In other words, **dim(5,Str1** is the same as **dim(5,Str1→Str1**

06-TokenHex dim(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 dim(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 dim(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 dim(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 dim(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	

For example, if I wanted to test for the arrows, I would use Group 1 so **dim(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. However, if you add 128 to the group value, this will removes the waiting. Also note that ON will not register during while it waits for a key to be pressed.

11-GetBytes dim(11,Offset,"VarName",#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
#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 use a prefix of "U" and 64 bytes are to be read: **dim(11,128,"UTILEMAPS",64**

12-StoBytes dim(12,Offset,"VarName","Data")

Offset is how far into the variable to start reading at. (Starts at 0)
"VarName" is the name of the variable to read from
"Data" is a string of data to copy starting at the offset

This will put the previous data in Ans. As an example, say prgmA has the first line as 000111222333 and you do this:

```
dim(12,3,"EA","AAA
```

Ans will contain the string "111" and prgmA will be 000AAA222333. For a more advanced use, 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:

dim(12,128,"UTILEMAPS",dim(8

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

Logic is the type of sprite logic to use:

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

"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

Tilemap data in BatLib 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 dim(14,"VarName",Offset,Value)

"VarName" is the name 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 dim(15,"VarName",Offset)

"VarName" is the name of the var. This can be archived.
Offset is the byte to read

Ans is the value of the byte at the offset.

16-TileMap2 dim(16,Logic,Pic#,"String"|Str)

Logic is the method of drawing the sprites
Pic# is a value from 0 to 255. This is where the sprites are. 0=Pic1, 1=Pic2...
"String"|Str this is the tilemap data and can either be a string or the number of a string.

To use a picture to store sprites, the sprites are 8x8 and stored from left to right, then down (like with xLIB or Celtic 3). This will only source sprites from one picture var. *All inputs can be archived* :)

17-MemEdit dim(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, BatLib 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 dim(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 dim(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 dim(20,Offset)

Offset is the offset into RAM

Ans is the value of the byte read

21-GetVar dim(21,"VarName")

"VarName" is the name of the var

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 dim(22,"Hex")

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

As an example, **dim(22**,"EF4045") will clear the LCD. The code is executed from address 86ECh

23-ASMToken dim(23,"TokenHex")

"TokenHex" is a string of tokenized hex data.

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

24-DPutSprite dim(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):
dim(24,2,16,0,0,dim(5,"03C00D7010A82054402A4016802B8015802B405630AC0E70018001800 3C007E0

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

25-VarType dim(25,NewType,"VarName")

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:
dim(25,4,"GPic1

26-BatteryLevel dim(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 dim(27)

This increases the contrast one point unless it is maxed

28-DecContrast dim(28)

This decreases the contrast one point unless it is maxed

29-Rectangle dim(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

This routine features full clipping, now

30-ScreenToGraph dim(30)

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

31-DispChar dim(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 dim(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 dim(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 dim(34,Flag)

Flag is the flag group to read

Ans contains the value of the flag group

35-GetSprite dim(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 BatLib. If you use **dim(6** on the data, it is the proper format for the Celtic 3 command **identity(5**.

36-PicHandle dim(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 LCD to the picture (not necessarily the graph).
- 2-DeletePic
- 3-Unarchive
- 4-Archive
- 5-Toggle Archived

37-OutputASCII dim(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 dim(38,Size,Offset,"Name")

Size is how many elements to read. 1 reads 1 element. 0 returns ".NO DATA"
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 L in the name. *If you use a number instead, this will be interpreted as a list number. For example, 0=L1, 1=L2, 2=L3,...*

As an example, if LEIGHT 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=LEIGHT
dim(38,3,5,"LEIGHT")

This works even if the list is archived.

39-Z-Address dim(39,z)

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

As an example, if you rotate the screen up 8 pixels, the top 8 pixels appear on the bottom.

40-BASIC ReCode dim(40)

This starts a ReCode block. See the BASIC ReCode PDF for commands and syntax.
Currently disabled. Sorry for the inconvenience :/

41-GetStats dim(41,[option],"VarName")

option is 1 if you want the adjusted size as you would see in the Mem menu.
"VarName" is the name of the variable

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.

The *Size* element might lead to confusion in some cases. For example, a string with "HELLO" will return a size of 5. In the memory menu, however, it shows up as 16. This is because the OS stores a VAT entry of 9 bytes plus the size of the name (two bytes) and then the data. So 9+1+5=16. For programs, we have a similar issue. If the name is 5 characters, it will be 9+5+n bytes.

This works with all variables, but there are some tricky issues with matrices, lists, and real numbers (and their complex valued counterparts):

Lists will return the number of elements in the list, not the size in bytes

Matrices will return the size as a number in the form 256R+C where C is the number of columns, R is the number of rows.

Real numbers: Just ignore the size, that is the healthiest option for your sanity. It does give info about the number of digits before the decimal, whether it is negative or positive and imaginary or real.

42-AnsType   dim(42)

This returns the type that Ans is. Use this with something like **If** so that Ans isn't modified. This is useful if you want to figure out if a string or number was returned since BatLib returns errors as strings. It has other uses, too, besides error catching :)

43-Get2Key   dim(43)

If a single key is pressed, a value from 1 to 56 is returned. Here is a chart.
For two keys: HighestKey*57+NextHighestKey

For example, if you press Alpha+Mode, Mode is the higher key and is 55, Alpha is 48. So if you press these two, you will get 55*57+48=3135.

44-PlayData   dim(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:
dim(44,4096,**dim(5**,"323225251A1A11110A0A0505
Hmm...sounds familiar... maybe if I ever get around to Pokémon Amber I can included this noise somewhere...

45-GetChar   dim(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   dim(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.

47-PortRead   dim(47,Port)

Port is the port to read.

Ans is the value read from the port

48-ScreenShotHook dim(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-SpeedyKeysHook dim(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 **dim(49**,11,3. Try scrolling through the catalog menu with that and enjoy.

50-Uninstall dim(50)

This uninstalls the BatLib Parser Hook (which executes all of these dim(commands). Any font hooks or key hooks remain active, though.

51-DisableKeyHooks dim(51)

This disables any active key hooks.

52-HexSprite dim(52,[option],"Hex",Height,X,Y,Logic)

option is 1 if you are using pixel coordinates.
"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 (0 to 11 unless using pixel coordinates)
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
- 4-Erase

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). When using the regular draw mode, the X coordinate is multiplied by 8. This is to maintain backwards compatibility with older versions of BatLib.

53-TokenSprite dim(53,[option],"Data",Height,X,Y,Logic)

option is 1 if you are using pixel coordinates.
"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 (unless using pixel coords)
Y is the Y coordinate of the sprite. Use 0 to 63
Logic is the method of drawing the sprite. See 52-HexSprite.

54-DBRead dim(54,LineByte,Line,"VarName")

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. Using 0 will return the number of lines in the file instead of a string.
"VarName" is the name of the database var. **This can be archived.**

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). If the *Line* argument is greater than the number of lines in the file, ".EOF" is returned. Here is an example... Say you have a prgmBLARGH that looks like this:

```
:MEOW?
:HELLO?
:
:RAH!
:GRRR!
```

Doing **dim(54,63,1,"EBLARGH** would return "MEOW?" in Ans. Changing that 1 to a 3 would return ".NO DATA?" as the error and the 5th line will return "GRRR!"

Now, if you changed that 63 to a 41, you would read between spaces, instead:

```
:MEOW? HELLO?  RAH! GRRR!
```

55-SetFontHook dim(55,FSType,Offset,"VarName")

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 (**dim(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

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 will not see the last 4 chars.

56-Draw dim(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

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 dim(57)

Returns a string telling which BatLib version is installed. For example, if it returns "4.01.116.XX" you would break it up like this:

4 is the app and is a given. This should never change.

01 means it is the first major revision. 00 was released in the first 6 months or so of major development back in 2011 and had some buggy commands. After a year of hibernation, 01 is now released and has more commands and bugs were all fixed.

116 means there are commands 0 to 116

XX means that ReCode is disabled. Otherwise, it should reflect a rough estimate of the number of available commands.

58-ShiftScreen dim(58,NumShifts,Direction)

NumShifts is the number of pixels to shift the graph screen by
Direction is the direction to shift in:

1-Right
2-Left
4-Down
8-Up

If you want to shift in 2 directions, add the values together. So for example, to shift the screen left and down 7 pixels, do **dim(58,7,6**. Technically you can add 3 or all 4 directions, but that has almost no uses.

59-BaseX dim(59,[base1],"Number"[,base2[,Digits)

base1 is the base of *"Number"*. If ommitted, it defaults to base 16. Use 1~36
"Number" is the number to convert.
base2 is the base to convert the number to. Default is base 10. Use 2~36
Digits is the size of the string to return. For example, 4 returns the last 4 digits (in case a specific number of digits is required). If you use 0, this will return the full number. This is the default.

This is no lightweight base converter. This can handle numbers hundreds of digits long in bases 2 to 32 (though the input string can be base 1, the output cannot be). No error checking is done on the digits to make sure they are in the right base. For example, if you use B0 in base 10, it will simply read the second digit as a B (11 in decimal) so B0=110. An example of its use:

```
:dim(59,10,"255",16,2
```

If you specify a number of digits greater than the size of the number, it will have leading zeroes. For example:

```
:dim(59,10,"382",16,4
```

You will get the hexadecimal number "017E". Using an input of 0 for the digits will instead return the full string.

60-DelVarArc dim(60,"Var")

This deletes a variable, even if it is archived.

61-DrawRectVar dim(61,"Var",X,Width,Y,Height,Type)

This draws a rectangle to a variable instead of the graph screen.

Note: that this was designed to be used by variables with 768 bytes (like the picture vars made by PicHandle):

:dim(61,"GPic1",0,8,8,8,2

62-DrawToVar dim(62,"Var",Func[,Arg1...)

This uses the drawing functions in a variable instead of the graph screen.

*See the note for the previous command.

63-PixelTestpic dim(63,Pic#,Y,X)

Pic# is a value from 0 to 255. 0 is Pic1, 8 is Pic9, et cetera
Y is the Y-coordinate to test at. Use 0~63
X is the X-coordinate to test at. Use 0~95

This is used to perform the pixel test option directly on a picture. 0 is returned if the pixel is off, otherwise it returns 1. The picture **can** be in archive.

Example: I want to test the pixel (33,0) in Pic0:

:dim(63,9,33,0

64-CopyProg dim(64,"VarToCopy","NewVar")

VarToCopy is the name of the var to copy
NewVar is the name of the var to copy to

If the var already exists, it is overwritten. This will copy any variable from RAM or archive. It will not handle lists, reals, complex lists, complex numbers, or matrices properly (yet), but all others work.

65-RealToStr dim(65,Value)

Value is converted to a token as a string in Ans

For example, if you do **dim(65,69** Ans will be returned as "E" because the token for "E" is 69.
If you are looking for a command to convert a number to a string, see NumStr.

66-StrToReal dim(66,"String")

String is a string.

The first byte is converted to a number and stored in Ans. So if you did **dim(66,"E** you would get 69 in Ans.

67-DataString dim(67,{List}|"Str")

This command can take either a string or a list as input. If it is a string, each byte is converted to a list element in Ans. If it is a list, each element is converted to a byte and stored to a string in Ans.

For example, **dim(67,{65,4,67** would return "A→C" in Ans because 65="A" 4="→" 67="C

Likewise, if you do **dim(67,"AC** you will get {65,67} in Ans.

As a note, this works like the program DataString found on TICalc.org. For tips and tricks, go here.

68-MakeString dim(68,Size)

Size is how many bytes of data to create.

This creates a string in Ans of whatever size you designate.

69-SubMatrix dim(69,MatrixNumber,ColumnOffset,RowOffset,Width,Height)

MatrixNumber is a value from 0 to 255 telling which matrix to read from.
Alternatively, you can use a matrix directly (like [[0,1][10,11]])
ColumnOffset is a value from 0 to 98 telling which column to start reading at
RowOffset is the row to start reading at
Width is how many columns to read
Height is how many rows to read

*[A] is the 0th matrix and [J] is the 9th matrix
This allows you to extract a portion of a matrix **even if it is in archive.** For example, if I wanted to extract the upper left 3x3 corner of [I], then I would do:
dim(69,8,0,0,3,3

70-PlayNumber dim(70,Duration,Value)

Duration is the length of each note. 256 is fast, 65535 is really slow
Value is a number from 1 to 127

The lower the number, the higher the pitch

71-Logic dim(71,Logic,Number1[,Number2)

Logic is the logic operation to perform on the number or numbers:
- 0-performs AND logic on Number1 and Number2
- 2-performs XOR logic on Number1 and Number2
- 4-performs OR logic on Number1 and Number2
- 6-performs NOT logic on Number1

Number1 is a number
Number2 is a number. Do not use this argument with NOT logic.

*If you add 1 to the values, the 8 bit value is returned. Default is 16 bit.

72-MatrixList dim(72,[matrix]|Matr)

[matrix]|Matr is either the matrix number to convert (use 0 to 255) or it is the actual matrix.

This converts a matrix to a list, storing row by row. For example, if the matrix was:

```
[[0,1,2]
 [3,4,5]
 [6,7,8]]
```

This would return:

{0,1,2,3,4,5,6,7,8}

This works even if the matrix is archived.

73-Left dim(73,Str|"String",Size)

Str|"String" this is either a string or a number naming the string. Use 0~255 where 0=Str1 and 9=Str0

Size is how many bytes to read.

This reads the first bytes of a string. If *Size* is 3, it reads the first 3 bytes. For example:

```
:dim(73,"HELLO",3
```

This outputs "HEL" in Ans. This command works even if the string is archived.

74-Right dim(74,Str|"String",Size)

Str|"String" this is either a string or a number naming the string. Use 0~255 where 0=Str1 and 9=Str0

Size is how many bytes to read.

This reads the last bytes of a string. For example, to read the last 3 bytes in "HELLO" you can do:

```
:dim(74,"HELLO",3
```

The output will be "LLO" . This command works even if the string is archived.

75-Mid dim(75,Str|"String",Offset,Size)

Str|"String" this is either a string or a number naming the string. Use 0~255 where 0=Str1 and 9=Str0

Offset is where to start reading bytes. The first byte is 0

Size is how many bytes to read.

This is like the BASIC sub(command, except the offset starts at 0, this can read archived strings, and it reads it by bytes, not tokens.

76-Diag dim(76,[matrix]|Matr,Row,Col,Size)

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
Size is how many elements to read

For example, if [J] had the following matrix and you wanted to read the highlighted numbers:

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
```

You would need:
Matr: 9
Row: 0
Col: 1
Size: 3

:dim(76,9,0,1,3

Ans would be a list {1,10,19}

77-DiagI dim(77,[matrix]|Matr,Row,Col,Size)

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
Size is how many elements to read

This reads a diagonal in the opposite direction. Using the matrix in the previous example, **dim(77,9,0,5,3** would output {5,12,19}

78-SubCol dim(78,[matrix]|Matr,Row,Col,Size)

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
Size is how many elements to read

This reads a section of a column based on the offset and how many elements you want to read. The output is a list.

79-SubRow dim(79,[matrix]|Matr,Row,Col,Size)

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
Size is how many elements to read

This reads a section of a row based on the given offset amd number of elements to read. The output is a list.

80-ListToDiag dim(80,[matrix]|Matr,Row,Col,List|{List})

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
List|{List} is the list to copy the data from

This copies a list diagonally (upper-left to lower-right) to a matrix.

81-ListToDiagI dim(81,[matrix]|Matr,Row,Col,List|{List})

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
List|{List} is the list to copy the data from

This copies a list diagonally (upper-right to lower-left) to a matrix.

82-ListToCol dim(82,[matrix]|Matr,Row,Col,List|{List})

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
List|{List} is the list to copy the data from

This copies a list vertically to a matrix.

83-ListToRow dim(83,[matrix]|Matr,Row,Col,List|{List})

[matrix]|Matr is either the matrix number or the matrix itself
Row is the row offset into the matrix (0 is the start)
Col is the column offset into the matrix (0 is the start)
List|{List} is the list to copy the data from

This copies a list horizontally to a matrix.

84-ExecVarBASIC dim(84,"VarName")

"VarName" is the name of a variable with BASIC data

This executes a variable as a BASIC program. You should only use Programs or Appvars for this.
*As a note, if it appears to freeze on the homescreen, just press clear. This used to be a bug, but I cannot consistently replicate it.

85-GetProgName dim(85)

This returns the name of the current executimg program or if from the homescreen, the last run program. The name includes a prefix byte (Usually "E" or "F").
*Useful if a program references itself for data. Also useful if the end user changes the name of the program that references itself :D

86-Timer dim(86,Time)

Time is a value from 0 to 65535 that is the offset for the timer.

This is like the checkTmr(command except it only deals with 0 to 65535. Doing this:

```
:dim(86→A
:Repeat getKey
:Output(1,1,dim(86,A
:End
```

Would return the same exact result (for 65536 seconds) as:

```
:checkTmr(0→A
:Repeat getKey
:Output(1,1,checkTmr(A
:End
```

This command uses slightly more memory than checkTmr or startTmr, but it is about 3 times faster to execute, so it can be more accurate.

87-DrawText dim(87,Y,X,Str|Char#)

Y is the Y pixel coordinate to draw the character or string at
X is the X coordinate (0~23) to draw the character or string at
Str|Char# is either the char # to draw or the string

This routine uses a fixed font of 6 pixels tall and 4 pixels wide and draws it to the graph screen. There are 24 characters to a line and it does wrap the letters to the next line. If a letter would go off the bottom of the screen, it is instead drawn 54 pixels higher (the first row).
*The font is a 768 byte BatLib Font.
*This does not update or dirty the screen.

88-DrawDisp dim(88,Char#)

Str|Char# is either the char # to draw or the string

This draws a character or string at the last cursor position.

89-ASCIILength dim(89,"String")

"String" is a string.

This finds the number of chars in a string. For example, **dim(89,"sin(** would return 4 because the sin(token is 4 chars long. Likewise, **dim(89,"sin(ln(3** would return 8. This is useful if you need to know how much space on the screen a string will use.

90-DrawTokenStr dim(90[,Y,X],"String")

Y is the Y pixel coordinate to draw the character or string at
X is the X coordinate (0~23) to draw the character or string at
"String" is the string to draw

☼This draws the token string instead of ASCII.
☼Strings wrap to the next line or to the top of the screen if a line goes past the bottom.
☼If the X and Y arguments are omitted, the string is drawn at the last coordinate drawn to.

91-DelElements dim(91,Offset,NumElements,List|#|"Name")

Offset is the offset into the list (0 deletes starting with the first element)
NumElements is the number of elements to delete
List|#|"Name" is either a list, the name of a list, or the list number (0=L1).

For example, if you did **dim(91**,1,2,{0,1,2,3,4}), you would get {0,3,4}

92-InsElements dim(92,Offset,NumElements,List|#|"Name")

Offset is the offset into the list (0 inserts before the first element)
NumElements is the number of elements to insert
List|#|"Name" is either a list, the name of a list, or the list number (0=L1).

For example, If you did **dim(92**,2,2,{0,1,2,3,4}}, you would get {0,1,0,0,3,4}

93-InsList dim(93,ListName,Offset,List|#|"Name")

ListName is the list to insert. Use a list, name, or list number. This can be archived.
Offset is the offset into the list (0 inserts before the first element)
List|#|"Name" is either a list, the name of a list, or the list number (1=L2).

For example, to insert L2 into $_L$HELLO after the second element:

:dim(93,1,2,"$_L$HELLO

94-InsString dim(94,Offset,"VarName","String"|Str#)

Offset is the insertion point into the var. 0 will insert at the beginning
"VarName" is the name of a var. If the var is archived, ".ARCH" will be returned
"String"|Str# is either a string like "HELLO" or the number of a string (0=Str1,1=Str2, et cetera). This can be in archive.

95-SplitNibbles dim(95,"String")

"String" is a string of bytes that should be split up.

This is a tough command to explain. If we look at a string as hex, for example, "3031", this pretty much inserts a 0 before each hex digit, so you get "03000301" However, this is tokenized data, so really you would be doing

:dim(95,"01

And you would get the tokens represented by 03000301. The usefulness of this is when you are using tilemaps that use 16 tile or less. You can compress the bytes to nibbles then, when you are using the tilemap command, use this command to expand the nibbles again :) This saves half the bytes!

96-NibbleComp dim(96,"String")

"String" is a string of bytes that should be compressed.

This is the opposite of the previous command. This makes a string using every other half byte (nibble) of the input string. So if you have tilemap data that uses 16 tiles or less, you can use this command to compress it and then the previous command to decompress it :)

97-ReadNibble dim(97,"VarName",Offset)

"VarName" is the name of a var.
Offset is the nibble to read. 0 reads the first nibble.

Ans is returned with the nibble value. This will only be 0~15. For example, if prgmHI contained the bytes HI on the first line, the hex for "HI" is 4849, so:

dim(97,"EHI",0 would return 4
dim(97,"EHI",1 would return 8
dim(97,"EHI",2 would return 4
dim(97,"EHI",3 would return 9

98-WriteNibble dim(98,"VarName",Offset,Value)

"VarName" is the name of a var. If it is archived, ".ARCH" is returned.
Offset is nibble to overwrite. 0 overwrites the first nibble.
Value is a value to overwrite the nibble with. The value is divided by 16 and the remainder is used. Use 0~15, but higher values won't harm it :)

Ans contains the previous value of the nibble

99-DispGraphBuffer dim(99)

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

This is executed as command 99 or after the last command. So, if the last command (besides this) was 129, anything from 130 to 65535 would execute this command.

100-BatLibRAM dim(100)

Some apps and assembly programs interfere with important RAM areas in BatLib, so using this will reload the BatLibRAM. This will be useful when using multiple hooks (like Celtic 3 and BatLib).

101-Sub2DData   dim(101,Offset,"VarName",Width,Columns,Rows)

Offset is where the upper left part of the matrix data starts in the var
"VarName" is the name of the var that has the data
Width is how many columns there are in the matrix
Columns is how many columns to read
Rows is how many rows to read

If you have a string that is "0123456789", by defining the width as 3 you give the string two dimensions:

```
0 1 2
3 4 5
6 7 8
9
```

If you read 1 columns wide and 3 rows down, you will get "036" in Ans. So, as an example:

```
:"HELLO0123456789→Str0
:<<Code>>
:dim(101,5,".Str0",4,2,3
```

That will return "014589" because it is looking at the matrix:

0 1 2 3
4 5 6 7
8 9

This is useful for BatLib tilemap data in the same way that SubMatrix is useful for xLIB/Celtic 3 tilemap data :)

102-StringWidth   dim(102,"String")

"String" is a string

This returns how wide in pixels the string is. For example, **dim(102,"HELLO** would return 20 and **dim(102,"HELLO**** would return 32. This is useful for centering text and whatnot.

103-ReplaceByte   dim(103,SearchByte,ReplaceByte,"VarName")

SearchByte is the number value of a byte to search a var for
ReplaceByte is the numerical value of the byte to replace with
"VarName" is the name of a var

This will search the contents of a variable for the search byte and then for every instance, it replaces it using the replace byte. For example, to turn every new line token in prgmRAH into a colon, you can do:

```
:dim(103,63,62,"ERAH
```

104-TokensToASCII `dim(104,"String")`

"String" is a string of tokens to convert to ASCII

This will convert the string to its ASCII values. For example:

```
:dim(104,"sin(
```

That will return a string of 4 bytes. When you use DispASCII on the result, it will display 'sin('. This is particularly useful when you have a lot of lowercase letters because a lowercase/accented/greek letter uses only 1 byte in ASCII instead of 2.

105-PxlLine `dim(105,X1,Y1,X2,Y2,Type[,{Pattern})`

X1,Y1,X2,Y2 are all pixel coordinates and may go off the screen. However, they wrap back around after 255 (so 256=0, for example).

Type is how to draw the line:

 0=Pxl-Off
 1=Pxl-On
 2=Pxl-Change
+4=Merth Pattern
+8=Ray Pattern

{Pattern} is the pattern to use when drawing with the Merth or Ray pattern options.

This routine very quickly draws a line at pixel coordinates. Here is a description and example with the pattern options.

Merth Pattern will draw a number of pixels and then ignore a number of pixels. So for example, {2,1} will draw 2 pixels, ignore one pixel, and then continue.

Ray Pattern will draw a number of rows and then ignore a number of rows. This was designed with the purpose of raycasting in mind, but can be useful in other areas, too.

For both pattern types, patterns can be more complicated. For example, {2,1,3,1} will draw 2, ignore 1, draw 3, ignore 1, and then repeat. If you have an odd number of arguments, you can get an interesting pattern or optimisation. For example, {2} will draw 2, ignore 2, draw 2, ignore 2, et cetera. {1,2,1} will draw 1, ignore 2, draw 1, but then it continues with ignore 1, draw 2, ignore 1.

An example of using this:

```
dim(105,X,Y,R,S,6,{2},99,105,X,Y,R,S,6,{2
```

106-PxlCircle `dim(106,Y,X,Radius,Type[,{Pattern})`

Y,X are pixel coordinates and may go off the screen. The wrap around after 255

Radius is in pixels and can be 0 to 127

Type is how to draw the circle:

 0=Pxl-Off
 1=Pxl-On
 2=Pxl-Change
+4=Merth Pattern

{Pattern} is the pattern to use when drawing with the Merth pattern options.

This routine very quickly draws a circle at pixel coordinates. The pattern option follows the same rules as PxlLine.

107-PxlLineVar dim(107,"VarName",X1,Y1,X2,Y2,Type[,{Pattern})

"VarName" is the variable to draw to.
X1,Y1,X2,Y2 are all pixel coordinates and may go off the screen. However, they wrap back around after 255 (so 256=0, for example).
Type is how to draw the line:

 0=Pxl-Off
 1=Pxl-On
 2=Pxl-Change
+4=Merth Pattern
+8=Ray Pattern

{Pattern} is the pattern to use when drawing with the Merth or Ray pattern options.

108-PxlCircleVar dim(108,"Varname",Y,X,Radius,Type[,{Pattern})

"VarName" is the variable to draw to.
Y,X are pixel coordinates and may go off the screen. The wrap around after 255
Radius is in pixels and can be 0 to 127
Type is how to draw the circle:

 0=Pxl-Off
 1=Pxl-On
 2=Pxl-Change
+4=Merth Pattern

{Pattern} is the pattern to use when drawing with the Merth pattern options.

This routine very quickly draws a circle at pixel coordinates. The pattern option follows the same rules as PxlLine.

109-PopUp dim(109,x,y,w,"Header","Item1","Item2","Item3",...)

y,x are pixel coordinates
w is the width in pixels of the popup menu

The rest of the arguments are strings. The first non string encountered will end the command. This uses the 4x6 fixed font of BatLib, so finding a good width should not be difficult. This returns the item number returned, starting at zero. Use the arrows to highlight the item and Enter to select it. The menu is drawn to the graph screen and will have its height adjusted accordingly. It can only hold as many items as will fit on the screen.

110-NumStr dim(110,Number)

This converts real number values to a string. It currently does not work for imaginary numbers.

111-ListStr dim(111,{list})

This converts a list to a string. This *does* work with imaginary numbers. Output is the list in a string. For example:

```
:dim(111,{.3+i,-.2i+3.1,4,i})
```

This will output "{.3+i,3.1-.2i,4,i

112-CompressText dim(112,Codec,"String"

Codec is the codec to use for compression. These are built in to BatLib:

0 uses tokens ETAOINSHRDLCUMFGYPBVKWXJQZ,?!.0123456789/-:'

1 uses all those tokens plus lowercase and +,*, negative, and a newline.

"String" is either a direct string or the number of the string.

The compression ratio from this is variable. Since I add in two leading bytes before the actual data, the best compression will be 50% plus 2 bytes, the worst will be a gain of two bytes. For an average body of text, I believe the compression ratio is about 60% (it has been a few years since I computed it).

113-DecompressText dim(113,Codec,"CompressedString"

Codec is the codec to use for compression. These are built in to BatLib:

0 uses tokens ETAOINSHRDLCUMFGYPBVKWXJQZ,?!.0123456789/-:'

1 uses all those tokens plus lowercase and +,*, negative, and a newline.

"String" is either a direct string or the number of the string. This must be an output from the previous command.

This takes compressed text from the previous command and decompresses it.

114-GetCodec dim(114,"String"

"String" is either a direct string or the number of the string.

The next two commands let you use custom codecs. This command will return an optimal codec for whatever data is in the string. If the data uses, for example, up to 46 different bytes, then 14 of those bytes will be compressed to half bytes. If it uses 61, then 13 of those will be half bytes. Here is a list:

```
16   16
31   15
46   14    ;An example is the bult in codec 0 for the previous command
61   13
76   12    ;The built in codec 1 is this type
91   11
106  10
131   9
146   8
161   7
176   6
191   5
206   4
221   3
236   2
241   1    ;Just hope the most common byte is really common
256   0    ;still scrambles the text, possibly
```

115-CompressData dim(115,"Codec","String"

Codec is the codec to use for compression. This is a custom codec. If you reference a string by name (using a number), the string can be in archive.

"String" is either a direct string or the number of the string.

One way to use this is for programs acting as text data storage. Store the codec to an appvar and use the same codec on several such files to compress/decompress them.

116-DecompressData dim(116,"Codec","String"

Codec is the codec to use for compression. This is a custom codec. If you reference a string by name (using a number), the string can be in archive.

"String" is either a direct string or the number of the string. This is an output from the compression routine.

117-SearchReplace dim(117,"VarName","SearchString","ReplaceString")

VarName is the name of a var in RAM
SearchString is the string (or a string number) to search for. Must be in RAM.
ReplaceString is the string (or a string number) to replace all matches with. This can be archived.

Every occurance of the search string in the variable will be replaced by ReplaceString. As a note, this is pretty slow for large pieces of data with many replacements. It took about 30 seconds to search and replace every instance of the letter L with "HELLO" in an 8000 byte file filled with the word HELLO. For smaller

118-GetVarName dim(118,VarType,VarNumber)

VarType is a value as defined here:

- 00-Programs (This includes protected programs.)
- 01-Appvars
- 02-TempPrograms
- 03-Strings
- 04-Pictures
- 05-GDB
- 06-Groups
- 07-Matrices
- 08-Equations
- 09-Real vars
- 10-Complex Numbers
- 11-Lists

VarNumber is the *n*th var name to return, starting at zero. Names are returned in alphabetical order.

For example, if you want the third item that would appear in the memory menu for programs, **dim(**118,0,2. Names are returned with a leading type byte.

119-Lowercase dim(119.x)

The inputs here are after the decimal place:

0 is lowercase enabled
1 is lowercase disabled
2 is lowercase toggled

For example, dim(119) enables lowercase, dim(119.2) toggles lowercase.

120-TextMode dim(120.x)

The inputs here are after the decimal place:

0 is text inverse enabled
1 is text inverse disabled
2 is text inverse toggled
3 is "erase below" enabled
4 is "erase below" disabled
5 is "erase below" toggled

* "erase below" is used when using the Text(command. If this is enabled, the row below any drawn text is erased. For advanced text graphics you should disable this (For example, vertical text sprites and horizontal text sprites).

121-HexSpriteVar dim(121,"VarName",[option],"Hex",Height,X,Y,Logic)

VarName is the name of the variable to draw to. For the rest of the arguments, see HexSprite.

122-TokenSpriteVar dim(122,"VarName",[option],"Hex",Height,X,Y,Logic)

VarName is the name of the variable to draw to. For the rest of the arguments, see TokenSprite.

123-DBDelLine dim(123,LineByte,LineNumber,"VarName")

LineByte is the value to read as a newline. 63 is the BASIC newline.
LineNumber is the line number to delete.
VarName is the name of the var to delete the line from.

This deletes a line of data from a var. If you use a *LineByte* value of 63, this can be used to delete a line from a BASIC program variable.

Info

Prefix Bytes

00=Real	log(	
01=List	A	
02=Matrix	B	
03=EQU	C	
04=String	D	
05=Program	E	
06=ProtProg	[	F
07=Picture	]	G
08=GDB	{	H
09=Unknown	}	I
10=Unknown Equ	J	
11=New EQU	K	
12=Complex	$^{-1}$	L
13=Complex List	^{2}	M
14=Undefined	N	
15=Window	^{3}	O
16=ZSto	(	P
17=Table Range	)	Q
18=LCD	2	R
19=BackUp	3	S
20=App	4	T
21=Appvar	5	U
22=TempProg	6	V
23=Group	7	W

	Format should not be used (unless asked for).
	Do not use.
	Symbol Var. Compatible with each other.
	Named Var. Compatible with each other.

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 **dim(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

This is thrown if an argument is too large. Used in the BaseX command.

.ARCH

This is thrown if a var to be edited is archived

.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

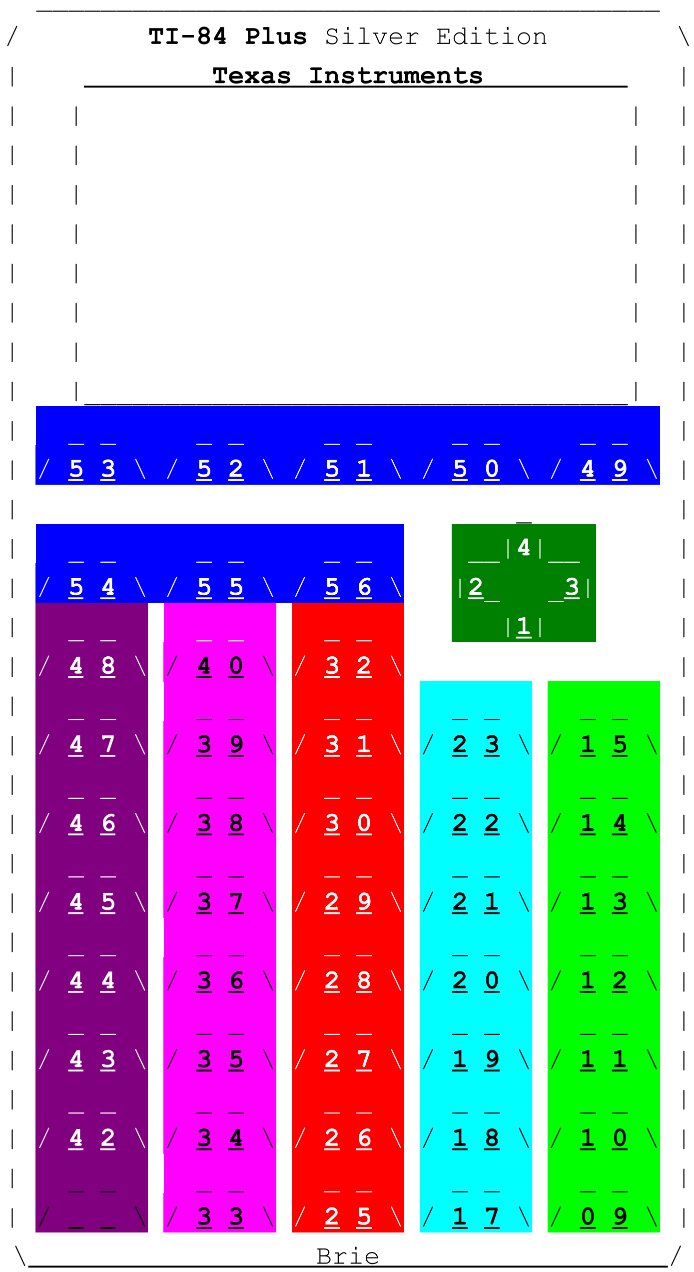
This is output if the width argument is 0 or too big.

.HEIGHT

This is output if the height argument is 0 or too big.

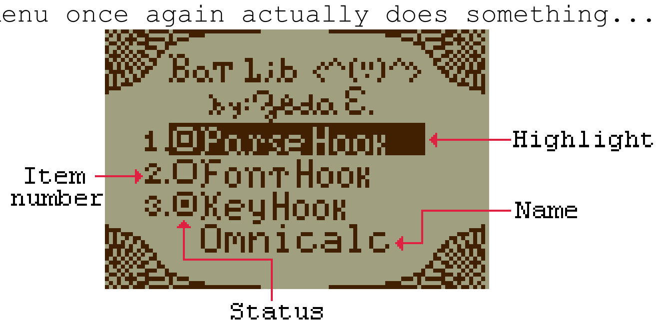
Key Codes

You can use this as a guide to the key values ouput by Get2Key. For example, Clear=15. Youwill need to do some math to figure out multiple key presses.



Main Menu

The main menu once again actually does something...



Item number- Press this number to highlight this item

Highlight-This tells which item is currently selected

Status-This tells whether there is an active hook installed.

Name-This is the name of the App that had a hook installed before BatLib was installed. In the picture, Omnicalc had a Parser hook installed.

Contols:

-Press clear to exit

-Use numbers 1,2, or 3 to highlight an item

-Use up/down to cycle throught menu items super fast >:D

Thanks

DJ_O

Thank you DJ_O for hosting this as a Major Community Project on Omnimaga as well as providing a few neat ideas (like pixel testing pictures).

Scout/Ephan (David Gomes)

For providing so much feedback and examples with regular BatLib BASIC and ReCode

ztrumpet

For providing feedback and making that example program way way back when :D

mrmprog

Honestly, you have requested several cool codes that were relatively easy to create and extremely useful.

BlakPilar

Thanks for providing feedback way way back when this was a program with almost 30 functions :P Also, thanks for the waffle (>^_^)>#

Merth

Thanks for the ideas! It only took about a year and a half to finally add the line patterns.

Sorunome

For your excitement and for giving me a confidence boost :) I was very happy to find that you liked BatLib :) I was getting almost no feedback for a while, so thanks :)

BuckEye

Wabbitemu has been extremely useful for debugging, so thanks much from all of us assembly programmers!

And thanks to all the others I haven't mentioned from United-TI, Cemetech, TIBD, and Omnimaga :)

Dear Programmer,

I have currently been programming for the TI-83+/84+/SE calcs for about 5 years and I have enjoyed this project greatly. As a BASIC programmer and an assembly programmer, it is my hope that this tool will prove useful for years to come and for many people. There are many very useful tools available including Celtic III, xLIB, Omnicalc, and DoorsCS7, all of which I have made use of in the past. I would not have developed my techniques or knowledge of coding without these programs. They exposed me to many concepts ranging from data storage, tilemapping, spriting, bit logic, and other forms of mathematics. I have tried to incorporate many of the same ideas in BatLib, as well as many more, and that is why BatLib is the largest compilation of commands of any library of its kind. Have fun and happy coding!

Zeda E.