Project.....BASIC ReCode
Program.....BatLib
Author.....Zeda Elnara (ThunderBolt)
E-mail.....xedaelnara@gmail.com
Size.....
Language.....English
Programming.....Assembly
Version.....2.00.50
Last Update.....15 March 2011

BASIC ReCode

BASIC ReCode was once a program of its own, but when I turned SpriteLib into an App (now known as BatLib), I decided to merge the two projects. For a while, I stopped working on BASIC ReCode, but after nine months of mostly dormancy, I have picked it back up.

For those who do not know, BASIC ReCode (otherwise known as ReCode) is an interpreted programming language that can be inserted into a BASIC program. Like any language, grammar is very important. Since ReCode is an interpreted language that uses BASIC tokens, it might be easy to confuse cognates, especially when you are going from one language to the next. For example, a few years ago in French class, one student was trying to say "I am very excited" and since he didn't know the word for "excited" he assumed it was a cognate and so said "Je suis tres exciter!"... let's just say that it *does* mean excited, just not in a G rated way. Likewise, in ReCode, do not assume certain commands work the same as they do in BASIC. For example, while Line(draws a line from one point to another in BASIC, in ReCode it draws a rectangle from one point to another.

How to use

To begin a ReCode block, use the **dim(40** command of BatLib. To end a ReCode block, simply use the "Stop" token. The BASIC program will then pick back up.

Commands

Stop

This ends a ReCode block

→

This stores Ans to a ReCode var. For example, **3→A**. Variables can be used to replace numbers, so **Pause B** is valid. These do not change the BASIC variables.

!

The exclamation point is used to distinguish between a ReCode var and a BASIC var. For example, to store to the BASIC real var B:

```
:3→!B
```

And then to read the BASIC real var B and store it to the ReCode var B:

```
:!B→B
```

ReCode vars are faster to use, by the way.

'

This is used to access more vars. For example, A' and A are two different vars.

"Strx

This is used to reference an OS string, such as Str1. Ans must be the number of bytes to read. If Ans is 0, the whole string is used, instead. For example:

```
:"EF4045C9→Str3
:dim(40
:0
:AsmPrgm"Str3
:Stop
```

Also, you can do some math on the string to get an offset into it. For example, to read 3 bytes into a string, you would do **"Str0+3**

Ans

This swaps the primary and secondary Ans. To get the remainder of a division, for example:

```
:33/4→Q
:Ans→R
```

Line(X,Width,Y,Height,Type

This draws a rectangle on the graph screen.

The arguments follow the same syntax as the **dim(29** command of BatLib. For example, to draw a 5x5 inverted rectangle at (0,0):

```
:Line(0,5,0,5,2
```

Pause xx

This pauses for approximately xx/100 seconds. for example, Pause 333 would make a delay of 3.33 seconds. 0 is interpreted as 65536.

Pause While xx

If you put a "While " after the pause instead of a number, you can pause until the statement after "While " is false. For that reason, you need to be careful about what you input. An example of waiting until clear is pressed:

```
:Pause While getKey≠15
```

You can quite literally read it as "Pause While getKey is not equal to 15"

Disp y,x,String

This displays a string on the homescreen at the coordinates indicated.

Y is a value from 0 to 7

X is a value from 0 to 15

String is a string of ASCII data without quotes

As an example, **Disp 0,0,HELLO** will display HELLO at the upper left corner.

AsmPrgm

This lets you execute an assembly opcode from the address 86ECh. For example, EF4045C9 is the opcode to clear the LCD. Using AsmPrgm, you can do **AsmPrgmEF4045C9** to execute the opcode.

Text(y,x,TokenString

This displays a string of tokens (not ASCII) on the graph screen at the cursor position. This uses the fixed width 4x6 BatLib font.

Y is a value from 0 to 58

X is a value from 0 to 23

TokenString is a string of tokens to display.

As an example, **Text(0,0,cos(ln(Hello))** will display the string "cos(ln(Hello))" at the upper left corner of the graph screen.

This routine will wrap text to the next line or if it goes off the bottom of the screen it wraps to the top.

*If you omit X and Y but you put a negative sign at the start of the screen, the text is drawn to the last cursor position.

getKey

This returns a value in Ans that reflects the current key press. This will return an individual value for up to two key presses. For example:

```
:getKey→K
```

IS>(x

If Ans is not 0, this will jump x lines forward

DS>(x

If Ans is not 0 this will execute the previous x lines again.

While xx

This works like the BASIC While command. As long as xx results in a non-zero value, anything between While... End is executed. For example, this will keep inverting the screen until Clear is pressed:

```
:While getKey≠15
:Fill(2
:DispGraph
:End
```

If xx

If xx is 0, the block is skipped. Here are a few examples:

```
:3→A
:If A=0
:Disp 3,3,THIS GETS SKIPPED BECAUSE A=0 IS NOT TRUE
:Disp 4,3,SO THE PARSER GOES TO THIS LINE AND EXECUTES IT
```

And another example:

```
:3→A
:If A=3
:Then
:Disp A,A,THIS GETS EXECUTED
:Disp A+1,A,AS DOES THIS
:End
:Pause 33
```

DispGraph

This displays the graph screen. Useful if you have just displayed text or drawn on the graph screen.

dim(

This executes a BatLib command. Be sure to include all syntaxes. Stringing commands will not work and only simple commands should be used as ReCode does not pass non-number values well.

Vertical x,Method

This draws a vertical line on the graph screen.
X is the X pixel coordinate to draw the line at.
Method is how to draw the line:

0 draws a white line
1 draws a black line
2 draws an inverted line

Horizontal y,Method

This draws a horizontal line across the graph screen.
Y is the Y pixel coordinate to draw the line at.
Method is how to draw the line:

0 draws a white line
1 draws a black line
2 draws an inverted line

Shade(xx

This sets the contrast to **xx** (use values 0 to 39). Normal is bout 24.

Fill(Method[,Value]

This will "fill" the graph screen using some method:

0 clears the graph
1 turns the graph black
2 inverts the graph
3 Draws vertical lines every other pixel (starting at pixel 1)
4 Draws vertical lines every other pixel (starting at pixel 0)
5 XORs the screen with vertical lines every other pixel (start=pixel 1)
6 XORs the screen with vertical lines every other pixel (start=pixel 0)
7 turns off pixels defined by **Value**. For example, **Fill(7,1** will turn off every 8th pixel.
8 turns on pixels defined by **Value**. For example, **Fill(8,1** will turn on every 8th pixel.
9 XORs pixels defined by **Value**. For example, **Fill(9,1** will XOR every 8th pixel.
10 Overwrites pixels defined by **Value**. For example, **Fill(10,1** will turn off 7 pixels and the 8th pixel is turned on.

*For commands 7 through 10, you use the **Value** argument. You will need to play with it, but if you understand binary and bytes, **Value** is an 8-bit value...

Return

This is a little complicated to explain, but it is a simple command. It has two syntaxes. For example:

```
Return→A
  OR
ReturnA
```

The first one stores the current address to A and the second one jumps to wherever A points to. As an example of its use:

```
:64→B
:Return→H        ;This stores the address of the next line to H
:dim(58,1,4      ;This shifts the graph screen down 1 pixel
:DispGraph       ;This shows the graph screen
:B-1→B
:If B            ;This tests if B is 0
:ReturnH         ;This points the parser back to "dim(58,1,4"
```

This is faster than a Goto or DS>(or IS>(, but it isn't as versatile as Goto and uses a few bytes more code than DS>(and IS>(

rand

This returns a random value from 0 to 65535. As a tip, to get randInt(0,x):

```
:rand*rand/x
:/x
:Ans
```

Full

This puts the calc at 15MHz mode (if possible). If you have a number following it, like **Full2**, you can perform these functions:

```
Full0 puts the calc at 6MHz mode
Full1 puts the calc at 15MHz mode
Any other values toggle the mode
```

RecallPic xx

This will copy a picture to the graph screen. xx is a number from 0 to 255 and the picture can be in archive.

```
0=Pic1
1=Pic2
...
8=Pic9
9=Pic0
```

And the rest are hacked pictures.

StorePic xx

This will store the graph screen to a picture. The picture is automatically overwritten, even if it is in archive. Also, this captures the bottom row of pixels.

Pt-On(HexData,Height,X,Y,Method

This is used to display a sprite on the graph screen.

HexData is a string of hexadecimal sprite data.

Height is the height of the sprite. Based on this, the width is automatically calculated.

X is the x-coordinate to draw the sprite at. Use 0 to 11

Y is the y pixel coordinate to draw the sprite at

Method is some form of logic:

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

Math Functions

These include:

+ this adds the following number to Ans.

- this subtracts the following number from Ans

***** this multiplies the following number by Ans

/ this divides Ans by the following number. Only the integer part is returned.

(-) the negative symbol does the operation 65536-the following number

Logic

and performs AND logic on two numbers

or performs OR logic on two numbers

xor performs XOR logic on two numbers

not(performs NOT logic on a numbers

= returns 1 if both numbers are equal, 0 otherwise

≠ returns 1 if both numbers are not equal, 0 otherwise

> returns 1 if the Ans is greater than the number, 0 otherwise

≥ returns 1 if the Ans is greater than or equal the number, 0 otherwise

< returns 1 if the Ans is less than the number, 0 otherwise

≤ returns 1 if the Ans is less than or equal to the number, 0 otherwise

Ans is the last computed value, so in other words, this:

```
:3+2
```

Will store 5 in Ans as well as this:

```
:3
:+2
```

Pixel Commands

Pxl-On(will turn a pixel on at (y,x). Returns the previous state of the pixel.

Pxl-Off(will turn a pixel off at (y,x). Returns the previous state of the pixel.

Pxl-Change(will toggle a pixel at (y,x). Returns the previous state of the pixel.

pxl-Test(will return the state of the pixel. 0=off, 1=on.

Tips&Tricks

For the **Disp** command, if you supply Y and X arguments that are too large, the mod value is taken (mod_8 and mod_{16} respectively). For example, if you supply a Y value of 8, it will be read as 0. If you supply a value of 10, it will be read as 2.

Advanced Notes

To understand how the parser works, we will look at some examples:

```
:337
```

When the parser comes across a number, it converts the 16-bit value and stores it to 'Ans'. After reading this line, Ans is equal to 337.

```
:Pause 337
```

When the parser reads the token for 'Pause', it then parses the 337 and uses that as the argument. Since 337 was the parsed number, 337 is in Ans.

```
:Pause Pause 337
```

In this case, the parser reads 'Pause' and then parses the next argument to find Ans. It then reads Pause again and so it parses the next argument as 337. After executing the second Pause, Ans is 337 and the first Pause executes.

This is a very important concept to optimizing ReCode. If you use the Disp trick, for example, you can do **Disp 2,Pause 33,HELLO** and it will Pause for .33 seconds and then display "HELLO" at the cursor position (2,1). However, if you do **Pause Disp 2,33,HELLO** it will display "HELLO" at (2,1) and then Pause for .33 seconds. This also explains how math works in ReCode. Math is interpreted from right to left, so:

3+2*6-12/**4+2**
3+2*6-12/**6**
3+2*6-**2**
3+2***4**
3+**8**
11