Project.....Grammer Documentation
Program.....Grammer (APP)
Author.....Zeda Elnara (ThunderBolt)
E-mail.....xedaelnara@gmail.com
Size.....1-Page App
Language.....English
Programming.....Assembly
Version.....2.25.03.12
Last Update.....25 March 2012

****Updates**: For THIS version
I am going to have to rewrite this whole document because it is a mess with inserting new commands everywhere. Anyways, here are useful additions since the last update:
-The user can now use lowercase letters instead of, say, A'. This is the same size, but looks cleaner.
-You can now store θ' and Ans at the same time by defining both vars after →. For example, R*P→aA will store the upper 16-bits of the multiplication in **a** and the lower 16-bits in **A**.
-**Output(2** is the Text(mode for drawing the fixed font to pixel coordinates
-**Text(** without arguments will return the coordinates as Ans=Y, θ'=X
-For the coordinates in a Text(command, you can do relative placement. For example, if you want text at 2 pixels below the last drawn text:

```
:Text(+2,,"2
```

The +2 draws it down 2 pixel from the last location and the empty argument for X means it uses the previous X coordinate. Useful for subscripts and super scripts
-You can set the Text(coordinates by using two arguments for Y and X. For example:

```
:Text(56,0
```

Now to get to rewriting this mess...

Grammer Tutorial

Introduction

Grammer is a programming language designed to be powerful, fast, efficient, and safe. However, with all of that said, though it isn't as tough to learn as Assembly, it will probably be a little more difficult than BASIC. One similarity to BASIC is that Grammer is interpreted. This makes it slower than assembly, but it is smaller and more safe.

This tutorial will be designed to show Grammer users all the cool features and hacks that Grammer provides, so have fun!

Terminology

There are ideas and terms in Grammer that would almost never appear in BASIC. So the first thing that needs to be covered is terminology:

Pointer

Pointers are used often in Grammer as a means of accessing data. This has a few positive side effects as well as a few negative ones. In easy(ish) terms, all data is located in memory. The "location" is given as a number and on the calulator, this number is from 0 to 65535. A pointer, then, points to the data location.

So where are pointers used in Grammer? Say, for example, you want to reference a string. In Grammer, you would do "HELLO→A. The location of the string "HELLO" is stored to A.

Sprites

A sprite is just an image. In Grammer, sprites are a multiple of 8 pixels wide up to 96 and up to 64 pixels tall.

Strings

Strings can be any form of data. They can be text, sprite data, label names, even code.

Pointer Vars

Pointer Vars are the letters A through Z and Θ as well as their primes. For example, S and S' are two different pointer vars. These are used to store pointers or values. Pointer vars are 16 bits, so only integers from 0 to 65535 are stored to them.

Arguments

Whenever you see a [in an argument, it means anything after is optional. For example, **Horizontal X**, **Horizontal X,2**, and **Horizontal X,2,H'** are all valid forms of the **Horizontal** command.

Subject 1-Numbers

In Grammer, values are integers from 0 to 65535. This is one of the key differences between Grammer and TI-BASIC. While this may make some things difficult, this has a few uses and effects not found in BASIC.

So first thing up, what happens when you go beyond 65535? The numbers simply loop back to 0! So what happens when you go below 0? You loop back starting at 65535! This is called modular math, so if you understand this section, you will actually be understanding an important concept in number theory. The uses of this will come later...

Subject 2-Math

In Grammer, math does not follow Order Of Operations and has some limiting factors (like only working with 16-bit integers). However, there are exploits that can be useful for this. Anyway, math is done from right to left, so as an example, we will look at 3*4+6/4-2:

3*4+6/4-2
3*4+6/**2**
3*4+**3**
3***7**
21

There are a bunch of math commands at your disposal, too, not just basic math. The tricks, however, come from θ'. There are several operations that modify θ' as a way to return additional information. For example, when adding, if the result exceeds 65535, 1 is stored to θ', otherwise it becomes 0:

+: If the result is greater than 65535, θ' is 1, otherwise it is 0.
-: If the result is less than 0, θ' is 65535, otherwise it is 0.
*: θ' is the upper 16 bits allowing for a 32-bit result.
/: θ' is the remainder
^{2}: Same as multiplication

See the Math Functions section for more info on other operations.

If you wanted to do **If 24=((A*8)+(B*12))**, you would do this:

```
:A*8
:+B*12
:If =24
```

However, if you want to keep it all on one line (because anything in an If or While statement does not modify Ans), you can use a space instead of a newline:

```
:If A*8 +B*12 =24
```

Subject 3-Logic

Logical operators are used to compare values. If the logic is true, the result is 1 and if it is false, the result is 0. As an example, 3>4 is false because 3 is not greater than 4, so this returns a 0. These are useful for conditional statements used in **If** , **While** , and **Repeat** .

Subject 4-If blocks

If blocks are often used because they are fairly useful. For example, if you want to do something if a **condition** is true, you will usually want to use an If statement. For example, if you want to increment A if getKey is 9 (if [ENTER] is pressed):

```
:If getKey=9
:A+1→A
```

As in TI-BASIC, you can also use an If...Then...End statement to handle multiple lines of code. For example:

```
:If A=1
:Then
```

```
:-C→C
:-D→D
:End
```

Although usually you can get away with putting it all on one line like this to save a few bytes:

```
:If A=1
:-C→C -D→D       ;note that there is a space instead of a newline
```

Subject 5-Loops

Most games or math algorithms will use loops and Grammer offers three loops that act the same way as in BASIC:

Repeat

This will repeat the code between Repeat...End until the statement is true. For example, to increment A until A=999:

```
:Repeat A=999
:A+1→A
:End
```

Note that this will execute the code first and then check the conditional.

While

This is a little different from Repeat in that it executes the code in the loop **while** the condition is true. If the condition is false before it enters the loop, the code inside never gets executed. So for example, to loop so long as A<999:

```
:While A<999
:A+1→A
:End
```

For(

This will increment a variable from a starting point to an ending point. For example, this will cycle starting with A=3, incrementing A by 1 every cycle until it reaches 9:

```
:For(A,3,9
:B-1→B
:End
```

Subject 6-Labels

Labels are a neat feature in Grammer that are very much unlike BASIC. In BASIC, you start a label name with Lbl and it can be up to 2 chars long and must be numbers or letters (or a combination).

In Grammer, labels start with a "." and can contain any character except a newline. There is also no size limit, but remember that a label of fifty chars will take longer to locate than one of 5 characters. Here is a label example:

```
:.HeLLO
```

We use **Lbl** to get the location of the label which is used by Goto. This also lets you store data to a label and get a pointer to it. So as an example:

```
:Lbl "HeLLO→Z
```

```
:<<Code>>
:.HeLLO
```

Subject 7-Code Flow

Oftentimes, linear code flow just won't work. Sometimes you need to jump around a bit and there are a few commands for this:

Goto

This is used to jump to wherever a pointer is pointing and execute code from there. Here are two ways to jump to a label:

```
:Lbl "HeLLO→A
:Goto A
```

Or:

```
:Goto Lbl "HeLLO
```

Return

This returns a pointer to the line following it. I should note that this was created before Repeat and While. So an example of a loop until [CLEAR] is pressed:

```
:Return→Z
:If 15=getKey
:Goto Z
```

prgm

This will execute a subroutine and then come back. See Subject 10 for info on subroutines. If the subroutine is located at eh label named HeLLO:

```
:Lbl "HeLLO→A
:prgmA
```

Subject 8-Strings

Strings are accessed through pointers. The quote token returns the pointer to the string. So for example, here are two ways to display text:

```
:"HELLO WORLD→A
:Text(0,0,A
```

Or:

```
:Text(0,0,"HELLO WORLD
```

NEW in v2.21.01.12

OS Strings can be stored to using →, as well, and these will hold a string.

Subject 9-Sprites

Using sprites is a pretty advanced technique, so don't expect to understand everything here.

Sprites are pretty much mini pictures. They are a quick way to get detailed objects that move around making them a powerful tool for graphics. In Grammer, the main sprite commands are **Pt-On(** and **Pt-Off(** and both have differences and advantages over the other.

Sprite Data

Sprite data is in the form of bytes or hexadecimal and you will want to understand binary to hex conversions for this. For example, to draw an 8x8 circle, all the pixels on should be a 1 in binary and each row needs to be converted to hex:

```
0 0 1 1 1 1 0 0  =3C
```

0 1 0 0 0 0 1 0 =42
1 0 0 0 0 0 0 1 =81
1 0 0 0 0 0 0 1 =81
1 0 0 0 0 0 0 1 =81
1 0 0 0 0 0 0 1 =81
0 1 0 0 0 0 1 0 =42
0 0 1 1 1 1 0 0 =3C

So the data would be 3C4281818181423C in hexadecimal.

Sprite Logic

There are 5 forms of sprite logic offered by Grammer, currrently. These tell how the sprite should be drawn and can all be useful in different situations.

Overwrite:

For an 8x8 sprite, this will erase the 8x8 area on the screen and draw the sprite.

AND:

This leaves the pixel on the screen on if and only if the sprites pixel is on and the pixel on the screen is on.

OR:

This will turn a pixel on on the screen if it is already on or the sprite has the pixel on. This never erases pixels.

XOR:

If the sprites pixel is the ame state as the one on the screen, the pixel is turned off, otherwise, it is turned on. For example, if both pixels are on, the result is off.

Erase:

Any pixels that are on in the sprite are erased on screen. The pixels that are off in the sprite do not affect the pixels on the graph buffer.

Pt-On(

This is used to display sprites as tiles. This means it displays the sprite very quickly, but you can only draw to every 8 pixels.

Pt-Off(

This is a slightly slower sprite routine, but it allows you to draw the sprite to pixel coordinates.

Subject 10-Sub Routines

Subroutines are very usefl for code organisation and saving memory. If you have a piece of code used multiple times throughout the program, you will probably benefit from this. A subroutine must end with an **End** token and is called with **prgm** instead of Goto. So for example, this is a way to call a subroutine:

```
:Lbl "SUB1→A
:prgmA
:prgmA
:Goto Lbl "Stop
:.SUB1
:Circle(rand/1024,rand/1024,rand/1024,3
:DispGraph
:End
:.Stop
```

```
:Stop
```

Subject 11-Text(

Text(has a lot of neat features in Grammer. First, Grammer uses its own font and it works like the homescreen in that it draws in 24 columns (the homescreen draws in 16 columns). The font is 4x6 and is fixed width. However, here are the neat aspects and how to use the command.

-To draw text, simply do this:

```
:Text(Y,X,"Text      ;"Text" can be a pointer to a string
```

-To draw a number, use the ' symbol:

```
:Text('Y,X,99
```

-To draw a number in a specific base (use 2 to 32), add another argument:

```
:Text('Y,X,99,16     ;drawn in hexadecimal (so it shows 63)
```

-To draw at the end of the last text drawn, use a degree symbol to replace coordinates:

```
:Text(°"Text
```

-Likewise, you can do this with numbers:

```
:Text('°99,2         ;draws 99 in binary
```

-*Another* feature is using /Text(or Text(ʳ for typewriter text mode (that is the superscript r found at [2nd][APPS]). This will display characters with a delay. The delay is chosen with Fix Text(. This will even display the individual letters in a token as if it is being typed. Here is an example:

```
:/Text(Y,X,"HELLO
```

-And you can use numbers and other operators, too!

-Another thing that is nice is that text wraps to the next line and if it goes off the bottom, it wraps to the top.

-To display a char by number, the arguments are:

```
:Text(Y,X,'#
```

The ' operator tells it to draw char(#).

-To draw text as an ASCII string, use ° before the string. For example:

```
:Text(Y,X,°"HIrandM(WORLD
```

That will display the text "HI WORLD" because randM(corresponds to the space char in the ASCII set.

Subject 12-Interrupts

An interrupt is similar to a subroutine, but there are a few things to keep in mind. You might want to back up your program before experimenting:

-The code cannot take too long to execute (not sure how long too long is)

-The code gets automatically executed over 100 times per second

-To set an interrupt, use **Func**

-To stop the interrupt, use **Func**0

So here is an example of an interrupt that updates the LCD automatically:

```
:FuncLbl "INTERRUPT
:<<Code>>
:.INTERRUPT
:DispGraph
:End
```

Subject 13-Data

The ability to access data is a powerful tool and if used improperly

can cause a crash. So the key here is to use commands as they are intended! Anyway, there are several ways to create and access data. The functions you will want to look at are:

int(
iPart(
(
{
Send(
Get(

Subject 14-Particles

Grammer has its own particle engine built in and this provides for some neat graphics. To use particles, first you need to add them and then you need to recalculate positions and whatnot. So for example:

```
:R▸Pr(            ;This clears the particle buffer
:P▸Rx(2,2         ;This adds a particle at pixel coordinate (2,2)
:Repeat getKey=15
:R▸Pθ(            ;Updates particle data
:DispGraph        ;Displays the graph
:End
```


Commands

Operators

→ This stores the last computed value to a variable. For example:

```
:Return→A'
```

That will store the value output from Return to A'.
Likewise, you can store to some OS variables such as real vars (A through Theta) and Strings (Str1 to Str256). For real vars, use the prefix i (the imaginary number) and Strings just use their string number. For example, 3→iA will store to the OS realvar A. "Hello→Str2 will strore the string "Hello" to Str2. With Strings, if you use the Str1 token followed by a number like 55, you will be accessing th hacked string 155. Str0 is the same as string 10, remember, so Str00 is

the same as Str100.

As another note, when storing to an OS string, if you want to use it as a Grammer string, you need to have ' follow the string name (it adds a newline token to the end of the string).

To store 32-bit results, you will need 2 vars. →AB will store θ' in A and Ans in B. So to store a 32-bit result from multiplication: A*D→AB.

// This is used to start a comment. The comment goes to the end of the line. A commented line is skipped. As a note, the user can include a comment after code.

" This starts a string. The output is a pointer to the string that can be used later to reference it.

π If you put a pi symbol before a number, the number is read as hexadecimal. For example, π3F would be read as 63.

! This has several uses. The first is to work like the not() token in TI-BASIC. So for example, 3=4 would return 0 because it is not true. However, !3=4 would return 1. Likewise, !3=3 would return 0. The other use is with loops. For example, **If A=3** will test if A is 3 and if it is, it executes the code. However, **!If A=3** will execute the code if A is **not** 3. See If , If... Then... End, While, Repeat, and Pause If.

i This is the imaginary i. Use this to access OS real vars. For example, to read OS var A and store it to Grammer var A:

```
:iA→A
```

And to store a Grammer var to an OS var:

```
:B'→iA
```

E This indicates the start of a binary string. This is the exponential E.

Math

/ Used to divide two numbers. The remainder is stored in θ'.
Follow this with a space and this will perform signed division. In other words, 65533/65535 will return 3 (that is -3/-1)

***** Used to multiply two values. The lower 16 bits are stored to "Ans" and the upper 16 bits (for the 32-bit value) are stored in θ'.

- Used to subtract two numbers. Numbers below 0 are calculated as if 65536 was added. For example, 3-6 would result in -3 which is 65536-3 or 65533. If the number goes below 0, θ' is 1, else it is 0.

+ Used to add two numbers. If the number exceeds 65535, 65536 is subtracted from it and θ' is 1. Otherwise, θ' is 0. For example, 65534+99 would return 97, and θ' as 1.

2 Multiplies a number by itself

√(Returns the square root of the number

√(' Returns the rounded square root of the number

sin(Returns the sine of a number. This has a period of 256 and returns a value from -127 to 127. An input of 64 is like 90 degress.

cos(Returns the cosine of a number. This has a period of 256 and returns a value from -127 to 127. An input of 64 is like 90 degress.

abs(Returns the absolute value of a number. If the number is greater than or equal to 32768 (2^15), this returns 65536 minus the number. For example, abs(65533) would return 65536-65533=3.

min(Returns the lower of two values. For example, min(3,A) returns 3 if A is larger than 3 or the value of A if A is less than 3.

max(Returns the larger of two values.

gcd(Returns the Greatest Common Divisor of two numbers

lcm(Returns the Least Common Multiple of two numbers

nCr Will perform the operation n choose r.

rand This generates a random value from 0 to 65535.

randInt(This generates a random integer between two problems. Note that **randInt(**3,7 will generate a random integer from 3 to 6

>Frac This will return the smallest factor of Ans in Θ' and the result of Ans divided by that smallest factor is kept in Ans. So to test for primality:

```
If Ans=Ans>Frac
Text(0,0,"Prime!
```

and performs the bit logic AND on two numbers

or performs the bit logic OR on two numbers

xor performs the bit logic XOR on two numbers

not(inverts the bits of the following value

Logic

This will be explained in terms of "x (logic) y" where x and y are values

= Returns 1 if x is equal to y. Otherwise, it returns 0.

< Returns 1 if x is less than y. Otherwise, it returns 0.

> Returns 1 if x is greater than y. Otherwise, it returns 0.

≤ Returns 1 if x is less than or equal to y. Otherwise, it returns 0.

≥ Returns 1 if x is greater than or equal to y. Otherwise, it returns 0.

≠ Returns 1 if x is not equal to y. Otherwise, it returns 0.

! Returns 0 if the following result is not 0, else it returns 1.

Loops/Conditionals

If x

If "x" is not 0, the line following it will be executed. The line is skipped if "x" is 0. "x" can be any operation resulting in a number. For example:

```
:3→A
:4→B
:If A=B         ;Since A=B is false, the following line is skipped
:9→A
```

!If will execute the code if the statement is false

If... Then... End

This is similar to If except if the statement results in 0, any code between amd including Then and End will be skipped. This works like the TI-BASIC command. For example:

```
:If 3=4              ;3=4 returns 0
:Then
:3→A
:9→B
:16→C
```

```
:End
```

!If...Then...End works if the condition is false.

For(

The arguments for this are:

```
For(Var,Start,End
```

Var is the name of a var
Start is the starting value to load to the var
End is the max value to load to the var

What this does is it loads the initial **Start** value into **Var**. It executes code until it reaches an End statement, then it incrememnts the var. If incrementing goes higher than **End**, the loop finishes and code continues, otherwise it executes the loop again. So for an example:

```
:For(R,0,48
:Circle(32,48,R,1
:DispGraph
:End
:Stop
```

Pause If

This will pause so long as the condition is true for example, to pause until a key is pressed, **Pause If !getKey**

Alternatively, using !Pause If will pause while the condition is false. So to pause until enter is pressed, do **!Pause If 9=getKey**

While

While loops are like If statements addicted to crack-- they just keep coming back. An If statement is content with just checking if the result is true (true=1), but a while loop will not only execute the code up to End if it is true, but it will loop back to try it again! To give you an idea, this will keep looping until Clear is pressed, and while it is at it, it will increment A and decrement B:

```
:0→A →B
:While getKey≠15
:A+1→A
:B-1→B
:End                ;This tells the While loop to End and restart!
```

Alternatively, **!While** will only execute the code if the statement is **not** true.

Repeat

This is a loop that is kind of the opposite of a While loop. This will repeat the code up to an End until the statement is true. So for example, to wait until clear is presed:

```
:Repeat getKey=15
:End
```

!Repeat checks if the statement is false in order to end. For example, to remain in the loop while Enter is being pressed:

```
:!Repeat getkey=9
:End
```

Control

Return This returns a pointer to the next line of code.

Goto This is unlike the BASIC Goto command. This jumps to a pointer as opposed to a label. For example:

```
:Return→L
:<<code>>
:Goto L        ;This jumps to the line after "Return→L"
```

Lbl This returns the pointer of a label. The argument is a pointer to the label name. For exaple, **Lbl "HI** will search for .HI in the program code.
Also, you can specify which variable the label is in. For example, if you wanted to jump to a laqbel in another program, you can add a second argument as the name of the var. For example, to find the label HI in prgmBYE:

```
Lbl "HI","BYE
```

Pause This will pause for approximately x/100 seconds. So **Pause 66** will pause for about .66 seconds.

prgm This is used to execute a sub routine.

Func This is used to define an interrupt. Use this like prgm or Goto.

Asm(This can be used to run an assmebly program. For technical info, see this section. Unsquished ASM programs are not yet supported.

AsmPrgm This allows you to input asm code in hex. (C9 is needed)

ln(This will let you jump forwards or backwards a given number of lines. For example:

```
:ln(3
:"NOT
:"Executed
:"YAY :D
```

Or to jump backwards:

```
:"YAY :D
:"Erm...
:"Yeah...
:ln(-3
```

Graphics

DispGraph Displays the graph screen. You can display another buffer by using a pointer.

Circle(The syntax is:

Circle(**Y**,**X**,**R**,**Method**[,pattern[,buffer

This draws a circle using **Y** and **X** as pixel coordinates and **R** as the radius of the circle in pixels. **Method** is how to draw the circle:

1-Black border
2-White border
3-Inverted border

Pattern is a number from 0 to 255 that will be used as a drawing pattern. For example, 85 is 01010101 in binary, so every other pixel will not be drawn. Use 0 for no pattern. **Buffer** is the buffer to draw to (useful with grayscale).

Pt-Off(

This is used to draw sprites to pixel coordinates. It is limited in some ways, compared to the Pt-On(command, but more flexible in others. The syntax is:

Pt-Off(Method,DataPointer,Y,X,~~*Width*~~,Height[,Buffer

Method is how the sprite is drawn:

0-Overwrite

This overwrites the graph screen data this is drawn to.

1-AND

This draws the sprite with AND logic

2-XOR

This draws the sprite with XOR logic

3-OR

This draws the sprite with OR logic

5-Erase

Where there are normally pixels on for the sprite, this draws them as pixels off.

DataPointer is a pointer to the sprite data

Y is the pixel Y-coordinate

X is the pixel X-coordinate

~~*Width*~~ is 1. More options may be due in the future, but for now, just put 1 :)

Height is the number of pixels tall the sprite is

***By adding 8 to the Method, the data will be read as hexadecimal**

Pt-On(

This also draws sprites, but only to 12 columns (every 8 pixels). This is slightly faster than Pt-Off(and has the advantage of variable width. It also has the DataSwap option that isn't present with the Pt-Off(command. Here is the syntax of the command:

Pt-On(Method,DataPointer,Y,X,Width,Height[,Buffer

Method-This is how the sprite is drawn:

0-Overwrite

1-AND

2-XOR

3-OR

4-DataSwap

This swaps the data on the graph screen with the sprite data. Doing this twice results in no change.

5-Erase

6-Mask

This will display a masked sprite.

7-Gray

This draws a frame of a 3 level gray sprite

DataPointer is a pointer to the sprite data

Y is the pixel Y-coordinate

X is a value from 0 to 11.

Width is how wide the sprite is. 1=8 pixels, 2=16 pixels,...

Height is the number of pixels tall the sprite is

***By adding 8 to the Method, the data will be read as hexadecimal**

Line('

This is used to draw lines. The syntax for this command is:

Line('x1,y1,x2,y2[,Method[,Buffer

So it is two sets of pixel coordinates and then the **Method**:

0=White
1=Black
2=Invert

If **Method** is ommitted, it uses 1 as the default.
Buffer is the buffer to draw to.

Text(

See Subject 11-Text(

To display 32-bit number display. The upper and lower 16-bits must be in a pVar. An example where B is the upper 16-bits and C' is the lower 16-bits:

:Text('0,0,BC'

Using the Text(command with no arguments returns the X position in Ans and the Y position in Θ'.

Line(

This is used to draw rectangles. The syntax for this command is:

Line(x,y,Height,Width,Method

x is a value from 0 to 95 and is the x pixel coordinate to begin drawing at
y is a value from 0 to 63 and is the y pixel coordinate to begin drawing at
Height is a value from 1 to 64 is the number of pixels tall the box will be
Width is a value from 1 to 96 is the number of pixels tall the box will be
Method is what kind of fill you want:

0-White. This turns off all of the pixels of the rectangle
1-Black. This turns on all of the pixels of the rectangle
2-Invert. This inverts all of the pixels of the rectangle
3-Black border. Draws a black perimeter not changing the inside
4-White border. Draws a white perimeter not changing the inside
5-Inverted border. Draws an inverted perimeter not changing the inside
6-Black border, White inside.
7-Black border, Inverted inside.
8-White border, Black inside.
9-White border, Inverted inside.
10-Shifts the contents in that rectangle up
11-Shifts the contents in that rectangle down

Pxl-On(

This turns a pixel on using coordinates (y,x). To draw to a specific buffer, add its pointer as a last argument.

Pxl-Off(

This turns a pixel off using coordinates (y,x). To draw to a specific buffer, add its pointer as a last argument.

Pxl-Change(

This inverts a pixel using coordinates (y,x). To draw to a specific buffer, add its pointer as a last argument.

ClrDraw This clears the graph screen buffer and resets the text coordinates. Optionally, you can clear a specific buffer by putting its pointer directly after. For example, ClrDrawπ9872

ClrHome This clears the home screen buffer and resets the cursor coordinates

Shade(This sets the contrast to a value from 0 to 39. 24 is normal and this is not permanent. An example is:

```
:Shade(30
```

Horizontal This draws a horizontal line on the graph. The syntax is
Horizontal y[,method,[,Buffer
y is a value from 0 to 63
method is how to draw the line:
0=draws a white line
1=draws a black line
2=draws an inverted line
Buffer is the buffer to draw to.

Vertical This draws a vertical line on the graph. The syntax is:

```
Vertical x[,method[,Buffer
```

x is a value from 0 to 95
method is how to draw the line:
0=draws a white line
1=draws a black line
2=draws an inverted line
Buffer is the buffer to draw to.

Tangent(This is used to shift the screen a number of pixels. The syntax is:

```
Tangent(#ofShifts,Direction
```

# of shifts is the number of pixels to shift the graph screen
Direction is represented as a number:

```
1 = Down
2 = Right
4 = Left
8 = Up
```

You can combine directions by adding the values. For example, Right and Up would be 10 because 2+8=10

Disp This will let you change the default graph buffer. For example, if you don't want to use the graph screen, you can put this at the start of the program:

```
:Disp Π9872
```

Also, if you are using grayscale, you can use the following:
Disp ' will set the black buffer.
Disp ° will set the gray buffer.

Pt-Change(This command is used to draw tilemaps. There is currently one method, but more should be added in the future. Here is the syntax:

```
Pt-Change(0,MapData,TileData,MapWidth,MapXOffset,MapYOffset,TileMethod
```

-MapData is a pointer to the map data
-TileData is a pointer to the tile set
-MapWidth is the width of the map (at least 12)
-MapXOffset is the X offset into the map data
-MapYOffset is the Y offset into the map data
-TileMethod is how the sprite will be drawn (see Pt-On())

Fill(

0-Black
This fills the screen buffer with black pixels
1-Invert
This inverts the screen buffer
2-Checker1
This fills the screen buffer with a checkered pattern
3-Checker2
This fills the screen buffer with another checkered pattern
4,x-LoadBytePatternOR
copies a byte to every byte of the buffer data with OR logic
5,x-LoadBytePatternXOR
copies a byte to every byte of the buffer data with XOR logic
6,x-LoadBytePatternAND
copies a byte to every byte of the buffer data with AND logic
7,x-LoadBytePatternErase
copies a byte to every byte of the buffer data with Erase logic
8,x-BufCopy
x points to another buffer. The current buffer gets copied there
9,x-BufOR
x points to another buffer. This gets copied to the current buffer with OR logic.
10,x-BufAND
x points to another buffer. This gets copied to the current buffer with AND logic.
11,x-BufXOR
x points to another buffer. This gets copied to the current buffer with XOR logic.
12,x-BufErase
x points to another buffer. This gets copied to the current buffer by erasing.
13,x-BufSwap
x points to a buffer. This swaps the current buffer with the other.
14,x-CopyDownOR
The current buffer is copied x pixels down to itself with OR logic
15,x-CopyDownAND

The current buffer is copied x pixels down to itself with OR logic

16,x-CopyDownXOR

The current buffer is copied x pixels down to itself with OR logic

17,x-CopyDownErase

The current buffer is copied x pixels down to itself with OR logic

18,x-CopyUpOR

The current buffer is copied x pixels up to itself with OR logic

19,x-CopyUpAND

The current buffer is copied x pixels up to itself with OR logic

20,x-CopyUpXOR

The current buffer is copied x pixels up to itself with OR logic

21,x-CopyUpErase

The current buffer is copied x pixels up to itself with OR logic

22,type-FireCycle

This burns the contents if the screen for one cycle. If **type** is 0, white fire is used, if it is 1, black fire is used.

23,Type,Y,X,Width,Height-Fire Cycle 2

Type is the same as FireCycle and the other inputs are the same as Pt-On(where X and Width go by every 8 pixels.

Input/Computing

getKey This returns a value from 0 to 56 that is the current key press. You can use this chart for values.
Also, getKey(will allow you to see if a key is being pressed. For example, getKey(9 will return 1 if enter is pressed

Input This allows you to input a string. The pointer to the string is returned. (this is not a permanent location, the data will be overwritten the next time Input is used) To get a value input from the user, you can use expr(:

```
:expr(Input →A
```

That will store the result to A

Ans This will return the value of the previous line

expr(This will compute a string as a line of code (useful with Input)

inString(This is similar to the TI-BASIC command. This will return the location of a sub-string. The inputs are where to start searching and the string to search for:

inString(SearchStart,SearchString

So an example would be:

```
:Lbl "DATA→A
:inString(A,"How→B
:.DATA
:HELLOHowdyWoRlD!
```

The size of the input string is returned in θ' and if there was no match found, 0 is returned.

length(This will return the size of a variable (in RAM or Archive) as well as the pointer to the data in θ'. For example, to get the size of the appvar Data:

```
:length("UData→A
```

If the var is not found, -1 is stored to θ'.

length(' This is used to search for a line. For example, if you want to find a specific line number in a program, this is what you would use. The syntax:

length('*StartSearch*,*Size*,*LineNumber*,[*LineByte*

StartSearch is where to begin the search
Size is how many bytes to search in. 0 will search all RAM.
LineNumber is the line number you are looking for
LineByte is an optional argument for what byte is considered a new line.

The output is the location of the string and θ' has the size of the string. If the line is not found, the last line is returned instead.

Memory Access

Get(This uses a string for the name of an OS var and returns a pointer to its data.
-If the variable does not exist, this returns 0
-If it is archived, the value returned will be less than 32768
-θ' contains the flash page the variable is on, if it is archived, otherwise θ' is 0
As an example, Get("ESPRITES→A' would return a pointer to the data of prgmSPRITES in A'.

(Use this to read a byte of data from RAM

{ Use this two read a two byte value from RAM (little endian)

int(Use this to write a byte of data to RAM.

iPart(Use this to write a word of data to RAM, little endian (a word is 2 bytes). For example, to set the first two bytes to 0 in prgmHI:

```
:Get("EHI→A
:iPart(A,0
```

Send(Use this to create Appvars or programs of any size (so long as there is enough memory). For example, to create prgmHI with 768 bytes:

```
:Send(768,"EHI
```

Programs must be prefixed with "E", protected programs "F" and appvars "U"
Also, you can use lowercase letters if you want :)

[

[[

[(

This allows you to write multiple bytes to a RAM location. For example, to write some bytes to the address pointed to by A:

```
:A[1,2,3,4
```

To store some values as words, you can use ° after the number. These will be stored little endian. For example:

```
:A[1,2,3°,4
```

In order to store all values as words, use **[[** instead:

```
:A[[1,2,3,4
```

To directly store hexadecimal, use **[(**. For example:

```
:A[(3C7EFFFFFFFF7E3C
```

IS>(

This is used to read memory. The argument is one of the pointer vars. It reads the byte pointed to by the pvar and then the pvar is incremented (so consecutive uses will read consecutive bytes). For example, to display the hex of the first four bytes of a var:

```
:Get("EPROG→Z
:Text('0,0,IS>(Z,16     ;The bold is the Text( arguments.
:Text('°,IS>(Z,16
:Text('°,IS>(Z,16
:Text('°,IS>(Z,16
```

Archive

Follow this with a var name to archive the var. For example, to archive prgmPROG, do this:

```
:Archive "EPROG
```

Unarchive

Use this like Archive, except this unarchives the var

Delvar

Use this like Archive, except this will delete a var

sub(

Use this to remove data from a variable. the syntax is:

```
:sub(#ofBytes,Offset,"Varname
```

For example, to delete the first 4 bytes of program Alpha:

```
:sub(4,0,"EAlpha
```

augment(

This is used to insert data into a var. The syntax is:

```
:augment(#ofbytes,Offset,"VarName
```

For example, to insert 4 bytes at the beginning of appvar Hello

```
:augment(4,0,"UHello
```

Modes

Fix Text(Use this to set the typewriter delay. The larger the number, the slower the typewriter text is displayed.

Fix Use this to set certain modes. For all the modes that you want set, add the corresponding values together. For example, to enable inverse text and inverse pixels, use **Fix 1+2** or simply **Fix 3**
Here are the modes:

1-Inverse text
2-Inverse pixels. Now, on pixels mean white and off means black. In assembly terms, it reads from the buffer, inverts the data and sends it to the LCD.
4-Disable ON key. This will allow ON to be detected as a key, too
8-Hexadecimal Mode. (Numbers are read as hexadecimal)
16-PixelTestOOB. Returns 1 for out of bounds pixel tests
32-TextMode. 0 is the default, fixed width font. 1 is the variable width font.

Full This is used to set 15MHz mode. Alternatively, if you add a number to the end:

Full0 sets 6MHz
Full1 sets 15MHz
Full2 toggles the speed

15MHz is only set if it is possible for the calc.

Output(This is used to change the font. The syntax is:

-Output(0 will change to the default 4x6 font.
-Output(1 will change to the variable width font.

Adding another argument will allow you to choose your own custom fontset. The argument simply points to the start of the fontset. The output is a pointer to the fontset (custom or standard set)

solve(command subset

CopyVar solve(0,"VarName1","VarName2"[,size[,offset

This will copy the program named by VarName1 from RAM or archive to a new program named by VarName2. If Varname2 already exists, it will be overwritten. So for example, to copy Str6 to Str7:

:solve(0,"DStr6","DStr7

This returns the pointer to the new var and the size of the var is in θ'

The last arguments are optional. Size lets you choose how many bytes are copied (instead of just copying the whole var). You can also add an offset argument to tell where to start reading from.

CopyDataI solve(1,loc_i,loc_f,size

This copies data from loc_i to loc_f. (Forward direction)

CopyDataD solve(2,loc_i,loc_f,size

This copies data from loc_i to loc_f. (Backward direction)

ErrorHandle solve(3,Pointer

This will allow your program to have a custom error handler. **Pointer** is 0 by default (meaning Grammer will handle it). Otherwise, set it to another value and grammer will redirect the program to that location. The error code is returned in Ans. For Example:

:solve(3,Lbl "ERR
:<<code>>

```
:.ERR
:If =1          ;Means there was a memory error
:Stop
:End
```

Ans and θ' are put back to normal when the error handler completes.

Errors:

0=ON

1=Memory

CallError `solve(4,Error#`

This will execute the error code of a Grammer error. For example, to make a Memory error:

```
:solve(4,1
```

Using Error 2, you can input a string for a custom error:

```
:solve(4,2,"Uh-Oh!
```

Physics

R▸Pr(This will clear the particle buffer.

R▸Pθ(This will recalculate the particle positions and draw them. If you want to change the particle buffer, just add a pointer argument. If you want to use a program, for example, as a buffer:

```
:Get("EBUF→A
:R▸Pθ(A-2
```

P▸Rx(This will add a particle to the buffer. Just use the pixel coordinate position. For example:

```
:P▸Rx(2,2
```

P▸Ry(This will cahnge the particle effect. 0 is normal sand, 1 is boiling.

P▸Rx(' This will convert a rectangular region of the screen to particles. The inputs are:

P▸Rx('*Y,X,Height,Width*

This scans the area for pixels that are turned on and adds them to the current particle buffer.

ANOVA(ERR:Does Not Work Yet

Miscellaneous

conj(

Warning: I have no knowledge of musical lingo, so excuse my mistakes

This is a sound command with three inputs. The syntax is:

conj(Note,Octave,Duration

Notes are:

0 =C	1 =C#	2 =D	3 =D#
4 =E	5 =F	6 =F#	7 =G
8 =G#	9 =A	10=A#	11=B

Octave is 0 to 6

Duration is in 64th notes. So for example, a 32nd dot note use 3/64th time. Duration is thus 3.

conj('

This sound routine has several inputs:

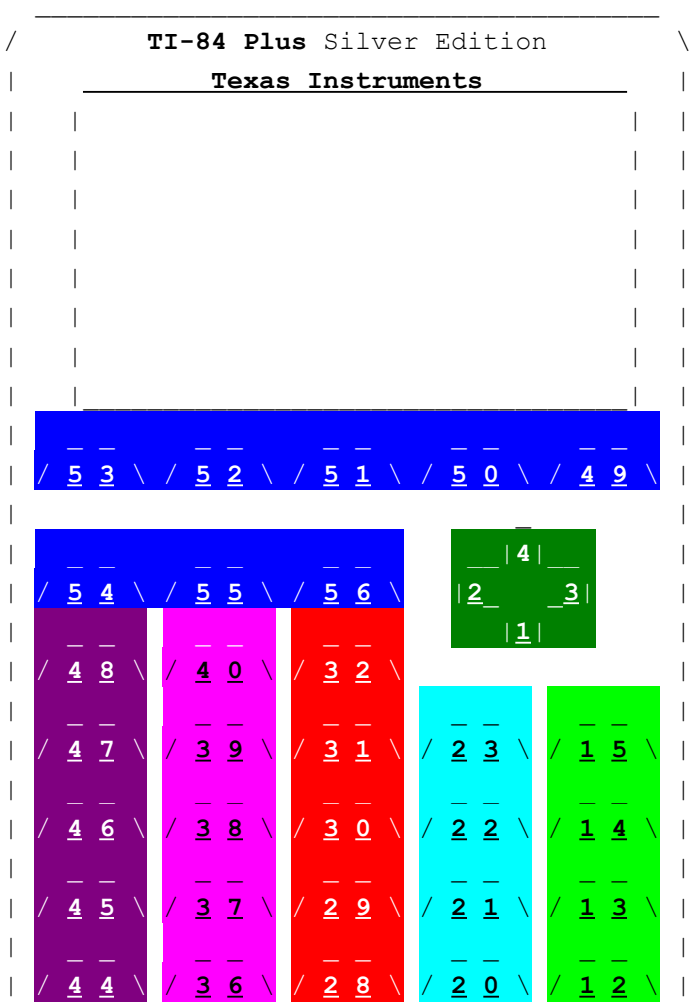
conj('*Duration*,'*Period*

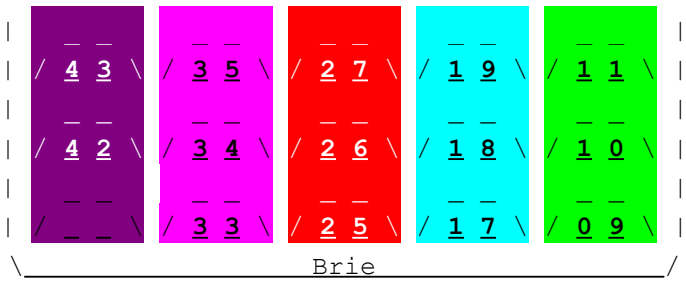
conj('*Duration*,*DataLoc*,*Size*

This reads data for the period directly to save time (intead of converting numbers on the fly). Size is the size of the data in words, not bytes.

getKey Values

Creepily enough (I just checked), this is almost the exact size of my real calc O.O That shows how much I use it... Anywho, you can use this as a guide to the key values ouput by getKey in Grammer. For example, Clear=15





Also, there are the diagonal directions:
5=Down+Left
6=Down+Right
7=Up+Left
8=Up+Right
16=All directions mashed

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 is not compatible
Do not use.
Symbol Var. Compatible with each other.
Named Var. Compatible with each other.

Example Codes

I will hold nothing back and include all sorts of advanced tricks if I can :D I am only doing this so that you can learn all sorts of neat optimisations and get a better feel for the language. I will try to comment the crazy parts: Here is a routine for a cursor:

Cursor Example

```
*use the arrows to move it, press clear to exit
      :.0:Return
      :ClrDraw
      :0→X →Y                ;There is a space after the "X"
      :Repeat A=15
      :Repeat 9              ;9 is never 0, so this will execute once
      :..                    ;This is a label called "."
      :Line(X+1,Y,6,4,2      ;draws a rectangle to the buffer
      :Line(X,Y+1,4,6,2      ;Draws a rectangle to the buffer
      :End                   ;Ends a loop... and also a subroutine >.>
      :DispGraph             ;Displays the buffer, updated
      :prgmLbl ".            ;calls a subroutine at the label "."
      :Repeat A              ;loops until A is not 0
      :getKey→A              ;stores the key press to A
      :End                   ;Ends the repeat loop
      :X+A=3                 ;Adds 1 to X if left is pressed
      :-A=2                  ;subtracts 1 from Ans if right is pressed
      :If >90                ;If X=-1, it is 65535, so this checks both bounds
      :X                     ;If Ans would be off screen, just set it back
      :→X
      :Y+A=1
      :-A=4
      :If >58
      :Y
      :→Y
      :End
```

```
:Stop
```

Error Handler Code

Use this as an error handler in your programs! If the error is an ON error, you get a menu that gives you the option to quit the program, goto the error, ignore it and keep the code going, or completely block on from being pressed again. To avoid drawing to the graph buffer, this uses a location in memory called AppBackUpScreen for the text (9872h). This also happens to be the location of the default particle buffer, so if your program uses the particle commands, be sure to use a custom buffer.

```
:.ERR
:If Ans                        ;If the error is anything except [ON]
:solve(4,Ans
:Disp π9872
:ClrDraw
:Text(0,0,"Oops! ON Pressed
:Text(6,0,"1)Quit
:Text(12,0,"2)Goto
:Text(18,0,"3)Ignore
:Text(24,0,"4)Block ON
:DispGraph
:Disp π9340
:Return
:If getKey(34
:Stop
:If getKey(26
:solve(4,0
:If getKey(18
:End
:If getKey(35
:Fix 4:End
:Goto Ans
```

Technical Assembly Info

Asm(command

For programs that are called in a Grammer program (using the Asm(command), there are two arguments passed via HL and BC. HL points to the next byte to be parsed and BC contains the last computed value. There are a large number of available calls located in a jump table. Use Grammer.inc and Grammer SDK.pdf for information on how to create Grammer programs.

Questions and Answers

Since I started this project 10 May 2011 (2 days into summer vacation), I haven't been able to get any kind of feedback, so I will make up some fun questions for now :D

Q: I have a question, what do I do?

A: You can email me with the email given at the beginning of the document or you can register at one of the sites that I frequent with a Grammer topic (United-TI and Omnimaga are where you are most likely to catch me, but Cemetech, too).

Q: I want to crash my calculator. How can Grammer help me do this?

A: You can either execute assembly code or write data to random spots in memory.

Q: Can I have an example of that?

A: Sure, I guess...

```
:AsmPrgmC7
```

OR

```
:Return→A
:int(rand,-1      ;the -1 is an optimisation for 65535 :)
:Goto A
```

I cannot guarantee that this will crash, but it should make things volatile!

Q: I found a bug! What do I do?!

A: Pick it up gently and inspect it. Try to find what brought the bug to you and let me know so that I can try to find it, too :)

Q: I have some ideas! Would you like to hear them?

A: Yes! I might not be able to implement them or I might have reasons to not implement them, but if I can, I would love to! If you can think of a syntax, too, that would be great!

Q: Can I take a look at the source?

A: Sure, I don't mind! However, if you want to release a modified version, please inform the end user that it is modified and how these modifications change program flow. For example, if you change Grammer to handle only 8 bit values, I would be pretty confused when I get a bunch of bug reports about not being able to use Goto and prgm and whatnot properly :)

Q: <<Your question goes here>>

A: <<My response goes here>>

Thanks

General thanks go to the sites yAronet, tout82, TICalc, TI-Planet, and Omnimaga for getting this attention. Thank you!

More specialised thanks go to:
yeongJIN_COOL from Omnimaga for all the help and interest so far! A lot of the command tokens came from you, too, so thanks! And on top of that, the games you've made are great!
Sorunome from Cemetech for making Tetris with Grammer! Awesome!
Runer112 from Omnimaga for the help in optimisation and coding ideas (this is the guy that performs magic with Axe :D)
Qwerty.55 (a.k.a. Fishbot) also from Omnimaga. Thanks for the tip on how to remove particles quickly that have fallen off the screen!
HOMER-16 and **aeTIos** for helping me organise the readme
Matt- My best friend from back home for getting me to create the custom fonts (and all the code to go with it)!
Paul Reichert for some ideas for additions (including the length(command)
awalden0808 from Omnimaga for your interest and question asking!
boot2490 from Omnimaga for your enthusiasm and example ideas :D

Programs

Programs can be found on TICalc.org at http://www.ticalc.org/pub/83plus/grammer/
The ones I suggest are:
Marsian Invasion (by **yeong**) and Mars2 (optimised a bit by me to be faster)
Block Eater (by me)
prgmGRPHJMP by **awalden0808**
Gravity Sim if you want to see some of the physicsy stuff it can do
Grammer Run (by **yeong**) -- very addicting
Tetris (by **Sorunome**)
And there are many more to come!
EDIT (05-02-2012): Like Tic-Tac-Toe and Checkers!

Grammer Diary

24 February 2012 (09:18)

Again, a bazillion more things have been added that I cannot remember. I fixed the line drawing routine, added grayscale and sound support, a few new tokens, and I just finished up rewriting the drawing commands to work on arbitrary buffers.

28 December 2011 (11:45)

I have added a bunch of stuff that I cannot remember at the moment, but I am right now redesigning the parser that could as much as double the speed of programs. To give you an idea, this code took 22 seconds with the previous version and 11 seconds with my current work:

```
:For(A,0,59999
:End
```

I expected maybe a 1 or 2 second speed increase, so I thought I might have messed up some code and put it into 15MHz to start. so when I plugged it in to that and it took 4 seconds... well, I am happy :) Plus, I rewrote the number conversion routine and it is now a lot simpler, smaller, and faster. It now takes less than half the speed. In other words, using regular numbers is a lot closer to the speed of using hexadecimal digits.

Since updating the LCD and pausing will take the same amount of time, regardless, programs using fewer pauses and LCD updates will get a big speed boost (I got one program to reach 8 times faster). The only issue is that there is at least one bug and I cannot find it :/ it is preventing Tetris from working, but I have tested other games that work fine.

10 November 2011 (14:25)

Fixed up the Asm(command (it only works with squished programs), I added the Hexadecimal mode, added the IS>(command (lovely for file reading). I also modified the commands that use var name inputs to allow for vars such as Str1 and Pic7 (as opposed to just programs and appvars).

Also just added Archive, Unarchive, Delvar

08 November 2011 (11:11)

Added a few more things that I cannot remember. But I did just add the Asm(command :D

20 October 2011 (19:54)

Added **!**, **Pause**, **Pause If**, **getKey(**, and I tried fixing up the sprite code a bit to clip sprites going out of the buffer. But even more important, Grammer is now an APP!

18 October 2011 (11:10)

Added the **[** token and I fixed up the Line(' routine to work well, now. The line routine is from Axe and has been modified a bit to work with Grammer (and has a few added features).

15 October 2011 (14:46)

Finally added in a system for modes. Currently there is inverse text and inverse pixels.

15 October 2011 (11:29)

Added a call that saved a few hundred bytes. Was 5916 bytes, now it is

5825 bytes. I said a few hundred, so I must have added stuff, right? I added the Fill(token with its 14 sub commands :)

14 October 2011 (22:52)

Changed the particle engine so that when a particle goes off screen, it is removed from the buffer. You can now have an infinite waterfall so long as it falls off the screen and your batteries last forever XD

14 October 2011 (11:13)

Added tons of neat features since then. Updated DispGraph and Text(and fixed the space token. Added these:

```
(          ;Added to the readme
{          ;Added to the readme
int(       ;Added to the readme
iPart(     ;Added to the readme
Send(
i
Fix Text(
R▸Pr(
R▸Pθ(
P▸Rx(
P▸Ry(
Func
Disp
```

18 September 2011 (19:59)

Added Tangent(a few days ago, made lots of examples (including a gravity simulator!). Also, I started work on the App version today which works except where SMC is used. The official program syntax will now be to start programs with **.0:Asm(prgmGRAMMER** because the . will make the rest of the line get skipped (read as a label). Also, I was requested to start a TI-82/83 version, so I want to do that :) Finally, Grammer has been recognised by TICalc as a valid programming language! There is now a directory for Grammer games and programs :)

2 August 2011 (18:00)

Added π as a way to use hexadecimal inputs.

2 July 2011 (19:26)

An EnPro fan e-mailed me with a request for a sprite command that could draw to pixel coordinates. Because of this, I have gotten to work on a better sprite command and I decided to test drive it with Grammer... and it works! It isn't as complete as I would like it to be (It doesn't have an option for width or DataSwap), but it is much more user friendly than before. It can only draw sprites 8 pixels wide and it does not clip the sprite. I left the Width argument in case I can later add code to use it.

15 June 2011 (07:43)

I modified the sine/cosine routine to speed it up slightly. I originally got the gist of the algorithm from Axe code, but I have since remodeled it and optimised it so much that it only loosely resembles its original form.

14 June 2011 (17:27)

After playing around with my other hobby (math), I wrote an algorithm to compute **nCr**. The cool part about it is that it is a legitimate use of some of

the math that I research, so I feel useful ^-^. For the curious, this is a polynomial time algorithm (no computing factorials). This is what the algorithm looks like in Grammer Code as a subroutine. Inputs are N and R and the result is output in D:

```
:N-R→N
:If <R
:N→A R→N A→R        ;Cuts down computation time
:1→C →D
:For(A,1,R
:C*N
:/A→C
:+D→D
:N+1→N
:End
:End                ;Exits the subroutine
```

13 June 2011 (17:09)

Added the Repeat command as another loop command.

13 June 2011 (08:06)

Added the For(command as a new loop command.

12 June 2011 (22:24)

I added code to convert tokens to ASCII for cases like text display or searching for a var. This means that named vars with lowercase letters (like some appvars) can now be accessed and tokens are now displayed properly with the text command. Also, I added the lcm(command and made a few mini games that still need finishing touches.

11 June 2011 (14:08)

The prerelease never happened, but I have added in several new Text(syntaxes to allow for displaying numbers, I added the Full command to allow manipulation of the processor speed (set to 15MHz, 6MHz, or toggle), I added ClrDraw, ClrHome, min(, max(, abs(, rand, gcd(, Shade(, Horizontal, Vertical

6 June 2011 (12:51)

Preparing for a pre release of Grammer... Also, I added a modified circle routine that is fast O.O

4 June 2011 (16:45)

I decided to add in one of my square root routines as well as Pxl-On, Pxl-Off, and Pxl-Change. I also made it so that the user could obtain the rounded square root (rounded to the nearest whole) and I made the pixels not draw if they were off screen. The first thing I did was make some circle sub routines to play with :) Now I think I will try to make the sin/cos routines a little more accurate by adding rounding to that, too...

1 June 2011 (19:48)

I randomely decided to add **prgm** as a method of executing sub routines. I also added **Pt-Off(** as a "concept command" for a sprite routine that draws variable size sprites to pixel coordinates. Currently it has a weird sprite data syntax and the sprites need to be a multiple of 8 pixels wide, but it does draw to pixel coordinates.

17 May 2011 (21:50)

I have been thinking about adding assembly support like this:
AsmPrgm will execute a simple hex opcode

AsmComp(will use multiple line support with whatever goodies I add. I have a program that can compress and execute code like this that has comment support. Maybe I will add support for labels and some instructions...
AsmComp(will load 5 pointer vars in a row (like R,S,T,U,V) into the register pairs af,bc,de,hl,ix and then use an argument to execute a call in a jump table. It will then return the values of the registers in the pointer vars. This will likely be used only by me as a way to debug new routines.

These are only tentative. The first will probably be added, but if the other two are added, they will likely be modified often and won't remain backward compatible.

17 May 2011 (19:58)

I have been working on other projects, but I came back to this today. I decided to try and make a text drawing routine that could draw to pixel columns (instead of every fourth one). I started thinking of ways to do this when I thought that making a general sprite routine would work better. The problem? I've never made a sprite routine that worked on pixel columns. So I started coming up with ideas for how to draw this and I thought that I could make a general sprite routine for any size sprite that could be mapped to pixel coordinates. To do that, I made a mask routine for masking the sprite data and the screen data. However, I side tracked myself and ended up using the mask routine as a way to make rectangle routines, so all that I accomplished was that. I wrote it from scratch and it should be faster than the one I made for BatLib because I draw the boxes as sprites, so I do not need to draw whole rows. Plus, it works unlike some of the OS routines and it has 10 fill methods.

14 May 2011 (16:11)

After just getting back from a baby shower 15 minutes ago, I have added reading and writing bytes and words. I plan to add the same ability for nibbles, too, but for now, this should be great for modifying sprites and game data.

14 May 2011 (11:00 ish)

I have now added While loops (that can be nested) as well as the Get(command to start referencing OS variable data. I have also been putting some work into the tutorial, but that is going to be a tad difficult. Pretty soon I will need to add in some rectangle routines and text routines

13 May 2011 (almost noon)

I have been trying to tease out a bug that causes a crash every so often. During program execution, there is no problem, but when it exits, it crashes. Since some LCD stuff was going crazy, I wrote an LCD routine to display the graph buffer and while I was at it, I decided to make a routine to test the ON key (to use as a delay for the LCD writing). Now, I have a way to break out of a program, but it didn't fix the crashing. I am going to see if it has anything to do with the stack pointer...

--A half hour later--

... and this is why I need to practice with mnemonics. I had the hex comment correct, but I had the mnemonic incorrect... For some reason, I was calling the routine to display the graph buffer instead of jumping to it. I am going to see if there is another bug there, though.

--Another half hour later--

... and this time it is corrected in full. What happened was in an If statement,

if the result was true, it jumped to the start of the program (where the address of the start is pushed). I should have either jumped to a spot 11 bytes ahead or used ret. I went with ret, this time. It worked before because the code structure was a little different than it is now.

13 May 2011

After working out some kinks in the program flow and fixing up how string arguments are handled, labels, strings, and commands with arguments will work properly, now. I thought labels were working properly yesterday, but when I decided to do **Lbl "HELLO→A,**it didn't work. After some debugging, I figured out that it was looking for .HELLO→ as the label. I only figured out this was an issue when I was testing out the new sprite command and I tried referencing a label to source data from it. I did **Pt-On(0,Lbl "HELLO",0,0,1,8** and it was completely malfunctioning. I then found out that there was also the problem that:

- Arguments weren't able to be read one after the other because the program counter wasn't being updated properly
- The ending quote was being parsed as a starting quote.

I tried fixing the parsing first and my first few attempts failed. I ended up doing things like making it so that only the last argument was read until I figured out that I would need to include code before each consecutive read to update the program counter (instead of making it update when it read a comma).

Fixing the string and label problem wasn't too bad and only required a few bytes of code.

So, yeah, the details of the new sprite command are that it uses Pt-On(, has six drawing methods. It uses byte data instead of nibble data

12 May 2011 (later)

Finished the Lbl command so now data referencing can be started! After *finally* finishing up that SearchString routine (it was a lot simpler than I thought it would be...), users can now use labels up to 764 bytes long, but really, I should cap that to 10 or something x.x

12 May 2011

Today I have added few more things including:

Addition (with 1-bit carry)
Subtraction (with 1-bit carry)
Multiplication (with 16-bit carry)
"Squared" symbol (^{2}) works (with 16-bit carry)
Logical operators like **>** and **=**
Negative **(-)**
sin(
cos(
If
If ... Then... End
getKey
End
"

I am also starting work on the **Lbl** command which will be very useful to have and I also want to get started on graphics commands. If I finish **Lbl**, I will be able to pretty easily implement some sprite commands *cough*

11 May 2011

Okay, actual progress was made besides planning and whatnot. The basic outline of the program was started and I made my first real use of an assembler. Thanks Kerm for your DCS SDK! I have finally managed to figure out how to compile assembly source code and have it packaged as a program! Now I need to figure out how to turn it into an App...

The current syntax allows the user to put the name of the var with Grammer code in Ans (as a string) and then do Asm(prgmGRAMMER to start the Grammer parser. It currently has Ans as a debugging tool (it displays a value) and it can convert numbers to 2 byte integers. I also added division, commenting, storing to and reading variables, DispGraph, Goto, and Return

10 May 2011

Progress on Grammer started after about a month of debating and ideas.