

TI basic:

The calculator language

by BIT01

Note: command placement may be different for your model.

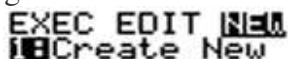
Getting Started

Programming on the Ti calculators is very similar to programming on the computer. Many of the concepts and some of the commands will carry over into programming languages you will learn for the computer. First, to follow this programming class, you must have a Ti-83, Ti-83plus, or Ti-83 plus silver edition. There are slight differences when programming for the other TI calculators, and the menus are different than the ones explained here.

Why Program?

Programming can be helpful in many ways. Once you learn to program, you will be able to write a program that will solve all of your frequently used math equations. You will be able to write a program as a game for other people. And once you learn the basics of this programming language, the concepts carry over into any other programming language you may encounter. If you're planning to get into computers or mathematics, or even the science department, learning to program is possibly one of the best things you can do.

To get started, turn on your calculator and press the program button, or [PRGM]. Press the right arrow twice until you get to the NEW menu.



EXEC EDIT NEW
Create New

Press enter to create a new program. Type in the name, and press enter.

Note: Program names may only be 8 characters long

This is a new program. You are now ready to program.

The “disp” command

The Disp, or display command is one of the most basic commands. It tells the program to display something on the calculator screen. To display something, press [PRGM], then go to the "I/O" menu. Go down to "Disp" (third one down) and press enter. If you want to get this function faster, press [PRGM], {RIGHT ARROW}, [3].

Note: These commands will not work if you yourself write them out. They can only be written from these menus.

You now need the program to know what to write. After the Disp command, put the number 123. This is telling the program to display 123. Your program should now look like this:

:Disp 123

and that's it. Now, quit the program editor ([2ND][DEL]). Press the [PRGM] button and go to program TEST and press enter. The screen should now say prgmTEST. This is how you run programs. Press enter again. It should say:

123

Done

The Done tells you that the program has finished.

Next, we'll learn how to have the program display words.

To display words, use the typical Disp command. Instead of putting a number, put quotes ([ALPHA][+]). Put whatever you want to say inside the quotes, and end it with another quote. For now, try putting in "Hello World!" (For the exclamation mark, press [MATH][Right][Right][Right][4].) Run the program. It should look like this:

Hello World!

Done

You can also display variables such as Disp A and Disp Str2

The "Input" Command

The next command is the input command. This is accessed by going to [PRGM][Right][ENTER]. It asks for a number or letters, and stores them to a variable. It looks like this:

Input *"Whatever you want it to ask"* , *Whatever variable you want it to store the answer to*

So, write a program that inputs two numbers, then adds the numbers together.

It should look like this:

:Input "A?",A

:Input "B?",B

:Disp A+B

And it should run like this:

A? 1

B? 1

2

If you wanted to add numbers into consecutive variables, then you could use the prompt command. This is accessed by [PRGM][Right][2]. Then You put the variables you want to ask and separate them by a comma. The completed command would look like this:

:Prompt A,B,R

And the program would look like this when run:

A=? 1
B=? 2
R=? 3

That's all with the input command. Next, we'll learn about strings.

Strings

The calculator can not only support numbers, it can also support text. Like numbers are stored to variables (A,B,C,D,E, etc.), text can also be stored to variables. These variables are called Strings. The Ti-83, 83+, and 83+SE have ten strings. With other calculators like the 89 and 92, variables and strings can be named anything they want. The ones used in this tutorial have Str0 through Str9. Strings can be used just like other variables. You can use the command "Disp Str7", or the command "Input "What is your name?",Str7". To store something to a string just use the store command, and put quotes around the text. Example: : "Hello World!" -> Str7.

To access the string variables. Press [VARS][7] and the number of the string you want.

If-then statements

If then statements work like this: If the variable you request equals what you say, then do the following lines of code. If not, then skip the following lines of code until an "End" is reached. It works like this:

```
:If A=1
:Then
:Disp "Hey"
:Pause
:End
:If A=2
:Then
:Disp "Bye"
:Pause
:End
```

If you have ONLY ONE LINE OF CODE under the If, you don't need the "Then" and the "End", for example, This:

```
:If A=1
:Then
:Disp "Hey"
:End
```

Could be replaced by this:

```
:If A=1
:Disp "Hey"
```

This has the same effect, but will only work on one line! If you have more than one line inside the if-then, be sure to use then and end. For the equals sign, press [2nd],[Test (Math)],[Enter].

Be Careful	Be careful about putting If-Then statements inside other If-Then statements. Although this works just
-------------------	---

fine, it soon gets extremely confusing, and usually one more or one less "End" gets put at the end than needed, causing a syntax error, or for the program to skip the rest of the code.

If then statements are very helpful, and you will use them many more times in your programming. Let's try writing a program. This program should ask you for a number, then, if the number equals 1, display "Hi!", and if the number equals 2, display "Bye!". See below for what it should look like, but try to write it yourself before you look at the answer.

```
:Input "What?", A  
:If A=1  
:Disp "Hi!"  
:If A=2  
:Disp "Bye!"
```

If you want it to do something after all the if-then statements, you know how to do this, but we'll remind you anyway. Put the commands that you always want to be executed after every If-Then statement after all the statements. Basically, add this on to the end of the program you just wrote:

```
:Pause  
:ClrHome
```

Pause is a command that will wait for you to press enter before it goes on with the rest of the program. It is obtained by pressing [PRGM],[8]. ClrHome clears the home screen. It is obtained by pressing [PRGM],[Right Arrow],[8]. Run the program. It should now ask you for a number. Type in 1 or 2. It should either say "Hi!" or "Bye!", and then wait for you to press enter. After you press enter, it should clear the home screen, and say "Done".

While Loops

While loops are commands that aren't used as much as other commands, but are an essential part of programming when you need them. While loops, like their names, repeat themselves as long as the variable equals or does not equal something. Here's an example:

```
:0->D  
:While D=0  
:Input "D?",D  
:Disp D  
:End
```

This program will keep asking what D is until you enter some number besides 0. How do these help? Well, here we will try writing a program that will show you.

The Guess Program

This is the most advanced program that you have written yet. a copy of the code is found below. What you have to do is make a program that has a number in mind, and it asks you to guess the number. If you get it right, then it says right, if you don't then it says that you got it wrong, and it asks you until you get the number. This program

requires the commands Disp, Input, While, and If-Then. Good Luck

If you're looking for the code for the guess program, here it is:

```
:15->A  
:ClrHome  
:0->G  
:While G=0  
:Input "What is the number?",N  
:If N=A  
:Then  
:Disp "You got it!"  
:1->G  
:End  
:If N<A  
:Disp "Too High"  
:If N >A  
:Disp "Too Low"  
:End
```

That was it. You might ask "What the heck is going on with the variable G and the While loop? It's what makes the program ask you again if you get it wrong. G is whether you have guessed the number or not. If you haven't, it's 0, and if you have, it's 1. While you haven't guessed the number (G=0), then ask for the number and tell whether it's right or wrong. After you guess it, G is set equal to 1, and since G no longer equals 0, it ends the while loop, thus ending the program. You will be faced with problems like these all the time when programming. It takes a good knowledge of the commands you have available to you and knowledge of how to use these commands. Don't be afraid to experiment!

That's the best way to learn the feel for the programming language. If you did good on this program, you'll do well on others. If you didn't play around with the language a bit to get a feel for the commands and how they work.

One last thing before you move on. What if you want the number to change every time? There's a function that makes a random number. Press [MATH],[Right Arrow],[Right Arrow],[Right Arrow],[5] to get the randInt(function. In the parenthesis, type a range (for this program, 1 to 100 is good). In the guess program, in place of 15->A, Put the function randInt(1,100)->A.

The “getKey” command

The getKey command is a command that gets a keypress and stores the information to a variable. This is useful if you have interactive graphics, or many other things. The command looks like this:

```
:getKey->D
```

First, you need to know how the keyboard is mapped out. The tens go from the top row down, starting with 10 and ending with 100. Ones go across to the right, and start with 1 (not 0!), and end with 5. It may be a little confusing, but here are some examples:

Y= (11).....Graph (15).....Enter (105).....2nd (21).....Alpha (31).....Apps (42)

So, find the coordinates of the Math key. You should have gotten 41. Next, try to find the coordinates of the 1 key. It should be 92. There are five exceptions to this:

The On key (You can't use this during a program or else it breaks it. This has no mapping).....Left Arrow (24).....Up Arrow (25).....Right Arrow (26).....Down Arrow (34)

However, getKey doesn't wait for you to press a key. If you don't have a key pressed, it skips over everything. So, how do you have it wait until you press a key? (Note, sometimes you don't want to do this, so just leave out the following lines of code).

```
:0->D
:While D=0
:getKey->D
:End
```

And here you go on to...

```
:If D=24
:BlahBlahBlah...
```

The Key Program

This next program that you should write should clear the homescreen, wait for the user to press a key, and then tell what they key is. You don't have to map out all the keys on the keyboard, just popular ones like 2nd and Arrow Keys. A copy of the code is found below.

Here's the code to the Key program.

```
:ClrHome
:0->D
:While D=0
:getKey->D
:End
:If D=21
:Disp "2nd"
:If D=31
:Disp "Alpha"
:If D=24
:Disp "Left Arrow"
And so on
```

This will be the end of section 1. Try programming a bit to get a feel for the commands. Remember, the best way to learn this stuff is to experiment!

Lbl's and Goto's

The Lbl command stands for Label. You use these to label certain parts of your program. Then, you use the Goto command to go to these labels. It works like this:

```
Lbl MM
:Input "1,2?",A
```

```

:If A=1
:Goto EF
:If A=2
:Goto YU
:Lbl EF
:Disp "You entered 1"
:Stop
:Lbl YU
:Disp "You entered 2"
:Pause
:Goto MM

```

The "Stop" command forces the program to stop. You can get this by going to [PRGM] {F}. You can have more than one goto going to a single Lbl. Work with this a little bit. We usually put a label at the very beginning of the program called ST, for start, and one right above the main menu, called MM.

Labels can only be 2 characters long, and they can only be uppercase letters or numbers.

Menus

Menus are a good way for the program to interact with your users, and a good way for them to navigate your program. A simple menu is done with one command, the "Menu(" Command. It is accessed by going to [PRGM] {C}. It works like this:

```
:Menu("Title","Option1",Label1,"Option2",Label2)
```

A sample menu would look like this:

```
:Menu("Main Menu","New Game",NG,"Load Game",LD,"Save  
Game",SV,"About",AB,"Quit",Q)
```

Now, obviously, you will need to make Labels NG,LD,SV,AB, and Q somewhere in your program for the Menu to go to. You can only have seven options in your menu. So, try adding a menu to the front of your guess program with a start and quit option. For this quit option, just use the "Stop" command under your label.

Setting up for drawing

There are a few things that you need to do before you start drawing menus. When a program draws graphics, it outputs them in the graph part of the interface. First, you need to get rid of anything that was previously drawn there. This is done by the command of **ClrDraw**. This is accessed by pressing [2nd] [DRAW] [Enter]. If you want the default window size to be -10, 10, -10, 10, then do the command of **ZStandard**. This sets the window size on the graph screen to it's default size. This is best to use, because most people's calculators have this already set. It is accessed by pressing [ZOOM] [6]. If you want to set the window to something else, press [VARS] [Enter], and store numbers to the different window variables. Sometimes, people will have equations entered into the Y= area of the calculator. It has to redraw these equations every time you enter something into the graph area, and they clutter your menus, so it is best to get rid of them. Tell the

program to delete the variable **DelVar** is accessed from the catalog. The Y1, Y2, etc. variables are accessed by [VARS] [Right Arrow] [Enter].

Finally, you will need to take the X and Y axes off of the screen if you are doing menus, etc. You can do this by pressing [2ND] [ZOOM], and scrolling down to **AxesOff**. You should now be ready to begin drawing.

Graphing

For graphing, it would be best to leave the axes on. To graph, you will need to have the user input equations using strings, then convert them to Y variables, then display the graph screen. Inputting to a string you know how to do, just use the input command. Next, you will need to convert the strings to Y= variables. This is done with the **String>Equ** command, which is accessed in the catalog [2nd] [0]. The syntax is this **String>Equ(StrX,Yx)**. This puts the equation that was input into the string into a Y variable. Finally, you will need to display the graph. This is done by the **DispGraph** command, in the program menu, or [PRGM] {Right Arrow} [4].

Section 3.3---Graph-Coordinates Drawing

There are two types of coordinates used when drawing. In this section we will discuss Graph-coordinates drawing. Commands such as lines, circles, functions, etc. use the coordinates on the graph axes. The way it's displayed on the screen changes as the window variable change, so that's why it's good to use ZStandard at the beginning of your program. These types of coordinates tend to be faster for the calculator to interpret and more dynamic, but have a tendency to move around if the user changes the window.

There are several commands that use graph coordinates drawing.

Line(function.

This will probably be most frequently used function that you will use on the graph, maybe besides text, depending on what you're doing. Line is accessed from the draw menu [2ND] [PRGM] [2]. The syntax of the function looks like this

:Line(X1,Y1,X2,Y2)

I give my thanks to my 3D programming buddy CG for this one. At the end of the Line(command, add a ,0), to make the function look like this: **Line(0,0,4,4,0)**. This draws a white line instead of a black one. This is great if you want to erase something from only one part of the screen, or do real-time moving of objects (white lines over the old lines, then redraw the black ones in the new location)

Circle(function

The circle function is not good to use for menus, because it takes a long time for the calculator to interpret it and draw it (this was Argon's problem for a while). The circle may seem distorted when it's drawn, but this is because of the window set on the

calculator. The X and the Y are equal, but the actual screen size is longer than it is high. The circle command is accessed from the Draw menu [2ND] [PRGM], and the syntax is this:

Circle(*CenterX1,CenterY1,Radius*).

Screen-Coordinates Drawing

The second type of coordinates for drawing are screen coordinates. These count actual pixels on the screen, and start at the upper left hand corner. The screen is 62 by 96 pixels. Using these coordinates keeps menus rigid and in place, but it takes the calculator longer to draw, and it may require a lot of trial and error until you get the idea.

Text(function

The text function works just like the Disp function (You can write text, numbers, variables, strings), but displays it on the graph screen, allowing for labels or menus. Text is in the draw menu [2ND] [PRGM] [0]. The syntax is this:

Text(*StartX1,StartY1,Text*)(Written text (not variables or numbers), needs to be in quotes)

An example of a menu using lines and text

Pxl-On(function

The Pxl-On function draws a single pixel on the screen at a point you specify. The syntax is Pxl-On(X,Y).

Pxl-Off(function

Works just like Pxl-On, but turns a pixel off (to white).

Pxl-Change(function

Changes the pixel (Black to white, white to black)

Pxl-Test(function:

Tests whether the pixel is on at the current coordinates. Returns 1 if true, or 0 if false

Pictures

Pictures are an efficient thing to use in your program. However, I would not use them too much. People have drawings that they store in pictures, and if you overwrite them, then they'll get mad at you. People may have archived pictures, making your program return an error "Error: Unarchived". So why use them at all? Maybe you want your program to be able to export a picture. Or sometimes it's essential to use a picture to save many lines of code. To store a picture, first you have to draw it on the graph screen. Once you've

done that, use the token **"StorePic"** from the [2nd]{Draw}{Rightarrow}{Rightarrow} menu. It should be the first one. **GDB's** are **Graph Data Bases**, and they store all the Y= variables into one variable, which can be recalled later. You want a Pic. Type in the name of the picture that you want. So it will look like this :StorePic 5. To recall a picture, use the **RecallPic** function in the same menu. That's all there is to it!

" Input" revisited

There is another way to use the **Input** function. Instead of adding stuff behind it, leave it blank. This will do something completely different. It will display the graph screen, and display a cursor that you can move around with the arrow keys. If you have coordinates turned on, it will also display the coordinates of the cursor. When you press enter, it stores the cursor's X and Y position to the variables X and Y. How is this useful? Well, this is the exact function that Argon uses. When at the main screen, or when drawing an object, notice the cursor and the coordinates. How do you use this in a menu? With a very long if statement. Say you have a button drawn at 9,9 with a radius of 1. This would make a 2x2 square in the upper right hand corner. Here's how you check:

```
:Input
:If X>8 and X<10 and Y>9 and Y<8
:Goto BP
```

This command may come in useful.

Advanced Menus

Now, with what you know, you can make BASIC a powerful tool. Something that your program should have is a good interface. The GUI stands for Graphical User Interface. Your GUI should match that of other applications/programs as much as possible, which makes it much easier for the user. As said in the miscellaneous section, you must pretend that your user is incredibly dumb. This will deal with all the possible errors they will make, and will reduce the length of manuals/tutorials. So, let's begin making advanced menus.

First, you will want your program to clear your graph screen, and remove the axes, so it's not showing up behind the menu. Then, you can use the Line(command to draw the lines in your menus. Using the text command, you can then draw text. How do you use this? Well, Argon, Sirius, Polaris, and Chromium use nothing except for lines and text to make all their menus. Sirius and Argon use something called Dynamic Menus in conjunction with this, which will be explained in the next section. How do you find out where to put the lines? At the place where you want to insert the lines, put the "Input" command. Then, run the program. It should stop at the place where you put the "Input", and show you the coordinates. Write down the coordinates, then press "On" to break the program. Press "Goto" and replace the "Input" command with the lines using the coordinates you wrote down. Some menus can be incredibly simple, and some can be complex.

Chromium's menu is set up in about 8 lines. Argon's however, gets set up in about a hundred lines.

How do you get user input? Here's an example menu:

```

:AxesOff
:Lbl MM
:ClrDraw
:Text(1,15,"Main Menu")
:Text(8,8,"1: Choice 1")
:Text(14,8,"2: Choice 2")
:Text(20,8,"3: Back")
:0->D
:While D=0
:getKey->D
:End
:If D=92
:Goto C1
:If D=93
:Goto C2
:If D=94
:Goto ST
:If D=45
:Goto ST
:Goto MM

```

Here's what's happening. The menu is being displayed, then it's waiting for you to press a key. If you press 1, it goes to Choice 1 (C1). If you press 2, it goes to Choice 2 (C2). If you press 3 (94) OR clear (45), it goes to the start. If you didn't press any of those, it goes back to the beginning of the menu (MM).

Dynamic Menus

Dynamic means changing. You can have your menu output prompts to the user. Argon does this, where it says "Welcome" on the top. Notice that this changes depending on what you do. Doing this is simple. Text will overwrite itself on the screen. Here's an example of Argon's dynamic Menu

```

//Setting up menu stuff here
Text(1,45,"Drawing...")
//Drawing stuff here
Text(1,45,"Welcome! ")
//Input goes here

```

You can do this with options being on and off, too such as in Chromium's options menu:

```

If X=1
Draw Stuff
If X=2
Draw Different Stuff

```

Dialog Boxes

There is no command for dialog boxes. Rather, they are made by outputting more lines and text to the screen. It's also good to have drop shadows behind the dialog boxes. And

yes, this is all possible in BASIC. When creating a dialog box, it is best to clear the portion of the screen where you will be putting your dialog box. Do this with a for loop and the inverse line command. Let's say we want to make a dialog box that takes up most of the screen, that is, from -8,8 to 8,-8. Here is the code for clearing the screen:

```
:For(I,8,-8,0.2)  
:Line(-8,I,8,I,0)  
:End
```

Next, you'll want to draw your outline of the box and the drop shadow. This is done with 6 lines. In the example above, the outline would be like this:

```
:Line(-8,8,8,8)  
:Line(8,8,8,-8)  
:Line(8,-8,-8,-8)  
:Line(-8,-8,-8,8)
```

The drop shadow is done by two lines 0.3 from the left and bottom of the dialog box. For example:

```
:Line(8.3,7.7,8.3,-8.3)  
:Line(-7.7,-8.3,8.3,-8.3)
```

Now that the dialog box is set up, you can begin to load content into it. Dialog boxes usually have a title, content text, and one or two buttons. The text is done just by the Text(command, and the button is done by more lines and text. So, using a dialog box requires a lot of code, but it's worth it for the graphics and the interface with your user.

Memory Management and Linking

Before you start creating files, it is good to learn how to do simple memory management with your BASIC program. This involves archiving, unarchiving, and deleting variables. For now, let's archive the variable "A". (Be sure to unarchive it before you're done!). The Archive and Unarchive commands are located in the memory menu. Press [2nd][+][5] to get Archive, and [2nd][+][6] to get Unarchive. DelVar (which deletes the variable) can only be found in the Catalog. Press [2nd][0][D], and find the DelVar command. So, let's make a program that stores a value to a, archives it, unarchives it, then deletes it.

```
:0->A  
:Archive A  
:Unarchive A  
:DelVar A
```

And that's it! Now, with this new knowledge, let's go on to files.

Creating lists, RAM files, and saved games

Creating Lists

Creating lists is fairly easy, but reading from them and using them as files gets very complicated. It's also a good introduction to what you'll be doing A LOT of in assembly. But, first thing's first. Why use lists over matrices, etc.? The nifty thing about lists is that they're the only variable on the calculator that can be modified by a BASIC program that can be named. There are only A-J in matrices, A-Z in variables, etc. But lists can be named LFILE or anything else. The calculator has six built in lists, L1-L6. These should not be modified by a user's program unless the purpose is to specifically import or export data from/to them. It's best to create one's own list. Storing data to a list is much like storing it to a variable. However, one must use {brackets} and commas. To store the numbers one through five to a list (L1, for example), use the following code:

:{1,2,3,4,5}->L1

To access the L1 token, press [2nd][1].

Custom Lists

Now, let's create a custom list! Custom list names can only be 5 letters long, and must have the small "L" in front of them to denote their listiness :). To create a list called LABC, store data, such as {1,2,3} to it. To get the small "L" token, press [2nd][List]{Right}[B]. So your code should look like:

:{1,2,3}->LABC

Make sure you name the lists for your program something unique to your program.

Specific Elements in a list:

It's important to know how to store and read specific elements in a list. This is done with using parentheses after a list variable. For example, to read the 7th number in L1, you would recall L1(7). To store a number to a place in the list, use something like ">L1(7)". Be warned! that reading or writing to an element that does not exist (for example, trying to store 1 to L1(6) when L1 only has 3 entries) will cause an error.

Reading Elements from a list:

A good way to read lots of elements from a list is using a For loop. To read every element from list 1, use this loop:

**:For(I,1,dim(L1),1)
:L1(I)
:End**

The dim(function returns the length of the list. It's accessible from [2nd][List]{Right}[3].

This list goes from 1 to the length of the list, and reads every element in the list, but doesn't do anything with it. What if you want to read every 4th entry? Increase the step in the For loop like this:

**:For(I,1,dim(L1),4)
:L1(I)**

:End:

How will this come in handy? Well, you will see in a minute here.

Files for your programs:

This is where things can get a little complicated and confusing, so I apologize in advance if this is a little less clear. Your programs want to store data, right? What a better place to store it than a list! You can name it whatever you want, you can easily read things from it, and it protects your data from other programs messing it up.

There is no way to tell whether a list exists when you start your program up. You will need to include any lists with the installation package. Let's say you want to store shapes to a file. In this example I will use Argon's file to demonstrate. Before you start writing a large program, you will want to map out the file structure, or what data will be stored where. Argon's file structure is this:

The first ten are information about the file, such as serial numbers, locking, view, etc.

Then, the structure repeats itself for each shape:

11: Shape Type

12: Location (X)

13: Location (Y)

14: Location (Z)

15: Radius

16: Direction (For faces and Cylinders)

17: Width (for Cylinders)

18: Empty

19: Empty

20: Empty

The *Empty*'s are to save space, just in case anything else is added. Imagine having to re-write the entire program and file structure when you realize you forgot something.

Adding "Empty" file data makes blocks you can come and use later.

Argon reads shapes from the list like this:

:For(I,11,LARG(5)*10+10,10)

;LARG(I) = Shape Type

;LARG(I+1) = Location (X)

;LARG(I+2) = Location (Y)

and so on.

:End

Starting from the top: The for loop uses the variable I. It goes from 11 (which is where the shape data starts) to a weird number. What does LARG(5)*10+10 do? Well, LARG(5) stores the number of shapes in the file. Each shape uses 10 elements in the list, so, thus LARG(5)*10. Shapes start at 11, and not 1, so everything's incremented by 10, (thus the +10 on the end). And the For loop counts in steps of 10 because each shape takes up 10 elements in the list. So, basically, this code starts at the first shape, and jumps to every shape to the last one.

Using with Mirage OS and Omnicalc

After creating some programs you may want to put them in Mirage OS. This is possible by putting a colon on the top line on your program. Well, here's the deal: **DON'T!**. Doing this on a calculator with an OS greater than 1.12 (which is any new Ti-83+ and all Silver Editions) will cause the calculator to crash when they try to run your program. There's a

1/10 chance that the person you give your program to will have OS 1.12 still on their calculator. This means that for every 10 people you distribute your program to, 5 of them will be after your neck, 4 of them won't care, and 1 will think your program's just great.

So the deal with putting your programs in Mirage OS is: Don't. Mirage OS 2.0 will hopefully have this bug fixed when it comes out.

You may also notice that programs such as Omnicalc and Symbolic add extra tokens to be used with BASIC programs. Once a BASIC program has these tokens in it, it cannot be transferred to another calculator. If you're writing a program just for yourself, then you can use these, but if you're writing it for distribution, then do not use these tokens.

Tips and Hints

Pressing [On] will break your program and display an "Error: Break" message. Press [2] to go to the point in your code. If you want to add a breakpoint (point where the program stops and automatically goes to, for debugging), then add a Pause in your code. When the calculator gets to the pause, press [On][2] and it will go to the point in the code.