

Programming

17

Preview of Programming.....	276
Running an Existing Program.....	278
Starting a Program Editor Session.....	280
Overview of Entering a Program.....	282
Overview of Entering a Function.....	285
Calling One Program from Another.....	287
Using Variables in a Program	288
Using Local Variables in Functions or Programs	290
String Operations	292
Conditional Tests	294
Using If, Lbl, and Goto to Control Program Flow.....	295
Using Loops to Repeat a Group of Commands.....	297
Configuring the TI-89 / TI-92 Plus.....	300
Getting Input from the User and Displaying Output	301
Creating a Custom Menu.....	303
Creating a Table or Graph.....	305
Drawing on the Graph Screen	307
Accessing Another TI-89 / TI-92 Plus, a CBL 2/CBL, or a CBR	309
Debugging Programs and Handling Errors.....	310
Example: Using Alternative Approaches	311
Assembly-Language Programs	313

Note: For details and examples of any TI-89 / TI-92 Plus program command mentioned in this chapter, refer to Appendix A.

This chapter describes how to use the TI-89 / TI-92 Plus's Program Editor to create your own programs or functions.

```
F1 F2 F3 F4 F5 F6
Tools Control I/O Var Find... Mode
:prgm(
:Prgm
:Request "Enter an integer
",n
:expr(n)→n
:0→temp
:For i,1,n,1
:  temp+i→temp
:EndFor
:Disp temp
:EndPrgm
MAIN RAD AUTO PAR
```


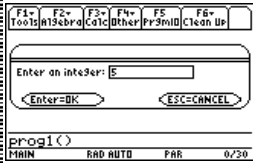
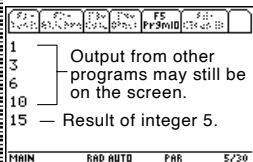

The chapter includes:

- Specific instructions on using the Program Editor itself and running an existing program.
- An overview of fundamental programming techniques such as **If...EndIf** structures and various kinds of loops.
- Reference information that categorizes the available program commands.
- Obtaining and running assembly-language programs.

Preview of Programming

Write a program that prompts the user to enter an integer, sums all integers from 1 to the entered integer, and displays the result.

Steps	TI-89 Keystrokes	TI-92 Plus Keystrokes	Display
1. Start a new program on the Program Editor.	APPS 7 3	APPS 7 3	
2. Type PROG1 (with no spaces) as the name of the new program variable.	1 P R O G 1	1 P R O G 1	
3. Display the “template” for a new program. The program name, Prgm , and EndPrgm are shown automatically. <i>After typing in an input box such as Variable, you must press twice.</i>			
4. Type the following program lines. Request "Enter an integer", <i>Displays a dialog box that prompts "Enter an integer", waits for the user to enter a value, and stores it (as a string) to variable n.</i> expr(n)→n <i>Converts the string to a numeric expression.</i> 0→temp <i>Creates a variable named temp and initializes it to 0.</i> For i,1,n,1 <i>Starts a For loop based on variable i. First time through the loop, i = 1. At end of loop, i is incremented by 1. Loop continues until i > n.</i> temp+i→temp <i>Adds current value of i to temp.</i> EndFor <i>Marks the end of the For loop.</i> Disp temp <i>Displays the final value of temp.</i>	Type the program lines as shown. Press at the end of each line.	Type the program lines as shown. Press at the end of each line.	

Steps	TI-89 Keystrokes	TI-92 Plus Keystrokes	Display
5. Go to the Home screen. Enter the program name, followed by a set of parentheses. <i>You must include () even when there are no arguments for the program.</i> <i>The program displays a dialog box with the prompt specified in the program.</i>	[HOME] [2nd] [a-lock] P R O G alpha 1 () ENTER	[2nd] [HOME] P R O G 1 () ENTER	
6. Type 5 in the displayed dialog box.	5	5	
7. Continue with the program. The Disp command displays the result on the Program I/O screen. <i>The result is the sum of the integers from 1 through 5.</i> <i>Although the Program I/O screen looks similar to the Home screen, it is for program input and output only. You cannot perform calculations on the Program I/O screen.</i>	ENTER ENTER	ENTER ENTER	
8. Leave the Program I/O screen and return to the Home screen. <i>You can also press [ESC], [2nd] [QUIT], or</i> <i>TI-89: [HOME]</i> <i>TI-92 Plus: [2nd] [HOME]</i> <i>to return to the Home screen.</i>	[F5]	[F5]	

Running an Existing Program

After a program is created (as described in the remaining sections of this chapter), you can run it from the Home screen. The program's output, if any, is displayed on the Program I/O screen, in a dialog box, or on the Graph screen.

Running a Program

Tip: Use **[2nd]** **[VAR-LINK]** to list existing PRGM variables. Highlight a variable and press **[ENTER]** to paste its name to the entry line.

Note: Arguments specify initial values for a program. Refer to page 283.

Note: The TI-89 / TI-92 Plus also checks for run-time errors that are found within the program itself. Refer to page 310.

On the Home screen:

1. Type the name of the program.

2. You must *always* type a set of parentheses after the name.

Some programs require you to pass an argument to the program.

prog1()

└ If arguments are not required

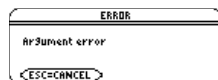
prog1(x,y)

└ If arguments are required

3. Press **[ENTER]**.

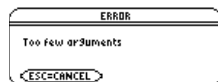
When you run a program, the TI-89 / TI-92 Plus automatically checks for errors. For example, the following message is displayed if you:

- Do not enter () after the program name.



This error message appears if you:

- Do not enter enough arguments, if required.



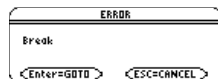
To cancel program execution if an error occurs, press **[ESC]**. You can then correct any problems and run the program again.

"Breaking" a Program

When a program is running, the BUSY indicator is displayed in the status line.

Press **[ON]** to stop program execution. A message is then displayed.

- To display the program in the Program Editor, press **[ENTER]**. The cursor appears at the command where the break occurred.
- To cancel program execution, press **[ESC]**.



Where Is the Output Displayed?

Depending on the commands in the program, the TI-89 / TI-92 Plus automatically displays information on the applicable screen.

- Most output and input commands use the Program I/O screen. (Input commands prompt the user to enter information.)
- Graph-related commands typically use the Graph screen.

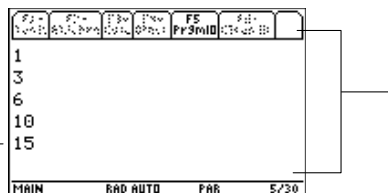
After the program stops, the TI-89 / TI-92 Plus shows the last screen that was displayed.

The Program I/O Screen

On the Program I/O screen, new output is displayed below any previous output (which may have been displayed earlier in the same program or a different program). After a full page of output, the previous output scrolls off the top of the screen.

Tip: To clear any previous output, enter the **ClrIO** command in your program. You can also execute **ClrIO** from the Home screen.

Last output —



On the Program I/O screen:

- **[F5]** toolbar is available; all others are dimmed.
- There is no entry line.

Tip: If Home screen calculations don't work after you run a program, you may be on the Program I/O screen.

When a program stops on the Program I/O screen, you need to recognize that it is *not* the Home screen (although the two screens are similar). The Program I/O screen is used only to display output or to prompt the user for input. You cannot perform calculations on this screen.

Leaving the Program I/O Screen

From the Program I/O screen:

- Press **[F5]** to toggle between the Home screen and the Program I/O screen.
— or —
- Press **[ESC]**, **[2nd]** **[QUIT]**, or
TI-89: **[HOME]**
TI-92 Plus: **[♦]** **[HOME]**
to display the Home screen.
— or —
- Display any other application screen (with **[APPS]**, **[♦]** **[Y=]**, etc.).

Starting a Program Editor Session

Each time you start the Program Editor, you can resume the current program or function (that was displayed the last time you used the Program Editor), open an existing program or function, or start a new program or function.

Starting a New Program or Function

- 1. Press [APPS] and then select 7:Program Editor.
- 2. Select 3:New.
- 3. Specify the applicable information for the new program or function.

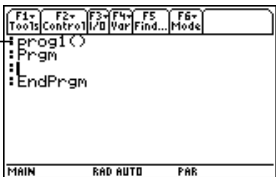


Item	Lets you:
Type	Select whether to create a new program or function. <div>1:Program 2:Function</div>
Folder	Select the folder in which the new program or function will be stored. For information about folders, refer to Chapter 5.
Variable	Type a variable name for the program or function. If you specify a variable that already exists, an error message will be displayed when you press [ENTER]. When you press [ESC] or [ENTER] to acknowledge the error, the NEW dialog box is redisplayed.

- 4. Press [ENTER] (after typing in an input box such as Variable, you must press [ENTER] twice) to display an empty “template.”

Note: A program (or function) is saved automatically as you type. You do not need to save it manually before leaving the Program Editor, starting a new program, or opening a previous one.

This is the template for a program. Functions have a similar template.



You can now use the Program Editor as described in the remaining sections of this chapter.

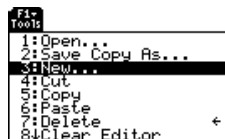
Resuming the Current Program

You can leave the Program Editor and go to another application at any time. To return to the program or function that was displayed when you left the Program Editor, press **[APPS]** 7 and select 1:Current.

Starting a New Program from the Program Editor

To leave the current program or function and start a new one:

1. Press **[F1]** and select 3:New.
2. Specify the type, folder, and variable for the new program or function.
3. Press **[ENTER]** twice.



Opening a Previous Program

You can open a previously created program or function at any time.

1. From within the Program Editor, press **[F1]** and select 1:Open.
— or —
From another application, press **[APPS]** 7 and select 2:Open.

2. Select the applicable type, folder, and variable.
3. Press **[ENTER]**.



Note: By default, Variable shows the first existing program or function in alphabetical order.

Copying a Program

In some cases, you may want to copy a program or function so that you can edit the copy while retaining the original.

1. Display the program or function you want to copy.
2. Press **[F1]** and select 2:Save Copy As.
3. Specify the folder and variable for the copy.
4. Press **[ENTER]** twice.

Note about Deleting a Program

Because all Program Editor sessions are saved automatically, you can accumulate quite a few previous programs and functions, which take up memory storage space.

To delete programs and functions, use the VAR-LINK screen (**[2nd]** [VAR-LINK]). For information about VAR-LINK, refer to Chapter 21.

Overview of Entering a Program

A program is a series of commands executed in sequential order (although some commands alter the program flow). In general, anything that can be executed from the Home screen can be included in a program. Program execution continues until it reaches the end of the program or a **Stop** command.

Entering and Editing Program Lines

Note: Use the cursor pad to scroll through the program for entering or editing commands. Use \leftarrow \rightarrow or \uparrow \downarrow to go to the top or bottom of a program, respectively.

Note: Entering a command does not execute that command. It is not executed until you run the program.

On a blank template, you can begin entering commands for your new program.

Program name, which you specify when you create a new program.

Enter your program commands between **Prgm** and **EndPrgm**.

All program lines begin with a colon.

You enter and edit program commands in the Program Editor by using the same techniques used to enter and edit text in the Text Editor. Refer to “Entering and Editing Text” in Chapter 18.

After typing each program line, press **ENTER**. This inserts a new blank line and lets you continue entering another line. A program line can be longer than one line on the screen; if so, it will wrap to the next screen line automatically.



Entering Multi-Command Lines

To enter more than one command on the same line, separate them with a colon by pressing **2nd** **[:]**.

Entering Comments

Tip: Use comments to enter information that is useful to someone reading the program code.

A comment symbol (**⦿**) lets you enter a remark in a program. When you run the program, all characters to the right of **⦿** are ignored.

	<pre>:prg1() :Prgm :⦿ Displays sum of 1 thru n :Request "Enter an integer",n :expr(n)>n:⦿ Convert to numeric expression :-----</pre>
Description of the program.	:⦿ Displays sum of 1 thru n
Description of expr .	:Request "Enter an integer",n
	:expr(n)>n:⦿ Convert to numeric expression
	:-----

To enter the comment symbol, press:

- **TI-89:** \leftarrow **1**
- **TI-92 Plus:** **2nd** **X**
— or —
- Press **F2** and select 9:⦿

Controlling the Flow of a Program

Tip: For information, refer to pages 295 and 297.

When you run a program, the program lines are executed in sequential order. However, some commands alter the program flow. For example:

- Control structures such as **If...EndIf** commands use a conditional test to decide which part of a program to execute.
- Loops commands such as **For...EndFor** repeat a group of commands.

Using Indentation

For more complex programs that use **If...EndIf** and loop structures such as **For...EndFor**, you can make the programs easier to read and understand by using indentation.

```
:If x>5 Then
: Disp "x is > 5"
:Else
: Disp "x is < or = 5"
:EndIf
```

Displaying Calculated Results

In a program, calculated results are not displayed unless you use an output command. This is an important difference between performing a calculation on the Home screen and in a program.

These calculations will not display a result in a program (although they will on the Home screen).

```
:12*6
:cos(π/4)
:solve(x^2- x- 2=0,x)
```

Tip: For a list of available output commands, refer to page 302.

Output commands such as **Disp** will display a result in a program.

```
:Disp 12*6
:Disp cos(π/4)
:Disp solve(x^2- x- 2=0,x)
```

Displaying a calculation result does not store that result. If you need to refer to a result later, store it to a variable.

```
:cos(π/4)→maximum
:Disp maximum
```

Getting Values into a Program

To input values into a program, you can:

- Require the users to store a value (with **STO►**) to the necessary variables before running the program. The program can then refer to these variables.
- Enter the values directly into the program itself.
- Include input commands that prompt the users to enter the necessary values when they run the program.
- Require the users to pass one or more values to the program when they run it.

```
:Disp 12*6
:cos(π/4)→maximum
```

```
:Input "Enter a value",i
:Request "Enter an integer",n
```

```
progl(3,5)
```

Tip: For a list of available input commands, refer to page 301.

Example of Passing Values to a Program

Note: In this example, you cannot use **circle** as the program name because it conflicts with a command name.

The following program draws a circle on the Graph screen and then draws a horizontal line across the top of the circle. Three values must be passed to the program: x and y coordinates for the circle's center and the radius r .

- When you write the program in the Program Editor:

In the () beside the program name, specify the variables that will be used to store the passed values.

Notice that the program also contains commands that set up the Graph screen.

```
:circ(x,y,r)
:Prgm
:FnOff
:ZoomStd
:ZoomSqr
:Circle x,y,r
:LineHorz y+r
:EndPrgm
```

Only **circ()** is initially displayed on the blank template; be sure to edit this line.

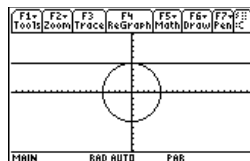
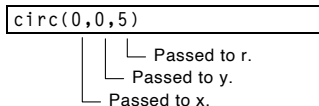
Before drawing the circle, the program turns off any selected Y= Editor functions, displays a standard viewing window, and “squares” the window.

- To run the program from the Home screen:

Note: This example assumes that the user enters values that can be displayed by the viewing window set up by **ZoomStd** and **ZoomSqr**.

The user must specify the applicable values as arguments within the ().

The arguments, in order, are passed to the program.



Overview of Entering a Function

A function created in the Program Editor is very similar to the functions and instructions that you typically use from the Home screen.

Why Create a User-Defined Function?

Note: You can create a function from the Home screen (see Chapter 5), but the Program Editor is more convenient for complex, multi-line functions.

Functions (as well as programs) are ideal for repetitive calculations or tasks. You only need to write the function once. Then you can reuse it as many times as necessary. Functions, however, have some advantages over programs.

- You can create functions that expand on the TI-89 / TI-92 Plus's built-in functions. You can then use the new functions the same as any other function.
- Functions return values that can be graphed or entered in a table; programs cannot.
- You can use a function (but not a program) within an expression. For example: $3 * \text{func1}(3)$ is valid, but not $3 * \text{prog1}(3)$.
- Because you pass arguments to a function, you can write generic functions that are not tied to specific variable names.

Differences Between Functions and Programs

This guidebook sometimes uses the word *command* as a generic reference to instructions and functions. When writing a function, however, you must differentiate between instructions and functions.

A user-defined function:

- Can use the following instructions only. Any others are invalid.

Cycle	Define	Exit
For...EndFor	Goto	If...EndIf (all forms)
Lbl	Local	Loop...EndLoop
Return	While...EndWhile	→ (STO> key)

- Can use all built-in TI-89 / TI-92 Plus functions except:

setFold	setGraph	setMode
setTable	switch	

- Can refer to any variable; however, it can store a value to a local variable only.
 - The arguments used to pass values to a function are treated as local variables automatically. If you store to any other variables, you *must* declare them as local from within the function.
- Cannot call a program as a subroutine, but it can call another user-defined function.
- Cannot define a program.
- Cannot define a global function, but it can define a local function.

Tip: For information about local variables, refer to pages 288 and 290.

Entering a Function

When you create a new function in the Program Editor, the TI-89 / TI-92 Plus displays a blank “template”.

Note: Use the cursor pad to scroll through the function for entering or editing commands.

Function name, which you specify when you create a new function.

Enter your commands between **Func** and **EndFunc**.

All function lines begin with a colon.

F1	F2	F3	F4	F5	F6
Tools	Control	1/0	Var	Find...	Mode
:Func1(x)					
:Func					
:EndFunc					
MAIN RAD AUTO PRG					

Be sure to edit this line to include any necessary arguments. Remember to use argument names in the definition that will never be used when calling the function.

If the function requires input, one or more values must be passed to the function. (A user-defined function can store to local variables only, and it cannot use instructions that prompt the user for input.)

How to Return a Value from a Function

Note: This example calculates the cube if $x \geq 0$; otherwise, it returns a 0.

There are two ways to return a value from a function:

- As the last line in the function (before **EndFunc**), calculate the value to be returned.

```
:cube(x)
:Func
:x^3
:EndFunc
```
- Use **Return**. This is useful for exiting a function and returning a value at some point other than the end of the function.

```
:cube(x)
:Func
:If x<0
:  Return 0
:x^3
:EndFunc
```

The argument x is automatically treated as a local variable. However, if the example needed another variable, the function would need to declare it as local by using the **Local** command (pages 288 and 290).

There is an implied **Return** at the end of the function. If the last line is not an expression, an error occurs.

Example of a Function

Note: Because x and y in the function are local, they are not affected by any existing x or y variable.

The following function returns the x th root of a value y ($\sqrt[x]{y}$). Two values must be passed to the function: x and y .

Function as called from the Home Screen	Function as defined in the Program Editor
<pre>3→x:125→y 4*xroot(3,125) 20</pre>	<pre>:xroot(x,y) :Func :y^(1/x) :EndFunc</pre>

Calling One Program from Another

One program can call another program as a subroutine. The subroutine can be external (a separate program) or internal (included in the main program). Subroutines are useful when a program needs to repeat the same group of commands at several different places.

Calling a Separate Program

To call a separate program, use the same syntax used to run the program from the Home screen.

```
:subtest1()  
:Prgm  
:For i,1,4,1  
:  subtest2(i,i*1000)  
:EndFor  
:EndPrgm
```

```
:subtest2(x,y)  
:Prgm  
:  Disp x,y  
:EndPrgm
```

Calling an Internal Subroutine

To define an internal subroutine, use the **Define** command with **Prgm...EndPrgm**. Because a subroutine must be defined before it can be called, it is a good practice to define subroutines at the beginning of the main program.

An internal subroutine is called and executed in the same way as a separate program.

Tip: Use the Program Editor's **[F4]** Var toolbar menu to enter the **Define** and **Prgm...EndPrgm** commands.

```
:subtest1()  
:Prgm  
:local subtest2  
:Define subtest2(x,y)=Prgm  
:  Disp x,y  
:EndPrgm  
:● Beginning of main program  
:For i,1,4,1  
:  subtest2(i,i*1000)  
:EndFor  
:EndPrgm
```

Notes about Using Subroutines

At the end of a subroutine, execution returns to the calling program. To exit a subroutine at any other time, use the **Return** command.

A subroutine cannot access local variables declared in the calling program. Likewise, the calling program cannot access local variables declared in a subroutine.

Lbl commands are local to the programs in which they are located. Therefore, a **Goto** command in the calling program cannot branch to a label in a subroutine or vice versa.

Using Variables in a Program

Programs use variables in the same general way that you use them from the Home screen. However, the “scope” of the variables affects how they are stored and accessed.

Scope of Variables

Note: For information about folders, refer to Chapter 5.

Note: If a program has local variables, a graphed function cannot access them. For example:
Local a
5→a
Graph a*cos(x)
may display an error or an unexpected result (if a is an existing variable in the current folder).

Scope	Description
System (Global) Variables	<p>Variables with reserved names that are created automatically to store data about the state of the TI-89 / TI-92 Plus. For example, Window variables (xmin, xmax, ymin, ymax, etc.) are globally available from any folder.</p> <ul style="list-style-type: none">You can always refer to these variables by using the variable name only, regardless of the current folder.A program cannot create system variables, but it can use the values and (in most cases) store new values.
Folder Variables	<p>Variables that are stored in a particular folder.</p> <ul style="list-style-type: none">If you store to a variable name only, it is stored in the current folder. For example: 5→startIf you refer to a variable name only, that variable must be in the current folder. Otherwise, it cannot be found (even if the variable exists in a different folder).To store or refer to a variable in another folder, you must specify a path name. For example: 5→class\start <div><div></div><div></div><div>Variable name</div><div>Folder name</div></div> <p>After the program stops, any folder variables created by the program still exist and still take up memory.</p>
Local Variables	<p>Temporary variables that exist only while a program is running. When the program stops, local variables are deleted automatically.</p> <ul style="list-style-type: none">To create a local variable in a program, use the Local command to declare the variable.A local variable is treated as unique even if there is an existing folder variable with the same name.Local variables are ideal for temporarily storing values that you do not want to save.

Circular Definition Errors

When evaluating a user-defined function or running a program, you can specify an argument that includes the same variable that was used to define the function or create the program. However, to avoid Circular definition errors, you must assign a value for x or i variables that are used in evaluating the function or running the program. For example:

```
x+1→x
– or –
For i,i,10,1
  Disp i
EndFor
```

Causes a **Circular definition** error message if x or i does not have a value. The error does not occur if x or i has already been assigned a value.

Variable-Related Commands

Note: The **Define**, **DelVar**, and **Local** commands are available from the Program Editor's **F4** Var toolbar menu.

Command	Description
STO▶ key	Stores a value to a variable. As on the Home screen, pressing STO▶ enters a → symbol.
Archive	Moves specified variables from RAM to user data archive memory.
BldData	Lets you create a data variable based on the graph information entered in the Y=Editor, Window Editor, etc.
CopyVar	Copies the contents of a variable.
Define	Defines a program (subroutine) or function variable within a program.
DelFold	Deletes a folder. All variables in that folder must be deleted first.
DelVar	Deletes a variable.
getFold	Returns the name of the current folder.
getType	Returns a string that indicates the data type (EXPR, LIST, etc.) of a variable.
Local	Declares one or more variables as local variables.
Lock	Locks a variable so that it cannot be accidentally changed or deleted without first being unlocked.
MoveVar	Moves a variable from one folder to another.
NewData	Creates a data variable whose columns consist of a series of specified lists.
NewFold	Creates a new folder.
NewPic	Creates a picture variable based on a matrix.
Rename	Renames a variable.
Unarchiv	Moves specified variables from user data archive memory to RAM.
Unlock	Unlocks a locked variable.

Using Local Variables in Functions or Programs

A local variable is a temporary variable that exists only while a user-defined function is being evaluated or a user-defined program is running.

Example of a Local Variable

Tip: As often as possible, use local variables for any variable that is used only within a program and does not need to be stored after the program stops.

The following program segment shows a **For...EndFor** loop (which is discussed later in this chapter). The variable *i* is the loop counter. In most cases, the variable *i* is used only while the program is running.

```
Declares variable i as local. _____ :Local i
                                         :For i,0,5,1
                                         : Disp i
                                         :EndFor
                                         :Disp i
```

If you declare variable *i* as local, it is deleted automatically when the program stops so that it does not use up memory.

What Causes an Undefined Variable Error Message?

An Undefined variable error message displays when you evaluate a user-defined function or run a user-defined program that references a local variable that is not initialized (assigned a value).

This example is a multi-statement function, rather than a program. Line breaks are shown here, but you would type the text in the entry line as one continuous line, such as: Define fact(n)=Func:Local... where the ellipsis indicates the entry line text continues off-screen.

For example:

```
Define fact(n)=Func:
Local m: _____ Local variable m is not assigned an
While n>1:          initial value.
  n*m→m: n-1→n:
EndWhile:
Return m:
EndFunc
```

In the example above, the local variable *m* exists independently of any variable *m* that exists outside of the function.

You Must Initialize Local Variables

All local variables must be assigned an initial value before they are referenced.

```
Define fact(n)=Func:
Local m: 1→m: _____ 1 is stored as the initial value for m.
While n>1:
  n*m→m: n-1→n:
EndWhile:
Return m:
EndFunc
```

The TI-89 / TI-92 Plus cannot use a local variable to perform symbolic calculations.

To Perform Symbolic Calculations

If you want a function or program to perform symbolic calculations, you must use a global variable instead of a local. However, you must be certain that the global variable does not already exist outside of the program. The following methods can help.

- Refer to a global variable name, typically with two or more characters, that is not likely to exist outside of the function or program.
- Include **DelVar** within the function or program to delete the global variable, if it exists, before referring to it. (**DelVar** does not delete locked or archived variables.)

Strings are used to enter and display text characters. You can type a string directly, or you can store a string to a variable.

How Strings Are Used

A string is a sequence of characters enclosed in "quotes". In programming, strings allow the program to display information or prompt the user to perform some action. For example:

```
Disp "The result is",answer
— or —
Input "Enter the angle in degrees",ang1
— or —
"Enter the angle in degrees"→str1
Input str1,ang1
```

Some input commands (such as **InputStr**) automatically store user input as a string and do not require the user to enter quotation marks.

A string cannot be evaluated mathematically, even if it appears to be a numeric expression. For example, the string "61" represents the characters "6" and "1", not the number 61.

Although you cannot use a string such as "61" or "2x+4" in a calculation, you can convert a string into a numeric expression by using the **expr** command.

String Commands

Note: See Appendix A for syntax for all TI-89 / TI-92 Plus commands and functions.

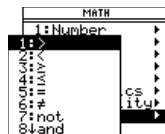
Command	Description
#	Converts a string into a variable name. This is called indirection.
&	Appends (concatenates) two strings into one string.
char	Returns the character that corresponds to a specified character code. This is the opposite of the ord command.
dim	Returns the number of characters in a string.
expr	Converts a string into an expression and executes that expression. This is the opposite of the string command.
	Important: Some user input commands store the entered value as a string. Before you can perform a mathematical operation on that value, you must convert it to a numeric expression.
format	Returns an expression as a character string based on the format template (fixed, scientific, engineering, etc.)
inString	Searches a string to see if it contains a specified substring. If so, inString returns the character position where the first occurrence of the substring begins.
left	Returns a specified number of characters from the left side (beginning) of a string.
mid	Returns a specified number of characters from any position within a string.
ord	Returns the character code of the first character within a string. This is the opposite of the char command.
right	Returns a specified number of characters from the right side (end) of a string.
rotate	Rotates the characters in a string. The default is -1 (rotate right one character).
shift	Shifts the characters in a string and replaces them with spaces. The default is -1 (shift right one character and replace with one space). Examples: <code>shift("abcde",2)⇒"cde "</code> and <code>shift("abcde")⇒" abcd"</code>
string	Converts a numeric expression into a string. This is the opposite of the expr command.

Conditional Tests

Conditional tests let programs make decisions. For example, depending on whether a test is true or false, a program can decide which of two actions to perform. Conditional tests are used with control structures such as **If...EndIf** and loops such as **While...EndWhile** (described later in this chapter).

Entering a Test Operator

- Type the operator directly from the keyboard.
— or —
- Press **[2nd]** **[MATH]** and select 8:Test. Then select the operator from the menu.
— or —
- Display the built-in functions.



Press: **TI-89:** **[CATALOG]**

TI-92 Plus: **[2nd]** **[CATALOG]**

The test operators are listed near the bottom of the **[F2]** Built-in menu.

Relational Tests

Relational operators let you define a conditional test that compares two values. The values can be numbers, expressions, lists, or matrices (but they must match in type and dimension).

Tip: From the keyboard, you can type:
 \geq for \geq
 \leq for \leq
 \neq for \neq
(To get the / character, press **[÷]**.)

Operator	True if:	Example
$>$	Greater than	$a > 8$
$<$	Less than	$a < 0$
\geq	Greater than or equal to	$a + b \geq 100$
\leq	Less than or equal to	$a + 6 \leq b + 1$
$=$	Equal	$\text{list1} = \text{list2}$
\neq	Not equal to	$\text{mat1} \neq \text{mat2}$

Boolean Tests

Boolean operators let you combine the results of two separate tests.

Operator	True if:	Example
and	Both tests are true	$a > 0$ and $a \leq 10$
or	At least one test is true	$a \leq 0$ or $b + c > 10$
xor	One test is true and the other is false	$a + 6 < b + 1$ xor $c < d$

The Not Function

The **not** function changes the result of a test from true to false and vice versa. For example:

$\text{not } x > 2$ is true if $x \leq 2$
false if $x > 2$

Note: If you use **not** from the Home screen, it is shown as \sim in the history area. For example, $\text{not } x > 2$ is shown as $\sim(x > 2)$.

Using If, Lbl, and Goto to Control Program Flow

An **If...EndIf** structure uses a conditional test to decide whether or not to execute one or more commands. **Lbl** (label) and **Goto** commands can also be used to branch (or jump) from one place to another in a program.

F2 Control Toolbar Menu

To enter **If...EndIf** structures, use the Program Editor's **F2** Control toolbar menu.



The **If** command is available directly from the **F2** menu.

To see a submenu that lists other **If** structures, select 2:If...Then.



When you select a structure such as **If...Then...EndIf**, a template is inserted at the cursor location.

```
:If | Then  
  
:EndIf  
The cursor is positioned so  
that you can enter a  
conditional test.
```

If Command

To execute only one command if a conditional test is true, use the general form:

Tip: Use indentation to make your programs easier to read and understand.

Executed only if $x > 5$; otherwise, skipped.	:If $x > 5$
Always displays the value of x .	: Disp "x is greater than 5"
	:Disp x

In this example, you must store a value to x before executing the **If** command.

If...Then...EndIf Structures

To execute multiple commands if a conditional test is true, use the structure:

Note: **EndIf** marks the end of the **Then** block that is executed if the condition is true.

Executed only if $x > 5$.	:If $x > 5$ Then
	: Disp "x is greater than 5"
	: 2* $x > x$
	:EndIf
Displays value of:	:Disp x
• 2x if $x > 5$.	
• x if $x \leq 5$.	

If...Then...Else... EndIf Structures

To execute one group of commands if a conditional test is true and a different group if the condition is false, use this structure:

Executed only if $x > 5$.		:If $x > 5$ Then
		: Disp "x is greater than 5"
		: 2* $x \rightarrow x$
		:Else
Executed only if $x \leq 5$.		: Disp "x is less than or equal to 5"
		: 5* $x \rightarrow x$
		:EndIf
Displays value of:		:Disp x
• 2x if $x > 5$.		
• 5x if $x \leq 5$.		

If...Then...Elseif... EndIf Structures

A more complex form of the **If** command lets you test a series of conditions. Suppose your program prompts the user for a number that corresponds to one of four options. To test for each option (If Choice=1, If Choice = 2, etc.), use the **If...Then...Elseif...EndIf** structure.

Refer to Appendix A for more information and an example.

Lbl and Goto Commands

You can also control the flow of your program by using **Lbl** (label) and **Goto** commands.

Use the **Lbl** command to label (assign a name to) a particular location in the program.

Lbl *labelName*

└─ name to assign to this location (use the same naming convention as a variable name)

You can then use the **Goto** command at any point in the program to branch to the location that corresponds to the specified label.

Goto *labelName*

└─ specifies which **Lbl** command to branch to

Because a **Goto** command is unconditional (it always branches to the specified label), it is often used with an **If** command so that you can specify a conditional test. For example:

		:If $x > 5$
If $x > 5$, branches directly to		: Goto GT5
label GT5.		:Disp x
		:-----
For this example, the program		:-----
must include commands (such		:Lbl GT5
as Stop) that prevent Lbl GT5		:Disp "The number was > 5"
from being executed if $x \leq 5$.		

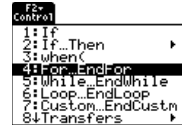
Using Loops to Repeat a Group of Commands

To repeat the same group of commands successively, use a loop. Several types of loops are available. Each type gives you a different way to exit the loop, based on a conditional test.

F2 Control Toolbar Menu

Note: A loop command marks the start of the loop. The corresponding **End** command marks the end of the loop.

To enter most of the loop-related commands, use the Program Editor's **F2** Control toolbar menu.



When you select a loop, the loop command and its corresponding **End** command are inserted at the cursor location.

:For |
:EndFor
If the loop requires arguments, the cursor is positioned after the command.

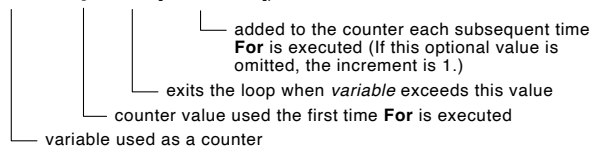
You can then begin entering the commands that will be executed in the loop.

For...EndFor Loops

A **For...EndFor** loop uses a counter to control the number of times the loop is repeated. The syntax of the **For** command is:

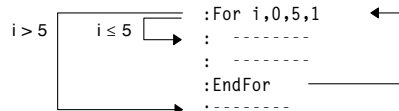
For(variable, begin, end [, increment])

Note: The ending value can be less than the beginning value, but the increment must be negative.



When **For** is executed, the *variable* value is compared to the *end* value. If *variable* does not exceed *end*, the loop is executed; otherwise, program control jumps to the command following **EndFor**.

Note: The **For** command automatically increments the counter variable so that the program can exit the loop after a certain number of repetitions.



At the end of the loop (**EndFor**), program control jumps back to the **For** command, where *variable* is incremented and compared to *end*.

For example:

Tip: You can declare the counter variable as local (pages 288 and 290) if it does not need to be saved after the program stops.

		<hr/>
		:For i,0,5,1
Displays 0, 1, 2, 3, 4, and 5.	——	: Disp i
		:EndFor
Displays 6. When variable	——	:Disp i
increments to 6, the loop		<hr/>
is not executed.		

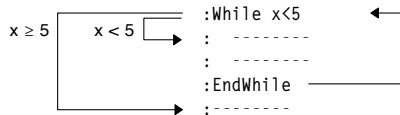
While...EndWhile Loops

A **While...EndWhile** loop repeats a block of commands as long as a specified condition is true. The syntax of the **While** command is:

While *condition*

When **While** is executed, the condition is evaluated. If *condition* is true, the loop is executed; otherwise, program control jumps to the command following **EndWhile**.

Note: The **While** command does not automatically change the condition. You must include commands that allow the program to exit the loop.



At the end of the loop (**EndWhile**), program control jumps back to the **While** command, where *condition* is re-evaluated.

To execute the loop the first time, the *condition* must initially be true.

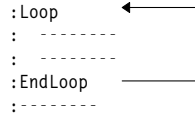
- Any variables referenced in the *condition* must be set before the **While** command. (You can build the values into the program or prompt the user to enter the values.)
- The loop must contain commands that change the values in the *condition*, eventually causing it to be false. Otherwise, the *condition* is always true and the program cannot exit the loop (called an infinite loop).

For example:

Initially sets x.	——	<hr/>
		:0>x
		:While x<5
Displays 0, 1, 2, 3, and 4.	——	: Disp x
Increments x.	——	: x+1>x
		:EndWhile
Displays 5. When x	——	:Disp x
increments to 5, the loop is		<hr/>
not executed.		

Loop...EndLoop Loops

A **Loop...EndLoop** creates an infinite loop, which is repeated endlessly. The **Loop** command does not have any arguments.



Typically, the loop contains commands that let the program exit from the loop. Commonly used commands are: **If**, **Exit**, **Goto**, and **Lbl** (label). For example:

Note: The **Exit** command exits from the current loop.

```
          :0> x
          :Loop
          : Disp x
          : x+1> x
An If command checks the condition.  : If x>5
          : Exit
          :EndLoop
Exits the loop and jumps to here when x increments to 6. :Disp x
```

In this example, the **If** command can be anywhere in the loop.

When the If command is:	The loop is:
At the beginning of the loop	Executed only if the condition is true.
At the end of the loop	Executed at least once and repeated only if the condition is true.

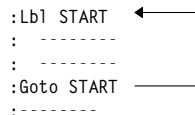
The **If** command could also use a **Goto** command to transfer program control to a specified **Lbl** (label) command.

Repeating a Loop Immediately

The **Cycle** command immediately transfers program control to the next iteration of a loop (before the current iteration is complete). This command works with **For...EndFor**, **While...EndWhile**, and **Loop...EndLoop**.

Lbl and Goto Loops

Although the **Lbl** (label) and **Goto** commands are not strictly loop commands, they can be used to create an infinite loop. For example:



As with **Loop...EndLoop**, the loop should contain commands that let the program exit from the loop.

Configuring the TI-89 / TI-92 Plus

Programs can contain commands that change the configuration of the TI-89 / TI-92 Plus. Because mode changes are particularly useful, the Program Editor's **Mode** toolbar menu makes it easy to enter the correct syntax for the **setMode** command.

Configuration Commands

Note: The parameter/mode strings used in the `setMode()`, `getMode()`, `setGraph()`, and `setTable()` functions do not translate into other languages when used in a program. See Appendix D.

Command	Description
<code>getConfig</code>	Returns a list of calculator characteristics.
<code>getFold</code>	Returns the name of the current folder.
<code>getMode</code>	Returns the current setting for a specified mode.
<code>getUnits</code>	Returns a list of default units.
<code>setFold</code>	Sets the current folder.
<code>setGraph</code>	Sets a specified graph format (Coordinates, Graph Order, etc.).
<code>setMode</code>	Sets any mode except Current Folder.
<code>setTable</code>	Sets a specified table setup parameter (<code>tblStart</code> , <code>Atbl</code> , etc.).
<code>setUnits</code>	Sets default units for displayed results.
<code>switch</code>	Sets the active window in a split screen, or returns the number of the active window.

Entering the SetMode Command

Note: The Mode menu does not let you set the Current Folder mode. To set this mode, use the **setFold** command.

In the Program Editor:

1. Position the cursor where you want to insert the **setMode** command.

2. Press:

TI-89: [2nd] [F6]

TI-92 Plus: [F6]

to display a list of modes.



3. Select a mode to display a menu of its valid settings.
4. Select a setting.


The correct syntax is inserted into your program.

```
:setMode("Graph","FUNCTION")
```

Getting Input from the User and Displaying Output

Although values can be built into a program (or stored to variables in advance), a program can prompt the user to enter information while the program is running. Likewise, a program can display information such as the result of a calculation.

I/O Toolbar Menu

To enter most of the commonly used input/output commands, use the Program Editor's  I/O toolbar menu.



To see a submenu that lists additional commands, select 1:Dialog.



Input Commands

Command	Description
getKey	Returns the key code of the next key pressed. See Appendix B for a listing of key codes.
Input	Prompts the user to enter an expression. The expression is treated according to how it is entered. For example: <ul style="list-style-type: none">• A numeric expression is treated as an expression.• An expression enclosed in "quotes" is treated as a string. Input can also display the Graph screen and let the user update the variables xc and yc (rc and θc in polar mode) by positioning the graph cursor.
InputStr	Prompts the user to enter an expression. The expression is always treated as a string; the user does not need to enclose the expression in "quotes".
PopUp	Displays a pop-up menu box and lets the user select an item.
Prompt	Prompts the user to enter a series of expressions. As with Input , each expression is treated according to how it is entered.
Request	Displays a dialog box that prompts the user to enter an expression. Request always treats the entered expression as a string.

Tip: String input cannot be used in a calculation. To convert a string to a numeric expression, use the **expr** command.

Output Commands

Note: In a program, simply performing a calculation does not display the result. You must use an output command.

Tip: After **Disp** and **Output**, the program immediately continues. You may want to add a **Pause** command.

Command	Description
ClrIO	Clears the Program I/O screen.
Disp	Displays an expression or string on the Program I/O screen. Disp can also display the current contents of the Program I/O screen without displaying additional information.
DispG	Displays the current contents of the Graph screen.
DispHome	Displays the current contents of the Home screen.
DispTbl	Displays the current contents of the Table screen.
Output	Displays an expression or string starting at specified coordinates on the Program I/O screen.
Format	Formats the way in which numeric information is displayed.
Pause	Suspends program execution until the user presses [ENTER] . Optionally, you can display an expression during the pause. A pause lets users read your output and decide when they are ready to continue.
Text	Displays a dialog box that contains a specified character string.

Graphical User Interface Commands

Tip: When you run a program that sets up a custom toolbar, that toolbar is still available even after the program has stopped.

Note: **Request** and **Text** are stand-alone commands that can also be used outside of a dialog box or toolbar program block.

Command	Description
Dialog... EndDlg	Defines a program block (consisting of Title , Request , etc., commands) that displays a dialog box.
Toolbar... EndTbar	Defines a program block (consisting of Title , Item , etc., commands) that replaces the toolbar menus. The redefined toolbar is in effect only while the program is running and only until the user selects an item. Then the original toolbar is redisplayed.
CustmOn... CustmOff	Activates or removes a custom toolbar.
Custom... EndCustm	Defines a program block that displays a custom toolbar when the user presses [2nd][CUSTOM] . That toolbar remains in effect until the user presses [2nd][CUSTOM] again or changes applications.
DropDown	Displays a drop-down menu within a dialog box.
Item	Displays a menu item for a redefined toolbar.
Request	Creates an input box within a dialog box.
Text	Displays a character string within a dialog box.
Title	Displays the title of a dialog box or a menu title within a toolbar.


Creating a Custom Menu

The TI-89 / TI-92 Plus custom menu feature lets you create your own toolbar menu. A custom menu can contain any available function, instruction, or set of characters. The TI-89 / TI-92 Plus has a default custom menu that you can modify or redefine.

Turning the Custom Menu On and Off

Note: When the custom menu is turned on, it replaces the normal toolbar menu. Unless a different custom menu has been created, the default custom menu is displayed.

When you create a custom menu, you can let the user turn it on and off manually, or you can let a program turn it on and off automatically.

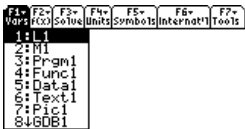
To:	Do this:
Turn on the custom menu	<p>From the Home screen or any other application:</p> <ul style="list-style-type: none">Press [2nd] [CUSTOM]. <p>From the Home screen or a program:</p> <ul style="list-style-type: none">Execute the <code>CustmOn</code> command.
Turn off the custom menu	<p>From any application:</p> <ul style="list-style-type: none">Press [2nd] [CUSTOM] again.— or —Go to a different application. <p>Using the default custom menu on the Home screen:</p> <ol style="list-style-type: none">Select the Tools menu: TI-89: [2nd] [F7] TI-92 Plus: [F7] Then select 3: CustmOff.  This pastes <code>CustmOff</code> in the entry line.Press [ENTER]. <p>You can also use <code>CustmOff</code> in a program.</p>

Defining a Custom Menu

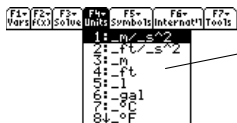
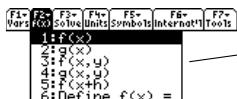
Note: When the user selects a menu item, the text defined by that **Item** command is pasted to the current cursor location.

To create a custom menu, use the following general structure.

```
:Custom
: Title title of F1 menu
:   Item item 1
:   Item item 2
:   . . .
: Title title of F2 menu
:   . . .
: Title title of F3 menu
:   . . .
:EndCustm
```



Note: The following may be slightly different than the default custom menu on your calculator.



Note: See how " _oC " and " _oF " display as $^{\circ}\text{C}$ and $^{\circ}\text{F}$ in the menu. Similarly, see the international accented characters.

For example:

```
:Custom
:Title "Vars"
:Item "L1":Item "M1":Item "Prgm1":Item "Func1":Item "Data1"
:Item "Text1":Item "Pic1":Item "GDB1":Item "Str1"

:Title "f(x)"
:Item "f(x)":Item "g(x)":Item "f(x,y)":Item "g(x,y)"
:Item "f(x+h)":Item "Define f(x) ="

:Title "Solve"
:Item "Solve(":Item " and ":Item "{x,y}"
:Item "Solve( and,{x,y})"

:Title "Units"
:Item "_m/_s^2":Item "_ft/_s^2":Item "_m":Item "_ft":Item "_l"
:Item "_gal":Item "_\oC":Item "_\oF":Item "_kph":Item "_mph"

:Title "Symbols"
:Item "#":Item "\beta\":Item "?":Item "~":Item "&"

:Title "Internat'l"
:Item "\e\ ":Item "\e'\ ":Item "\e^\ ":Item "\a\ "
:Item "\u\ ":Item "\u'\ ":Item "\o^\ ":Item "\c,\ ":Item "\u..\ "

:Title "Tools"
:Item "ClrHome":Item "NewProb":Item "CustmOff"

:EndCustm
:CustmOn
```

Note: This inserts all the commands on a single line. You do **not** need to split them into separate lines.

To modify the default custom menu, use 3:Restore custom default (as described below) to get the commands for the default menu. Copy those commands, use the Program Editor to create a new program, and paste them into the blank program. Then modify the commands as necessary.

You can create and use only one custom menu at a time. If you need more, write a separate program for each custom menu. Then run the program for the menu you need.

Restoring the Default Custom Menu

To restore the default:

- From the Home screen's normal menu (not the custom menu), select Clean Up:
TI-89: [2nd] [F6]
TI-92 Plus: [F6]

- Select 3:Restore custom default.

This pastes the commands used to create the default menu into the entry line.



- Press [ENTER] to execute the commands and restore the default.

When you restore the default, any previous custom menu is erased. If the previous menu was created with a program, you can run the program again if you want to reuse the menu later.

Creating a Table or Graph

To create a table or a graph based on one or more functions or equations, use the commands listed in this section.

Table Commands

Command	Description
DispTbl	Displays the current contents of the Table screen.
setTable	Sets the Graph \leftrightarrow Table or Independent table parameters. (To set the other two table parameters, you can store the applicable values to the tblStart and Δ tbl system variables.)
Table	Builds and displays a table based on one or more expressions or functions.

Graphing Commands

Command	Description
ClrGraph	Erases any functions or expressions that were graphed with the Graph command.
Define	Creates a user-defined function.
DispG	Displays the current contents of the Graph screen.
FnOff	Deselects all (or only specified) Y= functions.
FnOn	Selects all (or only specified) Y= functions.
Graph	Graphs one or more specified expressions, using the current graphing mode.
Input	Displays the Graph screen and lets the user update the variables xc and yc (rc and θ c in polar mode) by positioning the graph cursor.
NewPlot	Creates a new stat plot definition.
PlotsOff	Deselects all (or only specified) stat data plots.
PlotsOn	Selects all (or only specified) stat data plots.
setGraph	Changes settings for the various graph formats (Coordinates, Graph Order, etc.).
setMode	Sets the Graph mode, as well as other modes.
Style	Sets the display style for a function.
Trace	Lets a program trace a graph.
ZoomBox – to – ZoomTrig	Perform all of the Zoom operations that are available from the F2 toolbar menu on the Y= Editor, Window Editor, and Graph screen.

Note: For more information about using **setMode**, refer to page 300.

Graph Picture and Database Commands

Note: For information about graph pictures and databases, also refer to Chapter 12.

Command	Description
AndPic	Displays the Graph screen and superimposes a stored graph picture by using AND logic.
CyclePic	Animates a series of stored graph pictures.
NewPic	Creates a graph picture variable based on a matrix.
RclGDB	Restores all settings stored in a graph database.
RclPic	Displays the Graph screen and superimposes a stored graph picture by using OR logic.
RplcPic	Clears the Graph screen and displays a stored graph picture.
StoGDB	Stores the current graph settings to a graph database variable.
StoPic	Copies the Graph screen (or a specified rectangular portion) to a graph picture variable.
XorPic	Displays the Graph screen and superimposes a stored graph picture by using XOR logic.

Drawing on the Graph Screen

To create a drawing object on the Graph screen, use the commands listed in this section.

Pixel vs. Point Coordinates

When drawing an object, you can use either of two coordinate systems to specify a location on the screen.

- **Pixel coordinates** — Refer to the pixels that physically make up the screen. These are independent of the viewing window because the screen is always:
TI-89: 159 (0 to 158) pixels wide and 77 (0 to 76) pixels tall.
TI-92 Plus: 239 (0 to 238) pixels wide and 103 (0 to 102) pixels tall.
- **Point coordinates** — Refer to the coordinates in effect for the current viewing window (as defined in the Window Editor).

Tip: For information about pixel coordinates in split screens, refer to Chapter 14.

0,0	TI-89: 158,0 TI-92 Plus: 238,0
TI-89: 0,76 TI-92 Plus: 0,102	TI-89: 158,76 TI-92 Plus: 238,102

Pixel coordinates
(independent of viewing window)

-10,10	10,10
-10,-10	10,-10

Point coordinates
(for standard viewing window)

Note: Pixel commands start with Pxl, such as **PxlChg**.

Many drawing commands have two forms: one for pixel coordinates and one for point coordinates.

Erasing Drawn Objects

Command	Description
ClrDraw	Erases all drawn objects from the Graph screen.

Drawing a Point or Pixel

Command	Description
PtChg or PxlChg	Toggles (inverts) a pixel at the specified coordinates. PtChg , which uses point coordinates, affects the pixel closest to the specified point. If the pixel is off, it is turned on. If the pixel is on, it is turned off.
PtOff or PxlOff	Turns off (erases) a pixel at the specified coordinates. PtOff , which uses point coordinates, affects the pixel closest to the specified point.
PtOn or PxlOn	Turns on (displays) a pixel at the specified coordinates. PtOn , which uses point coordinates, affects the pixel closest to the specified point.
PtTest or PxlTest	Returns true or false to indicate if the specified coordinate is on or off, respectively.
PtText or PxlText	Displays a character string at the specified coordinates.

Drawing Lines and Circles

Command	Description
Circle or PxlCrcI	Draws, erases, or inverts a circle with a specified center and radius.
DrawSlp	Draws a line with a specified slope through a specified point.
Line or PxlLine	Draws, erases, or inverts a line between two sets of coordinates.
LineHorz or PxlHorz	Draws, erases, or inverts a horizontal line at a specified row coordinate.
LineTan	Draws a tangent line for a specified expression at a specified point. (This draws the tangent line only, not the expression.)
LineVert or PxlVert	Draws, erases, or inverts a vertical line at a specified column coordinate.


Drawing Expressions


Command	Description
DrawFunc	Draws a specified expression.
DrawInv	Draws the inverse of a specified expression.
DrawParm	Draws a parametric equation using specified expressions as its x and y components.
DrawPol	Draws a specified polar expression.
DrwCtour	Draws contours in 3D graphing mode.
Shade	Draws two expressions and shades the areas where <i>expression1</i> < <i>expression2</i> .

Accessing Another TI-89/TI-92 Plus, a CBL 2/CBL, or a CBR

If you link two TI-89 / TI-92 Plus calculators (described in Chapter 22), programs on both units can transmit variables between them. If you link a TI-89 / TI-92 Plus to a Calculator-Based Laboratory™ (CBL 2™/CBL™) or a Calculator-Based Ranger™ (CBR™), a program on the TI-89 / TI-92 Plus can access the CBL 2/CBL or CBR.

I/O Toolbar Menu

Use the Program Editor's  I/O toolbar menu to enter the commands in this section.

1. Press  and select 8:Link.
2. Select a command.



Accessing Another TI-89 / TI-92 Plus

When two TI-89 / TI-92 Plus calculators are linked, one acts as a receiving unit and the other as a sending unit.

Note: For a sample program that synchronizes the receiving and sending units so that **GetCalc** and **SendCalc** are executed in the proper sequence, refer to "Transmitting Variables under Program Control" in Chapter 22.

Command	Description
GetCalc	Executed on the receiving unit. Sets up the unit to receive a variable via the I/O port. <ul style="list-style-type: none">• After the receiving unit executes GetCalc, the sending unit must execute SendCalc.• After the sending unit executes SendCalc, the sent variable is stored on the receiving unit (in the variable name specified by GetCalc).
SendCalc	Executed on the sending unit. Sends a variable to the receiving unit via the I/O port. <ul style="list-style-type: none">• Before the sending unit executes SendCalc, the receiving unit must execute GetCalc.
SendChat	Executed on the sending unit as a general alternative to SendCalc . Useful if the receiving unit is a TI-92 (or for a generic "chat" program that allows either a TI-92 or TI-92 Plus to be used).

Accessing a CBL 2/CBL or CBR

For additional information, refer to the manual that comes with the CBL 2/CBL or CBR unit.

Command	Description
Get	Gets a variable from an attached CBL 2/CBL or CBR and stores it in the TI-89 / TI-92 Plus.
Send	Sends a list variable from the TI-89 / TI-92 Plus to the CBL 2/CBL or CBR.

Debugging Programs and Handling Errors

After you write a program, you can use several techniques to find and correct errors. You can also build an error-handling command into the program itself.

Run-Time Errors

The first step in debugging your program is to run it. The TI-89 / TI-92 Plus automatically checks each executed command for syntax errors. If there is an error, a message indicates the nature of the error.

- To display the program in the Program Editor, press **[ENTER]**. The cursor appears in the approximate area of the error.



- To cancel program execution and return to the Home screen, press **[ESC]**.

If your program allows the user to select from several options, be sure to run the program and test each option.

Debugging Techniques

Run-time error messages can locate syntax errors but not errors in program logic. The following techniques may be useful.

- During testing, do not use local variables so that you can check the variable values after the program stops. When the program is debugged, declare the applicable variables as local.
- Within a program, temporarily insert **Disp** and **Pause** commands to display the values of critical variables.
 - **Disp** and **Pause** cannot be used in a user-defined function. To temporarily change the function into a program, change **Func** and **EndFunc** to **Prgm** and **EndPrgm**. Use **Disp** and **Pause** to debug the program. Then remove **Disp** and **Pause** and change the program back into a function.
- To confirm that a loop is executed the correct number of times, display the counter variable or the values in the conditional test.
- To confirm that a subroutine is executed, display messages such as "Entering subroutine" and "Exiting subroutine" at the beginning and end of the subroutine.

Error-Handling Commands

Command	Description
Try...EndTry	Defines a program block that lets the program execute a command and, if necessary, recover from an error generated by that command.
ClrErr	Clears the error status and sets the error number in system variable Errornum to zero.
PassErr	Passes an error to the next level of the Try...EndTry block.

Example: Using Alternative Approaches

The preview at the beginning of this chapter shows a program that prompts the user to enter an integer, sums all integers from 1 to the entered integer, and displays the result. This section gives several approaches that you can use to achieve the same goal.

Example 1

This example is the program given in the preview at the beginning of the chapter. Refer to the preview for detailed information.

	:prog1()
	:Prgm
Prompts for input in a dialog box.	:Request "Enter an integer",n
Converts string entered with Request to an expression.	:expr(n)→n
	:0→temp
Loop calculation.	:For i,1,n,1
	: temp+i→temp
	:EndFor
Displays output on Program I/O screen.	:Disp temp
	:EndPrgm

Example 2

This example uses **InputStr** for input, a **While...EndWhile** loop to calculate the result, and **Text** to display the result.

	:prog2()
	:Prgm
Prompts for input on Program I/O screen.	:InputStr "Enter an integer",n
Converts string entered with InputStr to an expression.	:expr(n)→n
	:0→temp:1→i
Loop calculation.	:While i≤n
	: temp+i→temp
	: i+1→i
	:EndWhile
Displays output in a dialog box.	:Text "The answer is "&string(temp)
	:EndPrgm

Tip: For ≤, type $\boxed{\leq}$ (zero).

For &, press:

TI-89: $\boxed{\&}$ (times)

TI-92 Plus: $\boxed{2nd}$ H

Example 3

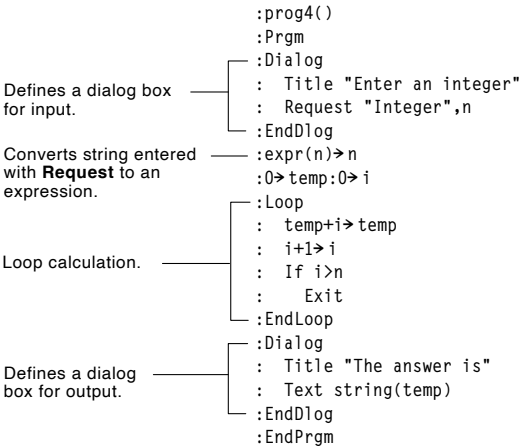
This example uses **Prompt** for input, **Lbl** and **Goto** to create a loop, and **Disp** to display the result.

	:prog3()
	:Prgm
Prompts for input on Program I/O screen.	:Prompt n
	:0→temp:1→i
Loop calculation.	:Lbl top
	: temp+i→temp
	: i+1→i
	: If i≤n
	: Goto top
Displays output on Program I/O screen.	:Disp temp
	:EndPrgm

Note: Because **Prompt** returns *n* as a number, you do not need to use **expr** to convert *n*.

Example 4

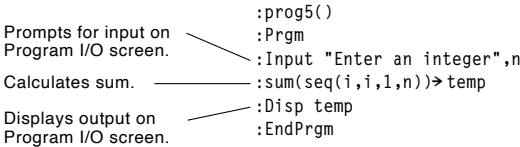
This example uses **Dialog...EndDlog** to create dialog boxes for input and output. It uses **Loop...EndLoop** to calculate the result.



Example 5

This example uses the TI-89 / TI-92 Plus built-in functions to calculate the result without using a loop.

Note: Because **Input** returns *n* as a number, you do not need to use **expr** to convert *n*.



Function	Used in this example to:
seq	Generate the sequence of integers from 1 to n. <div><pre>seq(expression, var, low, high [,step])</pre><div><div>expression used to generate the sequence</div><div>variable that will be incremented</div><div><div>initial and final values of var</div><div>increment for var ; if omitted, uses 1.</div></div></div></div>
sum	Sum the integers in the list generated by seq .

Assembly-Language Programs

You can run programs written for the TI-89 / TI-92 Plus in assembly language. Typically, assembly-language programs run much faster and provide greater control than the keystroke programs that you write with the built-in Program Editor.

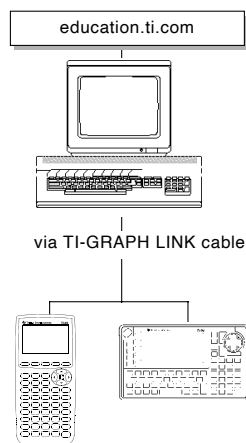
Where to Get Assembly-Language Programs

Assembly-language programs, as well as keystroke programs, are available on the Texas Instruments web site at:

education.ti.com

The programs available from this site provide additional functions or features that are not built into the TI-89 / TI-92 Plus. Check the Texas Instruments web site for up-to-date information.

After downloading a program from the web to your computer, use a TI-GRAPH LINK™ computer-to-calculator cable to send the program to your TI-89 / TI-92 Plus. Refer to the manual that comes with the TI-GRAPH LINK cable.



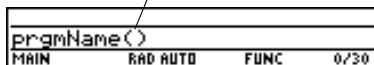
Note about TI-GRAPH LINK

If you have a TI-GRAPH LINK computer-to-calculator cable and software for the TI-92, be aware that the TI-92 TI-GRAPH LINK *software* is not compatible with the TI-89 / TI-92 Plus. The cable, however, works with both units. For information about obtaining TI™ Connect or TI-GRAPH LINK software or a TI-GRAPH LINK cable, check the Texas Instruments web site at **education.ti.com** or contact Texas Instruments as described in Appendix C of this guidebook.

Running an Assembly-Language Program

After a TI-89 / TI-92 Plus assembly-language program is stored on your unit, you can run the program from the Home screen just as you would any other program.

Tip: If the program is not in the current folder, be sure to specify the pathname.



If the program requires one or more arguments, type them within the (). Refer to the program's documentation to find out about required arguments.

You can call an assembly-language program from another program as a subroutine, delete it, or use it the same as any other program.

Shortcuts to Run a Program

Note: The programs must be stored in the MAIN folder. Also, you cannot use a shortcut to run a program that requires an argument.

On the Home screen, you can use keyboard shortcuts to run up to nine user-defined or assembly-language programs. However, the programs must have the following names.

On Home screen, press: **To run a program, if any, named:**

<input type="checkbox"/> 1	kbdprgm1()
⋮	⋮
<input type="checkbox"/> 9	kbdprgm9()

If you have a program with a different name and you would like to run it with a keyboard shortcut, copy or rename the existing program to kbdprgm1(), etc.

You Cannot Edit an Assembly-Language Program

You cannot use your TI-89 / TI-92 Plus to edit an assembly-language program. The built-in Program Editor will not open assembly-language programs.

Displaying a List of Assembly-Language Programs

To list the assembly-language programs stored in memory:

1. Display the VAR-LINK screen ($\boxed{2nd}$ [VAR-LINK]).
2. Press $\boxed{F2}$ View.
3. Select the applicable folder (or All folders) and set Var Type = Assembly.
4. Press \boxed{ENTER} to display the list of assembly-language programs.



Note: Assembly-language programs have an ASM data type.

For Information about Writing an Assembly-Language Program

Note: You must use a computer to write assembly-language programs. You cannot create assembly-language programs from the calculator keyboard.

The information required to teach a novice programmer how to write an assembly-language program is beyond the scope of this book. However, if you have a working knowledge of assembly language, please check the Texas Instruments web site (education.ti.com) for specific information about how to access TI-89 / TI-92 Plus features.

The TI-89 / TI-92 Plus also includes an **Exec** command that executes a string consisting of a series of Motorola 68000 op-codes. These codes act as another form of an assembly-language program. Check the Texas Instruments web site for available information.

Warning: **Exec** gives you access to the full power of the microprocessor. Please be aware that you can easily make a mistake that locks up the calculator and causes you to lose your data. We suggest you make a backup of the calculator contents before attempting to use the **Exec** command.