TI-Nspire™ Program Editor Guidebook

Important Information

Except as otherwise expressly stated in the License that accompanies a program, Texas Instruments makes no warranty, either express or implied, including but not limited to any implied warranties of merchantability and fitness for a particular purpose, regarding any programs or book materials and makes such materials available solely on an "as-is" basis. In no event shall Texas Instruments be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of the purchase or use of these materials, and the sole and exclusive liability of Texas Instruments, regardless of the form of action, shall not exceed the amount set forth in the license for the program. Moreover, Texas Instruments shall not be liable for any claim of any kind whatsoever against the use of these materials by any other party.

The TI-Nspire™ software uses Lua as scripting environment. For copyright and license information, see http://www.lua.org/license.html.

The TI-Nspire™ software uses Chipmunk Physics version 5.3.4 as simulation environment. For license information, see http://chipmunk-physics.net/release/Chipmunk-5.x/Chipmunk-5.3.4-Docs/.

Microsoft® and Windows® are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Mac OS®, iPad® and OS X® are registered trademarks of Apple Inc.

Unicode® is a registered trademark of Unicode, Inc. in the United States and other countries.

Actual products may vary slightly from provided images.

Contents

Getting Started with the Program Editor

You can create user-defined functions or programs by typing definition statements on the Calculator entry line or by using the Program Editor. The Program Editor offers some advantages, and it is covered in this section. For more information, see *Calculator*.

- The editor has programming templates and dialog boxes to help you define functions and programs using correct syntax.

- The editor lets you enter multiple-line programming statements without requiring a special key sequence to add each line.

- You can easily create private and public library objects (variables, functions, and programs). For more information, see *Libraries*.

- Syntax highlighting provides visual aid for different program elements as follows: Keywords/built-in commands in blue, Operators in red, Strings in green, Comments in gray, and numbers in black.

Note: In most cases, functionality is identical between the Computer and the Handheld views. You may see some differences between computing platforms.
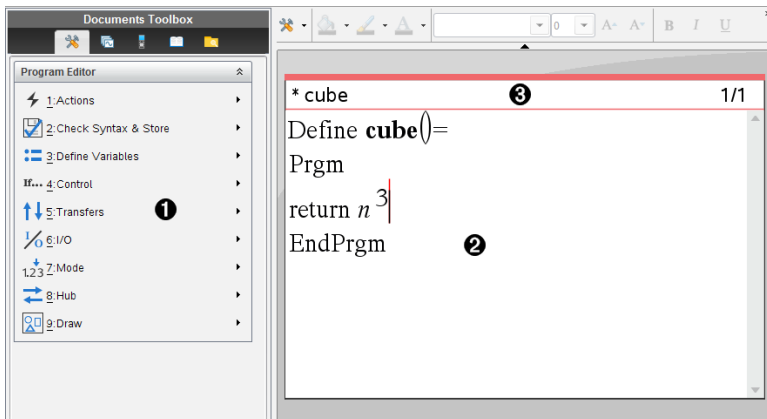
Launching the Program Editor

▶ To add a new Program Editor page in the current problem:

From the toolbar, click **Insert > Program Editor > New**.

Handheld: Press $\boxed{\text{doc} \blacktriangledown}$ and select **Insert > Program Editor > New**.

Note: The editor is also accessible from the **Functions & Programs** menu of a Calculator page.

❶ Program Editor menu – This menu is available anytime you are in the Program Editor work area using the Normal view mode.

❷ Program Editor work area

❸ Status line shows line-number information and the name of the function or program being edited. An asterisk (*****) indicates that this function is "dirty," which means that it has changed since the last time its syntax has been checked and it has been stored.

Defining a Program or Function

Starting a new Program Editor

1. Make sure you are in the document and problem in which you want to create the program or function.

2. Click **Insert** button 🔳 on the application toolbar, and select **Program Editor > New**. (On the handheld, press [doc▾] and select **Insert > Program Editor > New**.)
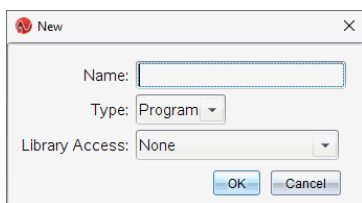
3. Type a name for the function or program you are defining.

4. Select the **Type** (**Program** or **Function**).

5. Set the **Library Access**:

   - To use the function or program only from the current document and problem, select **None**.

   - To make the function or program accessible from any document but not visible in the Catalog, select **LibPriv**.

   - To make the function or program accessible from any document and also visible in the Catalog, select **LibPub (Show in Catalog)**. For more information, see *Libraries*.
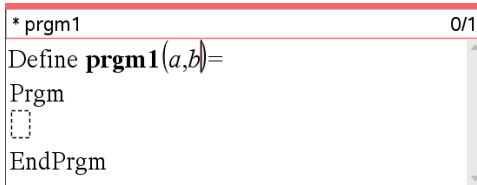
6. Click **OK**.

   A new instance of the Program Editor opens, with a template matching the selections you made.
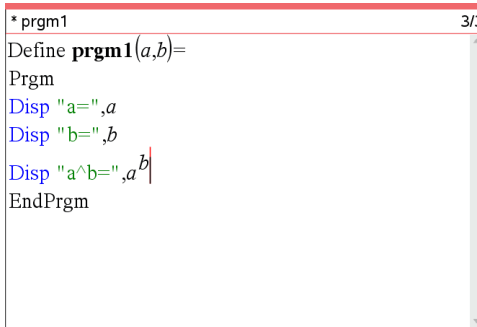
Entering Lines into a Function or Program

The Program Editor does not execute the commands or evaluate expressions as you type them. They are executed only when you evaluate the function or run the program.

1. If your function or program will require the user to supply arguments, type parameter names in the parentheses that follow the name. Separate parameters with a comma.

* prgm1	0/1

Define **prgm1**$(a,b)=$
Prgm
□
EndPrgm

2. Between the Func and EndFunc (or Prgm and EndPrgm) lines, type the lines of statements that make up your function or program.

* prgm1	3/3

Define **prgm1**$(a,b)=$
Prgm
Disp "a=",a
Disp "b=",b
Disp "a^b=",a^b|
EndPrgm

-   You can either type the names of functions and commands or insert them from the Catalog.
-   A line can be longer than the width of the screen; if so, you might have to scroll to view the entire statement.
-   After typing each line, press **Enter**. This inserts a new blank line and lets you continue entering another line.
-   Use the ◄, ►, ▲, and ▼ arrow keys to scroll through the function or program for entering or editing commands.

Inserting Comments

Comments can be useful to someone viewing or editing the program. They are not displayed when the program runs and have no effect on program flow. The © symbol displays at the beginning of the line with the comment.

Define LibPub **volcyl**(ht,r)=

Prgm

©volcyl(ht,r) => volume of cylinder ❶

Disp "Volume=",approx$\left(\pi \cdot r^2 \cdot ht\right)$

©This is another comment.

EndPrgm

❶ Comment showing required syntax. Because this library object is public and this comment is the first line in a Func or Prgm block, the comment is displayed in the Catalog as help. For more information, see *Libraries*.

To insert a comment:

1. Position the cursor at the end of the line in which you want to insert a comment.

2. From the **Actions** menu, click **Insert Comment**, or press **Ctrl+T**.

3. Type the text of the comment after the © symbol.

Checking Syntax

The Program Editor lets you check the function or program for correct syntax.

▶ From the **Check Syntax & Store** menu, click **Check Syntax**.

If the syntax checker finds any syntax errors, it displays an error message and tries to position the cursor near the first error so you can correct it.

Define **prgm**

Prgm

Disp "a=",a

Disp "b=",b

Disp "a^b=",a^b

EndPrgm

> Error   ×
>
> Syntax
>
> OK

Storing the Function or Program

You must store your function or program to make it accessible. The Program Editor automatically checks the syntax before storing.

An asterisk (*) is displayed in the upper left corner of the Program Editor to indicate that the function or program has not been stored.

▶ From the **Check Syntax & Store** menu, click **Check Syntax & Store**.

If the syntax checker finds any syntax errors, it displays an error message and tries to position the cursor near the first error.

If no syntax errors are found, the message "Stored successfully" is displayed in the status line at the top of the Program Editor.

Note: If the function or program is defined as a library object, you must also save the document in the designated library folder and refresh libraries to make the object accessible to other documents. For more information, see *Libraries*.
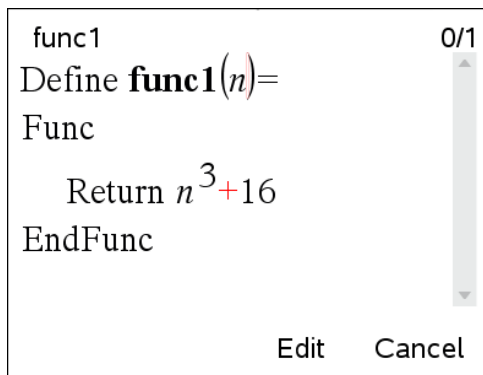
Viewing a Program or Function

1. From the **Actions** menu, click **View**.
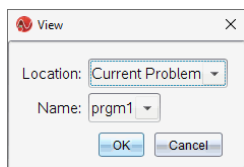
2. If the function or program is a library object, select its library from the **Location** list.
3. Select the function or program name from the **Name** list.

The function or program is displayed in a viewer.

4. Use the arrow keys to view the function or program.
5. If you want to edit the program, click **Edit**.

Handheld: Press `tab` to highlight **Edit**, and then press `enter`.

Note: The **Edit** selection is available only for functions and programs defined in the current problem. To edit a library object, you must first open its library document.

Opening a Function or Program for Editing

You can open a function or program from the current problem only.

Note: You cannot modify a locked program or function. To unlock the object, go to a Calculator page and use the **unLock** command.

1. Display the list of available functions and programs.

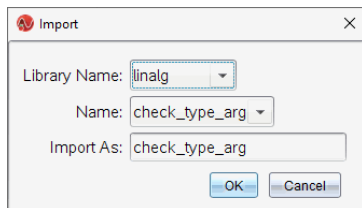   -   From the **Actions** menu, click **Open**.
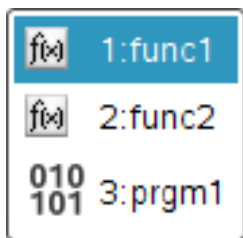
2. Click the item to open.

Importing a Program from a Library

You can import a function or program defined as a library object into a Program Editor within the current problem. The imported copy is not locked, even if the original is locked.

1. From the **Actions** menu, click **Import**.

2. Select the **Library Name**.
3. Select the **Name** of the object.
4. If you want the imported object to have a different name, type the name under **Import As**.

Creating a Copy of a Function or Program

When creating a new function or program, you might find it easier to start with a copy of the current one. The copy that you create is not locked, even if the original is locked.
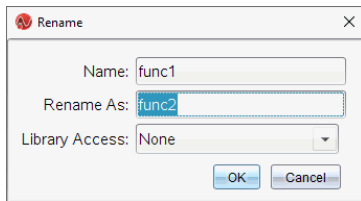
1. From the **Actions** menu, click **Create Copy**.

2. Type a new name, or click **OK** to accept the proposed name.

3. If you want to change the access level, select **Library Access**, and select a new level.

Renaming a Program or Function

You can rename and (optionally) change the access level of the current function or program.

1. From the **Actions** menu, click **Rename**.
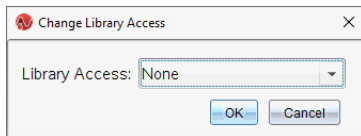
2. Type a new name, or click **OK** to accept the proposed name.

3. If you want to change the access level, select **Library Access**, and select a new level.

Changing the Library Access Level

1. From the **Actions** menu, click **Change Library Access**.
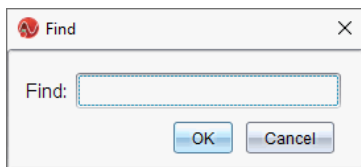
2. Select the **Library Access**:

   - To use the function or program only from the current Calculator problem, select **None**.

   - To make function or program accessible from any document but not visible in the Catalog, select **LibPriv**.

   - To make the function or program accessible from any document and also visible in the Catalog, select **LibPub**.

Finding Text

1. From the **Actions** menu, click **Find**.

2. Type the text that you want to find, and click **OK**.

   - If the text is found, it is highlighted in the program.
   - If the text is not found, a notification message is displayed.

Finding and Replacing Text

1. From the **Actions** menu, click **Find and Replace**.

2. Type the text that you want to find.

3. Type the replacement text.

4. Click **Replace** to replace the first occurrence after the cursor position.
   —or—
   Click **Replace All** to replace every occurrence.

   Note: If the text is found in a math template, a message is displayed to warn you that your replacement text will replace the whole template—not just the found text.
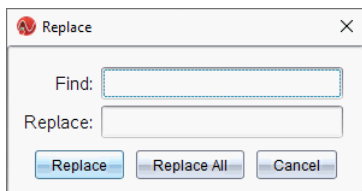
Closing the Current Function or Program

▸ From the **Actions** menu, click **Close**.

   If the function or program has unstored changes, you are prompted to check syntax and store before closing.

Running Programs and Evaluating Functions

After defining and storing a program or function, you can use it from an application. All the applications can evaluate functions, but only the Calculator and Notes applications can run programs.

The program statements are executed in sequential order (although some commands alter the program flow). The output, if any, is displayed in the application's work area.

- Program execution continues until it reaches the last statement or a **Stop** command.
- Function execution continues until it reaches a **Return** command.

Running a Program or Function from the Program Editor

1. Make sure you have defined a program or function and the Program Editor is the active pane (computer) or page (handheld).

2. On the toolbar, click the **Document Tools** button [icon] and select **Check Syntax & Store > Run**.
—or—
Press **Ctrl+R**.

Handheld: Press menu ② ③, or press ctrl **R**.

This will automatically:

- check the syntax and store the program or function,

- paste the program or function name on the first available line of the Calculator application immediately following the Program Editor. If no Calculator exists in that position, a new one is inserted.

prgm1()
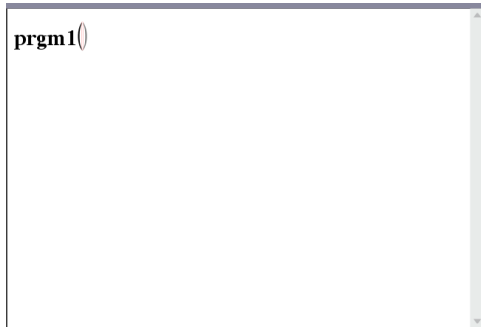
3. If the program or function requires you to supply one or more arguments, type the values or variable names inside the parentheses.

4. Press enter.

Note: You can also run a program or function in Calculator or Notes applications by typing the name of the program with parentheses and any required arguments and pressing enter.

Using Short and Long Names

Anytime you are in the same problem where an object is defined, you can access it by entering its short name (the name given in the object's **Define** command). This is the case for all defined objects, including private, public, and non-library objects.

You can access a library object from any document by typing the object's long name. A long name consists of the name of the object's library document followed by a backslash "\" followed by the name of the object. For example, the long name of the object defined as **func1** in the library document **lib1** is **lib1\func1**. To type the "\" character on the handheld, press ⇧shift ÷.

Note: If you cannot remember the exact name or the order of arguments required for a private library object, you can open the library document or use the Program Editor to view the object. You also can use **getVarInfo** to view a list of objects in a library.

Using a Public Library Program or Function

1. Make sure you have defined the object in the document's first problem, stored the object, saved the library document in the MyLib folder, and refreshed the libraries.

2. Open the TI-Nspire™ application in which you want to use the program or function.

   Note: All applications can evaluate functions, but only the Calculator and Notes applications can run programs.

3. Open the Catalog and use the library tab to find and insert the object.
   —or—
   Type the name of the object. In the case of a program or function, always follow the name with parentheses.

   libs2\func1()

4. If the program or function requires you to supply one or more arguments, type the values or variable names inside the parentheses.

   libs2\func1(34,power)

5. Press ⌊enter⌋.

Using a Private Library Program or Function

To use a Private library object, you must know its long name. For example, the long name of the object defined as **func1** in the library document **lib1** is **lib1\func1**.

Note: If you cannot remember the exact name or the order of arguments required for a private library object, you can open the library document or use the Program Editor to view the object.

1. Make sure you have defined the object in the document's first problem, stored the object, saved the library document in the MyLib folder, and refreshed the libraries.

2. Open the TI-Nspire™ application in which you want to use the program or function.

   Note: All applications can evaluate functions, but only the Calculator and Notes applications can run programs.

3. Type the name of the object. In the case of a program or function, always follow the name with parentheses.

   libs2\func1()

4. If the object requires you to supply one or more arguments, type the values or variable names inside the parentheses.

   libs2\func1(34,power)

5. Press ⌊enter⌋.

Interrupting a Running Program or Function

While a program or function is running, the busy pointer ⟳ is displayed.

▶ To stop the program or function,

- Windows®: Hold down the **F12** key and press **Enter** repeatedly.

- Mac®: Hold down the **F5** key and press **Enter** repeatedly.

- Handheld: Hold down the [⌂ on] key and press [enter] repeatedly.

A message is displayed. To edit the program or function in the Program Editor, select **Go To**. The cursor appears at the command where the break occurred.
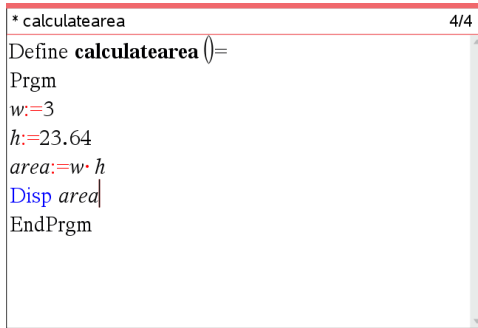
Getting Values into a Program

You can choose from several methods to supply the values that a function or program uses in calculations.

Embedding the Values Within the Program or Function

This method is useful primarily for values that must be the same each time the program or function is used.

1. Define the program.

* calculatearea	4/4
Define **calculatearea** ()=	
Prgm	
w:=3	
h:=23.64	
area:=w· h	
Disp *area*	
EndPrgm	

2. Run the program.

calculatearea()	
	70.92
	Done

Letting the User Assign the Values to Variables

A program or function can refer to variables created beforehand. This method requires users to remember the variable names and to assign values to them before using the object.

1. Define the program.

* calculatearea	2/2

```
Define calculatearea ()=
Prgm
area:=w· h
Disp area|
EndPrgm
```

2. Supply the variables, and then run the program.

w:=3	3
h:=23.64	23.64
calculatearea()	
	70.92
	Done

Letting the User Supply the Values as Arguments

This method lets users pass one or more values as arguments within the expression that calls the program or function.

The following program, **volcyl**, calculates the volume of a cylinder. It requires the user to supply two values: height and radius of the cylinder.

1. Define the **volcyl** program.

```
* volcyl                                    1/1
Define volcyl(height,radius)=
Prgm
Disp "Volume =", approx(π· radius² · height)
EndPrgm
```

2. Run the program to display the volume of a cylinder with a height of 34 mm and a radius of 5 mm.

$volcyl(34,5)$

Volume $= 2670.35$

Done

Note: You do not have to use the parameter names when you run the **volcyl** program, but you must supply two arguments (as values, variables, or expressions). The first must represent the height, and the second must represent the radius.

Requesting the Values from the User (Programs Only)

You can use the **Request** and **RequestStr** commands in a program to make the program pause and display a dialog box prompting the user for information. This method does not require users to remember variable names or the order in which they are needed.

You cannot use the **Request** or **RequestStr** command in a function.

1. Define the program.

```
Define calculatearea ()=
Prgm
Request "Width: ", w
Request "Height: ", h
area:=w· h
EndPrgm
```

2. Run the program and respond to the requests.

$calculatearea()$: $area$

Width: 3

Height: 23.64

70.92

Use **RequestStr** instead of **Request** when you want the program to interpret the user's response as a character string rather than a math expression. This avoids requiring the user to enclose the response in quotation marks ("").

Displaying Information from a Function or Program

A running function or program does not display intermediate calculated results unless you include a command to display them. This is an important difference between performing a calculation on the entry line and performing it in a function or program.

The following calculations, for example, do not display a result in a function or program (although they do from the entry line).

Define **prgm2**()=
Prgm
$x:=12 \cdot 6$
$\cos\left(\dfrac{\pi}{4}\right)$
EndPrgm

Displaying Information in the History

You can use the **Disp** command in a program or function to display information, including intermediate results, in the history.

Define **prgm2**()=
Prgm
Disp $12 \cdot 6$
Disp "Result: ",$\cos\left(\dfrac{\pi}{4}\right)$
EndPrgm

Displaying Information in a Dialog Box

You can use the **Text** command to pause a running program and display information in a dialog box. The user clicks **OK** to continue or clicks **Cancel** to stop the program.

You cannot use the **Text** command in a function.

Define **sample**()=
Prgm
Text "Area=" & *area*
EndPrgm

Note: Displaying a result with **Disp** or **Text** does not store that result. If you expect to refer later to a result, store it to a global variable.

Define **sample**$()=$
Prgm
$\cos(\pi/4) \rightarrow maximum$
Disp $maximum$
EndPrgm

$sample()$

0.707107

$Done$

Using Local Variables

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

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

```
* loop_prog                                    0/5
Define loop_prog()=
Prgm
Local i    ❶
For i,0,5,1
  Disp i
EndFor
Disp i
EndPrgm
```

❶ Declares variable i as local.

Note: When possible, declare as local any variable that is used only within the program and does not need to be available after the program stops.

What Causes an Undefined Variable Error Message?

An **Undefined** variable error message is displayed 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).

For example:

```
* fact                                         5/5
Define fact(n)=
Func
Local m    ❶
While n>1
  n· m → m: n−1 → n
EndWhile
Return m
EndFunc
```

❶ Local variable m is not assigned an initial value.

Initialize Local Variables

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

```
Define fact(n)=
Func
Local m: 1 → m  ❶
While n>1
  n· m → m: n−1 → n
EndWhile
Return m
EndFunc
```

❶ 1 is stored as the initial value for *m*.

Note (CAS): Functions and programs cannot use a local variable to perform symbolic calculations.

CAS: Performing 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 a program to delete the global variable, if it exists, before referring to it. (**DelVar** does not delete locked or linked variables.)

Differences Between Functions and Programs

A function defined in the Program Editor is similar to the functions built into the TI-Nspire™ Software.

•   Functions must return a result, which can be graphed or entered in a table. Programs do not return a result.

•   You can use a function (but not a program) within an expression. For example: **3 • func1(3)** is valid, but not **3 • prog1(3)**.

•   You can run programs from Calculator and Notes applications only. However, you can evaluate functions in Calculator, Notes, Lists & Spreadsheet, Graphs & Geometry, and Data & Statistics.

•   A function can refer to any variable; however, it can store a value to a local variable only. Programs can store to local and global variables.

  Note: Arguments used to pass values to a function are treated as local variables automatically. If you want to store to any other variables, you must declare them as **Local** from within the function.

•   A function cannot call a program as a subroutine, but it can call another user-defined function.

- You cannot define a program within a function.
- A function cannot define a global function, but it can define a local function.

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 that you use to run the program from the entry line.
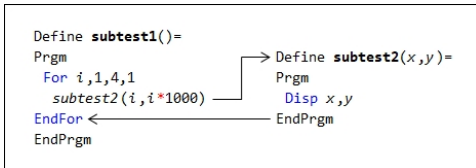
Defining and Calling an Internal Subroutine

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.

❶ Declares the subroutine as a local variable.

❷ Defines the subroutine.

❸ Calls the subroutine.

Note: Use the Program Editor's **Var** menu to enter the **Define** and **Prgm...EndPrgm** commands.

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 **Return** with no argument.

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.

Avoiding 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 variables that are used in evaluating the function or running the program. For example:

$x+1 \rightarrow x$ ❶

– or –

For $i,i,10,1$
  Disp i ❶
EndFor

> ❶ 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.

Controlling the Flow of a Function or Program

When you run a program or evaluate a function, 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.
- Loop commands such as **For...EndFor** repeat a group of commands.

Using If, Lbl, and Goto to Control Program Flow

The **If** command and several **If...EndIf** structures let you execute a statement or block of statements conditionally, that is, based on the result of a test (such as x>5). **Lbl** (label) and **Goto** commands let you branch, or jump, from one place to another in a function or program.

The **If** command and several **If...EndIf** structures reside on the Program Editor's **Control** menu.

When you insert a structure such as **If...Then...EndIf**, a template is inserted at the cursor location. The cursor is positioned so that you can enter a conditional test.

If Command

To execute a single command when a conditional test is true, use the general form:

If $x > 5$
  Disp "x is greater than 5 " ❶
Disp x ❷

❶ Executed only if x>5; otherwise, skipped.

❷ Always displays the value of x.

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

If...Then...EndIf Structures

To execute one group of commands if a conditional test is true, use the structure:

If $x > 5$ Then
  Disp "x is greater than 5 " ❶
  $2 \cdot x \rightarrow x$ ❶
EndIf
Disp x ❷

❶ Executed only if x>5.

Displays the value of:

❷ 2x if x>5
  x if x≤5

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

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:

```
If x>5 Then
  Disp "x is greater than 5 " ❶
  2·x→x ❶
Else
  Disp "x is less than or equal to 5 " ❷
  5·x→x ❷
EndIf
Disp x ❸
```

❶ Executed only if x>5.

❷ Executed only if x≤5.

❸ Displays value of:
  2x if x>5
  5x if x≤5

If...Then...ElseIf... EndIf Structures

A more complex form of the **If** command lets you test for multiple conditions. Suppose you want a program to test a user-supplied argument that signifies one of four options.

To test for each option (If Choice=1, If Choice=2, and so on), use the **If...Then...ElseIf...EndIf** structure.

Lbl and Goto Commands

You can also control the flow by using **Lbl** (label) and **Goto** commands. These commands reside on the Program Editor's **Transfers** menu.

Use the **Lbl** command to label (assign a name to) a particular location in the function or 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 function or 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
  Goto GT5  ❶
Disp x

............

Lbl GT5  ❷
Disp "The number was > 5"
```

❶ If x>5, branches directly to label GT5.

❷ For this example, the program must include commands (such as **Stop**) that prevent **Lbl** GT5 from being executed if x≤5.

Using Loops to Repeat a Group of Commands

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

Loop and loop-related commands reside on the Program Editor's **Control** and **Transfers** menus.

When you insert one of the loop structures, its template is inserted at the cursor location. You can then begin entering the commands that will be executed within 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*]
    ❶   ❷   ❸   ❹

❶ Name of a variable to be used as a counter

❷ Value assigned to *variable* when the **For** loop begins.

❸ Value compared to the current value of *variable* at each iteration of the loop. The loop exits when *variable* exceeds *end*.

❹ Value added to *variable* at each iteration of the loop (This argument is optional. The default *increment* is 1.)

At each iteration of the **For** loop, the *variable* value is compared to the *end* value. If *variable* does not exceed *end*, the commands within the **For...EndFor** loop are executed and the loop repeats; otherwise, control jumps to the command following **EndFor**.
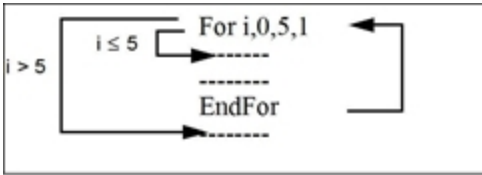
Note: The **For** command automatically increments the counter variable so that the function or program can exit the loop after a certain number of repetitions.

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

For example:

```
For i,0,5,1
  Disp i ❶
EndFor
Disp i ❷
```

❶ Displays 0, 1, 2, 3, 4, and 5.

❷ Displays 6. When *variable* increments to 6, the loop is not executed.

Notes:

- You can declare *variable* as local if it does not need to be saved after the function or program stops.

- You can set *end* to a value less than *begin*, provided you also set *increment* to a negative value.

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, *condition* is evaluated. If *condition* is true, the loop is executed; otherwise, control jumps to the command following **EndWhile**.
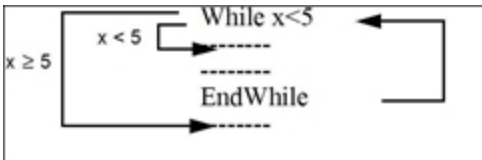
Note: The **While** command does not automatically change the condition. You must include commands that allow the function or program to exit the loop.

At the end of the loop (**EndWhile**), 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 function or program, or you can 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 function or program cannot exit the loop (called an infinite loop).
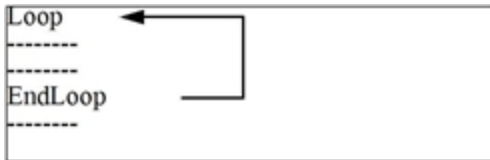
For example:

$$0 \rightarrow x \quad \text{❶}$$
$$\text{While } x < 5$$
$$\quad \text{Disp } x \quad \text{❷}$$
$$\quad x+1 \rightarrow x \quad \text{❸}$$
$$\text{EndWhile}$$
$$\text{Disp } x \quad \text{❹}$$

❶ Initially sets x.

❷ Displays 0, 1, 2, 3, and 4.

❸ Increments x.

❹ Displays 5. When x increments to 5, the loop is 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, you insert commands in the loop that let the program exit from the loop. Commonly used commands are: **If**, **Exit**, **Goto**, and **Lbl** (label). For example:

$0 \rightarrow x$
Loop
  Disp x
  $x+1 \rightarrow x$
  If $x>5$ ❶
    Exit
EndLoop
Disp x ❷

❶ An **If** command checks the condition.

❷ Exits the loop and jumps to here when x increments to 6.

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

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:

Lbl START ◄─────────┐
--------   │
--------   │
Goto START ─────────┘

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

Changing Mode Settings

Functions and programs can use the **setMode()** function to temporarily set specific calculation or result modes. The Program Editor's **Mode** menu makes it easy to enter the correct syntax without requiring you to memorize numeric codes.

Note: Mode changes made within a function or program definition do not persist outside the function or program.

Setting a Mode

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

2. From the **Mode** menu, click the mode to change, and click the new setting.

   The correct syntax is inserted at the cursor location. For example:

   setMode(1,3)

Debugging Programs and Handling Errors

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

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

Techniques for Debugging

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

- Temporarily insert **Disp** commands to display the values of critical variables.

- To confirm that a loop is executed the correct number of times, use **Disp** to display the counter variable or the values in the conditional test.

- To confirm that a subroutine is executed, use **Disp** to display messages such as "Entering subroutine" and "Exiting subroutine" at the beginning and end of the subroutine.

- To stop a program or function manually,

   - Windows®: Hold down the **F12** key and press **Enter** repeatedly.

   - Mac®: Hold down the **F5** key and press **Enter** repeatedly.

   - Handheld: Hold down the `⌂ on` key and press `enter` repeatedly.

Error-handling Commands

Command	Description
Try...EndTry	Defines a block that lets a function or program execute a command and, if necessary, recover from an error generated by that command.

Command	Description
ClrErr	Clears the error status and sets system variable errCode to zero. For an example of using errCode, see the Try command in the *Reference Guide*.
PassErr	Passes an error to the next level of the **Try...EndTry** block.

General Information

Online Help

education.ti.com/eguide

Select your country for more product information.

Contact TI Support

education.ti.com/ti-cares

Select your country for technical and other support resources.

Service and Warranty Information

education.ti.com/warranty

Select your country for information about the length and terms of the warranty or about product service.

Limited Warranty. This warranty does not affect your statutory rights.

Texas Instruments Incorporated

12500 TI Blvd.

Dallas, TX 75243