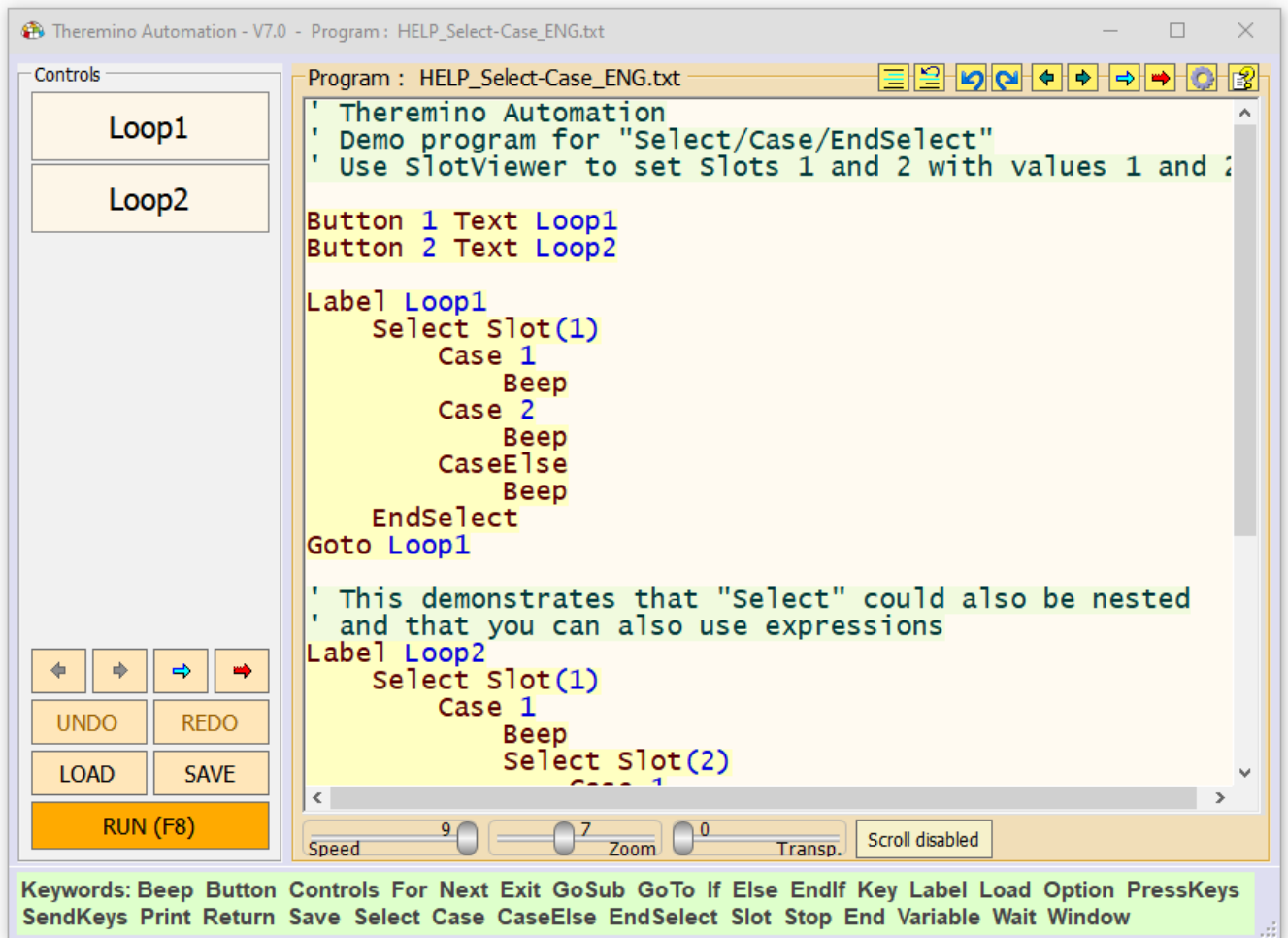


# Theremino System



# Theremino Automation V7.9

## Main Index

<b>Premises.....</b>	<b>6</b>
Run multiple instances of Automation.....	7
Load and run programs.....	7
Edit and save programs.....	8
Program execution.....	9
Executed lines visualization.....	10
The vertical scrollbar and the Bookmarks.....	11
Test the values during the execution.....	12
Simple programs to start.....	13
Program structure.....	14
Resize the application controls.....	15
Keywords.....	16
<b>Keywords one by one.....</b>	<b>17</b>
Array.....	17
ArrayClear.....	17
ArrayLength function.....	17
ArrayToString function.....	17
Beep.....	18
Beep Frequency Duration.....	18
Beep "String of chars".....	18
Button.....	19
Button - Identify the corresponding labels.....	19
Button Text.....	20
Button Disabled / Enabled.....	20
Button Slot.....	20
Button Color.....	21
Button Color Flashing and RGB.....	22
Button used as Labels.....	23
Button Text and Identifiers.....	24
Button - Using formula instead of numbers.....	25
Button that change state.....	26
Buttons with images.....	27
COM (serial port) - Commands .....	28
COM (serial port) - Functions .....	29
COM (serial port) - Receive buffer.....	30
Controls.....	31
Controls - SetCursorPos <X Y>.....	31
Controls - SetBackColor <color>.....	31
Controls - TextBox.....	32

<b>Controls - Close and resize the TextBox.....</b>	<b>33</b>
<b>For - Next.....</b>	<b>34</b>
<b>For - Next with Step.....</b>	<b>35</b>
<b>For - Next instead of While.....</b>	<b>36</b>
<b>Exit.....</b>	<b>37</b>
<b>Exit "n".....</b>	<b>38</b>
<b>Goto - Gosub - Return.....</b>	<b>39</b>
<b>Eliminate Gosubs and send parameters to Functions.....</b>	<b>39</b>
<b>If - Else - Endif.....</b>	<b>40</b>
<b>Key.....</b>	<b>41</b>
<b>Label.....</b>	<b>42</b>
<b>Eliminate Gosubs and send parameters to Functions.....</b>	<b>42</b>
<b>Label EventStop.....</b>	<b>43</b>
<b>Label EventTimer.....</b>	<b>44</b>
<b>Label Event_DroppedFile.....</b>	<b>45</b>
<b>Label Event_ExternalCommands.....</b>	<b>46</b>
<b>Commands from COBOT to Automation.....</b>	<b>47</b>
<b>Commands from Automation to COBOT.....</b>	<b>48</b>
<b>External commands to Automation Buttons.....</b>	<b>49</b>
<b>Load (Applications) .....</b>	<b>50</b>
<b>Load OpenApps.....</b>	<b>51</b>
<b>Load CloseApps.....</b>	<b>51</b>
<b>Load CloseAll.....</b>	<b>51</b>
<b>Load CloseApps file-name.....</b>	<b>52</b>
<b>Load CloseApps process-name.....</b>	<b>52</b>
<b>Load CloseApps Windows.....</b>	<b>52</b>
<b>Load Slots / Save Slots.....</b>	<b>53</b>
<b>Load Vars / Save Vars.....</b>	<b>53</b>
<b>Load Vars - Beware of Initializations.....</b>	<b>54</b>
<b>Load (Images, Videos and Sounds) .....</b>	<b>55</b>
<b>Load (txt, pdf, doc, etc...) .....</b>	<b>56</b>
<b>Load (Program) .....</b>	<b>57</b>
<b>Load (Variable from File) .....</b>	<b>57</b>
<b>Load (web address) .....</b>	<b>58</b>
<b>Load (options for web pages).....</b>	<b>59</b>
<b>The "Option" statements.....</b>	<b>60</b>
<b>PressKeys.....</b>	<b>61</b>
<b>SendKeys.....</b>	<b>62</b>
<b>Print.....</b>	<b>63</b>
<b>Save.....</b>	<b>64</b>
<b>Select - Case - CaseElse - EndSelect.....</b>	<b>65</b>
<b>Select - Case (special features).....</b>	<b>66</b>

Slot.....	67
The Command Slot.....	68
SlotText.....	69
Speed.....	69
Stop, End.....	70
TTS (Text to speech).....	70
Variable.....	71
Variable - Immediate assignments.....	72
VarsFromFile.....	73
Wait.....	74
Window commands.....	74
Window.....	75
Calling functions.....	76
Function parameters.....	77
Expressions and Functions.....	78
Expressions results.....	79
The Slot () function.....	79
Numerical functions.....	80
String functions.....	81
Conversion functions.....	82
The Format() and Str() functions.....	83
The MouseX, Y, XP and YP functions.....	84
The MouseButton function.....	84
The MouseWheel function.....	84
The functions that read the pixel colors.....	85
The Key() function.....	86
The “Input” function.....	86
The “Message” function .....	87
The date and time functions.....	88
The “ElapsedTime” function.....	88
The filtering functions.....	89
The “Media” functions.....	90
The XML file functions.....	91
The File and Folder functions.....	92
Auto indentation.....	93
Constructs auto completing.....	93
Windows and Menus.....	94
The control buttons .....	94
The application menu.....	95
The program menu.....	96
Full Screen operation.....	98

<a href="#"><u>The top bar controls.....</u></a>	<a href="#"><u>99</u></a>
<a href="#"><u>The bottom bar controls.....</u></a>	<a href="#"><u>100</u></a>
<a href="#"><u>The Debug Window.....</u></a>	<a href="#"><u>101</u></a>
<a href="#"><u>The Debug Window (Breakpoints).....</u></a>	<a href="#"><u>102</u></a>
<a href="#"><u>The Debug Window (Watches).....</u></a>	<a href="#"><u>103</u></a>
<a href="#"><u>The Debug Window (Exec).....</u></a>	<a href="#"><u>104</u></a>
<a href="#"><u>The Debug Window (Pause Button).....</u></a>	<a href="#"><u>105</u></a>
<a href="#"><u>The Debug Window (Pause Button with LEDs).....</u></a>	<a href="#"><u>106</u></a>
<a href="#"><u>The Debug Window (Pause Button adapters).....</u></a>	<a href="#"><u>107</u></a>
<a href="#"><u>The Debug Window (Recycle Bin).....</u></a>	<a href="#"><u>108</u></a>
<a href="#"><u>Programming techniques.....</u></a>	<a href="#"><u>109</u></a>
<a href="#"><u>Special applications and folders.....</u></a>	<a href="#"><u>110</u></a>
<a href="#"><u>Open files with Notepad and other apps.....</u></a>	<a href="#"><u>111</u></a>
<a href="#"><u>Open folders.....</u></a>	<a href="#"><u>112</u></a>
<a href="#"><u>The "Theremino Editor" application.....</u></a>	<a href="#"><u>113</u></a>
<a href="#"><u>Start "Theremino Editor" with files.....</u></a>	<a href="#"><u>114</u></a>
<a href="#"><u>Open sequences with Notepad.....</u></a>	<a href="#"><u>114</u></a>
<a href="#"><u>Using "Sequences".....</u></a>	<a href="#"><u>115</u></a>
<a href="#"><u>Using the graphic chars.....</u></a>	<a href="#"><u>116</u></a>
<a href="#"><u>Sew Machines functions.....</u></a>	<a href="#"><u>117</u></a>
<a href="#"><u>Sew functions.....</u></a>	<a href="#"><u>118</u></a>
<a href="#"><u>Sew functions - Examples and Notes.....</u></a>	<a href="#"><u>119</u></a>
<a href="#"><u>Known issues.....</u></a>	<a href="#"><u>120</u></a>

# Premises

We simplified and minimized everything possible to facilitate those who have no experience in programming.

This language is extremely easy to use and understand, but complex operations are also possible, such as playing audio and video, or surfing the internet.

With a single instruction performing tasks that in other languages require specialized knowledge and many pages of code.

## Theremino Automation limits

**Keywords:** Beep Button Controls For Next Exit GoSub GoTo If Else EndIf Key Label Load Option PressKeys SendKeys Print Return Save Select Case CaseElse EndSelect Slot Stop End Variable Wait Window

In Automation instructions are few, about twenty. No classes, types, structures threads, and none complex mechanisms of the classic programming languages.

So for complex tasks, you should switch to more powerful programming environments (e.g. Visual Studio). As a guideline you can count the lines, beyond a thousand or two thousand lines it is better to switch to a more powerful language.

## Theremino Automation is an interpreted language.

In languages "interpreted", unlike those "compiled", the instructions are not pre compiled, but are read, character by character, and interpreted during the execution itself.

The execution of an interpreted language is therefore slower. In our case, the instructions are on average ten times slower than the same instructions written in Visual Studio (Csharp, C ++ or VBNET).

Speed is still abundant for simple tasks it is intended for this language. So much so that we had to add the ability to slow it down further, even thousands of times, to facilitate the understanding of what is happening.



# Run multiple instances of Automation

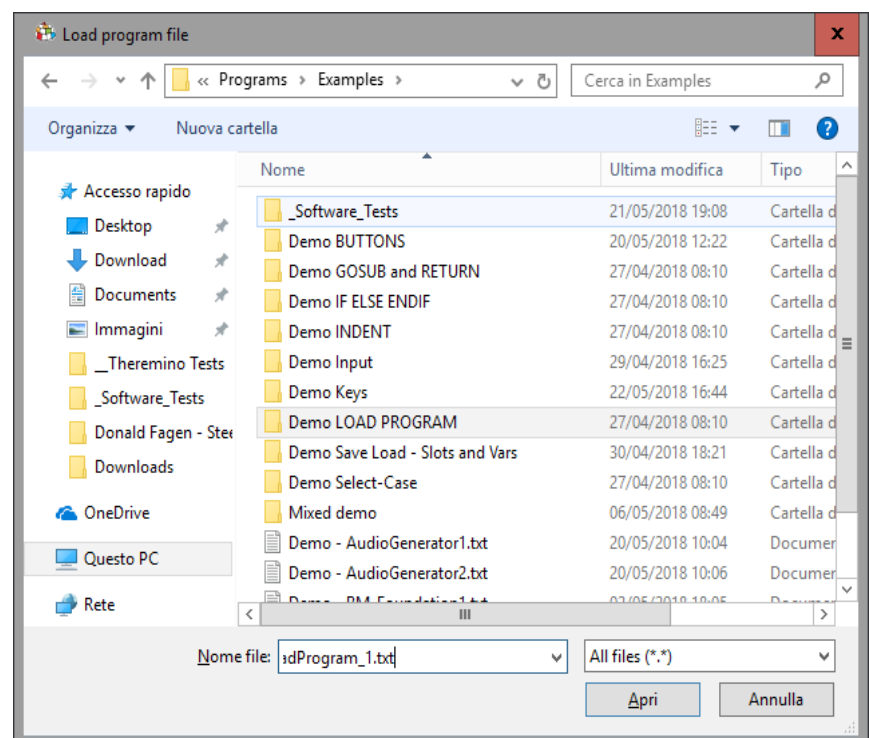
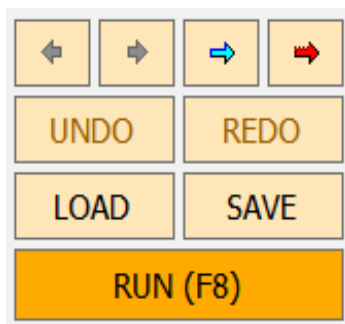
If you try to run a second instance of Automation from the same file it won't open. This is a behavior intended to prevent opening two windows by mistake and also to prevent the options of the two instances from mixing. If necessary, you can force the opening of a second instance by holding SHIFT during startup.

If you want two or more instances of Automation that can work simultaneously and each with independent options, just copy the "Theremino\_Automation.exe" file with a different name.

If you prefer you can keep the files "Theremino\_Automation.exe" in separate folders or even all in the same folder.

## Load and run programs

Using the LOAD button opens a window that lets you choose the programs to load.



After loading the program you execute it with the RUN button.

To stop the program, press the RUN key (which has become STOP),

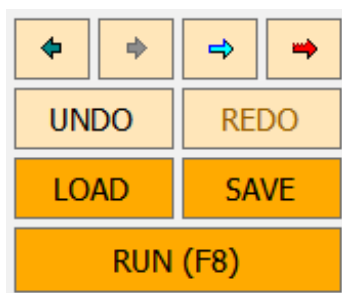
*You can also stop the program by clicking on the program text.*

# Edit and save programs

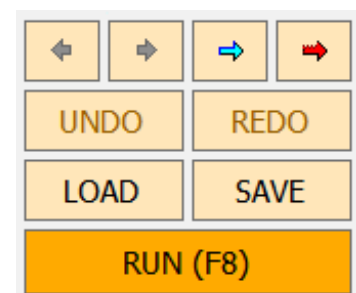
Any changes you make to a program is saved automatically, so there is no need to remember to save it, before closing the application Automation, or load another program.

When you re-open the program will be exactly as you left it the last time.

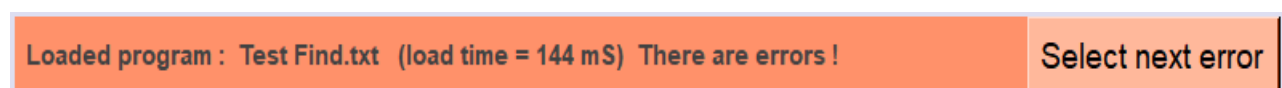
So if you do not want to lose the original, before changing a program you should save it with a different name, using the SAVE button.



The LOAD and SAVE keys highlighted in orange indicate that the program has been modified and not yet saved.



- ◆ **Pressing SAVE with the right mouse button** saves the program quickly.
- ◆ **Pressing LOAD with the right mouse button** save the program and then reload it immediately. With this operation the program is checked from beginning to end and if it contains errors the bottom line of the application will show a message.



If the program contains errors then on the bottom row appers the "Select next error" button, clicking on you can select all the lines with errors.

- - - - -

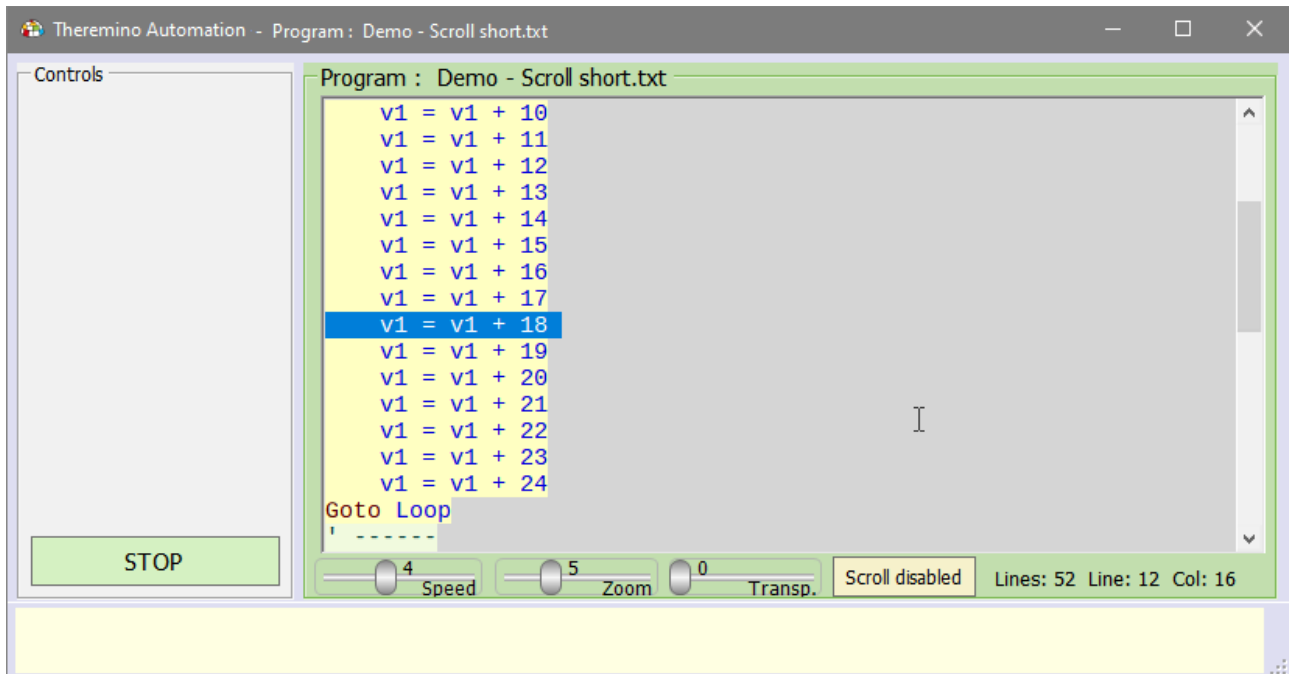
It is also possible to send all old versions of programs to the system trash, see the "Recycle bin" option in [this page](#) .

If needed, you can also recover the original examples from the original ZIP file which you downloaded from the [theremino site](#).



# Program execution

During the execution of the program the line in execution is highlighted with a blue background, as in the following image. In the new versions of Automation (from 7.0 onwards) the last lines executed are also highlighted, as shown on the next page.



The program window behaves in different ways depending on the "Scroll button".

## If the scroll is enabled

- ◆ The window is scrolled vertically to always show the executed line.
- ◆ If the "Speed" is low (1 to 4) then the highlighted line is also kept in the center of the screen.

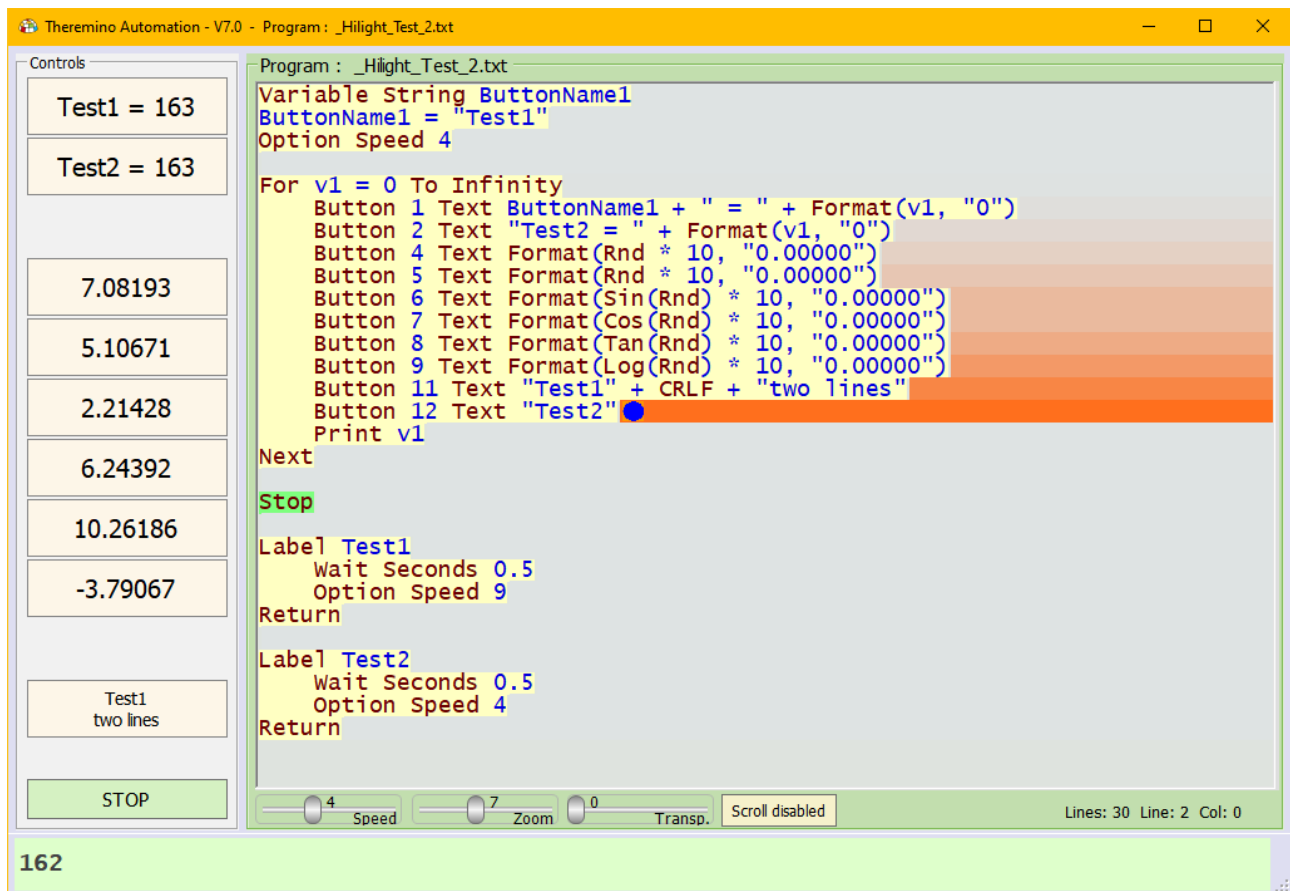
## If the scroll is disabled

- ◆ The lines in execution are highlighted, but the window does not scroll to show them.
- ◆ You can then use the mouse wheel, or the vertical scroll bar, to permanently see the preferred area of the program.

Test also the examples in the folder  
"Demo programs\Demo Execution and Scroll"

# Executed lines visualization

From version 7.0 onwards, the lines performed are highlighted in red, as shown in this image.



The blue cursor indicates the line executed instantly and is followed by a red background color. This color fades slowly until the light gray color of the background is restored after about one second.

If the program runs slowly, for example at speed 4, then you will see the last lines executed with different gradations.

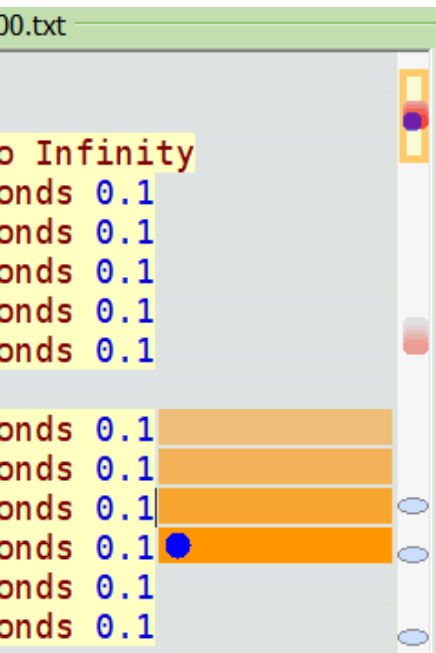
If the program is run quickly then you will see blocks of lines color together and fade together.

These colors are very useful for following the progress of the program. It is advisable to keep the Scroll button disabled and to manually scroll the program with the mouse wheel or the window cursor and also to view parts of the program by clicking the Buttons with the CTRL key pressed as explained at the end of the first page on "Buttons".

Test also the examples in the folder  
"Demo programs" "Demo Hilight Running Lines"

# The vertical scrollbar and the Bookmarks

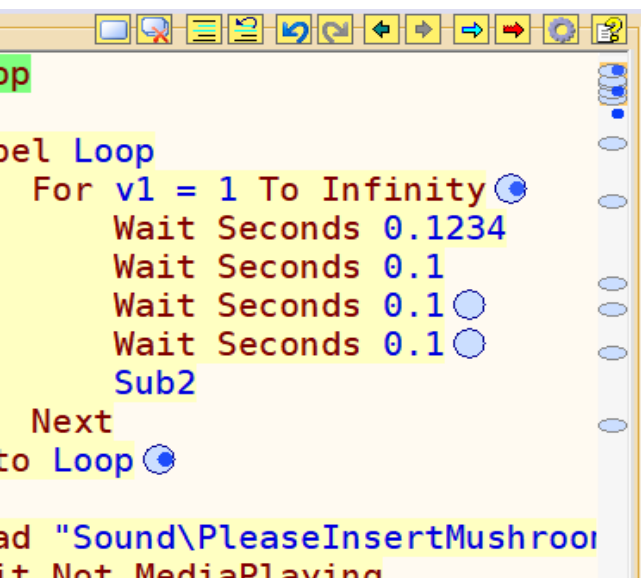
The vertical bar has two different behaviors depending on whether the program is running or editing.



While the program is running, moving orange bars appear in the vertical bar.

By moving the cursor over the colored parts of the bar, the main window shows the parts of the program running. The orange notches are of a degrading color, the more intense ones refer to the last rows performed. The blue dot indicates the program line that is currently running.

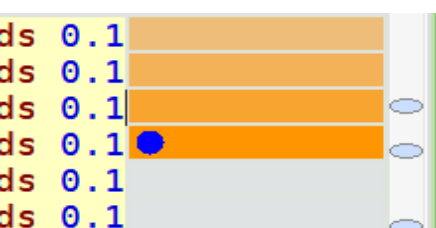
If the program is stopped waiting for something, it may happen that the orange bands fade completely into the gray background color and in this case the blue dot is used to identify where the execution has stopped.



When the program is not running, useful information appears in the vertical bar to identify the various parts of the program. The blue circles on the program and the blue ovals in the bar indicate the lines with bookmarks.

The blue circles and ticks indicate the last lines of the program that have been modified.

The green lines and ticks indicate all occurrences of the part of the program that has been selected (in this case the word "Loop").

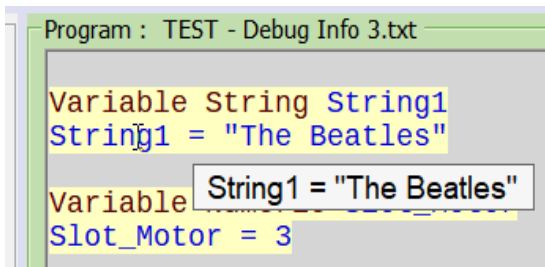


The bookmarks (blue ovals and circles) are also visible during program execution and are very useful for marking important areas and finding them quickly.

# Test the values during the execution

On the previous pages we saw that you can stop the automatic scrolling, then with the mouse wheel you can inspect various parts of the program.

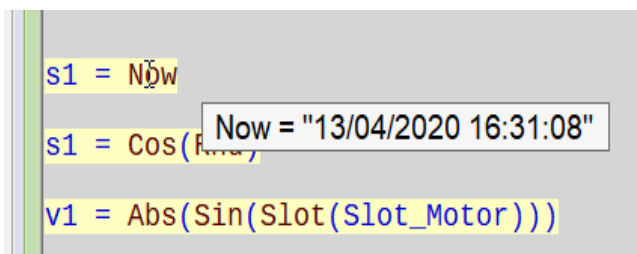
If you want you can also use an [external key](#) to pause the application, or you could place **Stop** instructions at strategic points, or insert **Brakpoints** and open the Debug panel. When the program is running, or the Debug panel is open, you can know the values of variables and functions by moving the mouse cursor on them.



```
Program : TEST - Debug Info 3.txt  
Variable String String1  
String1 = "The Beatles"  
Variable String1 = "The Beatles"  
Slot_Motor = 3
```

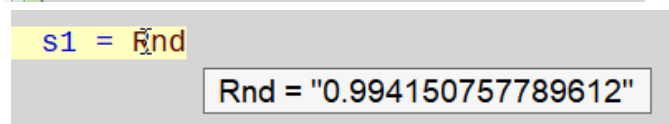
By positioning the mouse on a variable you can immediately know its value, as shown in this image.

Note that the mouse cursor (not very visible) is positioned between the "n" and "g" of "String1".



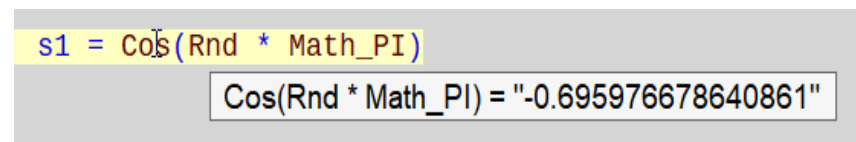
```
s1 = Now  
s1 = Cos(Rnd * Math_PI)  
v1 = Abs(Sin(Slot(Slot_Motor)))
```

Functions can be queried. For example, the "Now" function responds a value that changes over time every second.



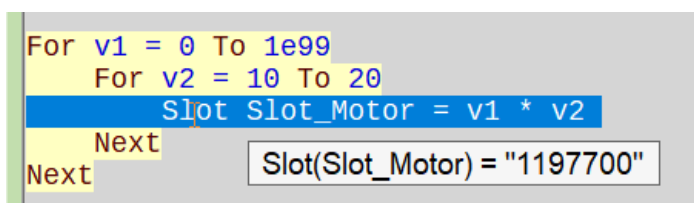
```
s1 = Rnd  
s1 = Cos(Rnd * Math_PI)
```

Also the Rnd function is continuously recalculated and changes over time.



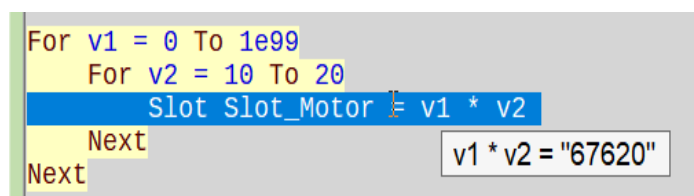
```
s1 = Cos(Rnd * Math_PI)
```

Positioning the cursor on "Cos" you get the result of a complex function.



```
For v1 = 0 To 1e99  
  For v2 = 10 To 20  
    Slot Slot_Motor = v1 * v2  
  Next  
Next  
Slot(Slot_Motor) = "1197700"
```

You can also instantly know the numerical values of the Slots.



```
For v1 = 0 To 1e99  
  For v2 = 10 To 20  
    Slot Slot_Motor = v1 * v2  
  Next  
Next  
v1 * v2 = "67620"
```

And also the results of numerical formulas (to define interesting parts can be enclosed in parentheses)

Experiment with the gli examples in the folder  
"Demo Programs\Demo Fast Debug"

# Simple programs to start

About the program for the first time might find the simple examples that are in the "Simple Programs" folder.

The best way to start is to load these examples, and follow the instructions one at a time, with the "Single Step" of the "Debug" window button (to open it using the right mouse button and select Debug).

In the Debug window you are already present some lines that show the value of variables, of the slot and expressions. However, it is also easy to add new ones.

Pressing repeatedly Single Step, you can see the values change and understand the progress of the program, and how to run the calculations and assigned values.

## Here is an example of a simple program

```
v1 = 10
v2 = 20
v3 = v1 + v2
v4 = v3 * 12
Slot (1) = Sqrt(v4)
Slot (2) = Rnd * 1000
```

Note that, after performing the last line, the program starts automatically from the first line. In case you want to stop it you should add a "Stop" line at the end.

## Here is a second example

```
v1 = Rnd
If v1 > 0.5
    Beep
    Print ""
else
    Print "The v1 value is : " + v1
EndIf
```

In this example the Rnd statement assigns, to the variable v1, a random value between 0 and 1. Then emits a sound if it is greater than 0.5, otherwise outputs its value.

- - - - -

See also other examples who are in the "Simple Programs" folder

# Program structure

The simplest programs have a first "initialization" section that ends with a STOP and then all the procedures that begin with LABEL and that will be executed only when they are called by a button or other events.

```
Button 1 Text "Button1"  
Button Button1 Color Yellow7
```

```
Stop
```

```
Label Button1  
    Beep  
Return
```

The simplest programs have a first "initialization" section that ends with a STOP and then all the procedures that begin with LABEL and that will be executed only when they are called by a button or other events.

```
Button 1 Text "Button1"  
Button Button1 Color Yellow7
```

```
Label MainLoop  
    Gosub CheckMotorTemperature  
    ' -----  
    Wait Seconds 0.1 ' <<< 0.02 sec. or more  
    ' -----  
Goto MainLoop
```

```
Stop
```

```
Label Button1  
    Beep  
Return
```

```
Label CheckMotorTemperature  
    ...  
    ...  
Return
```

The MainLoop must contain a Wait Seconds 0.02 instruction (0.02 is the minimum time to use) otherwise it is executed so fast that it leaves no time for other operations such as keyboard key or button execution.

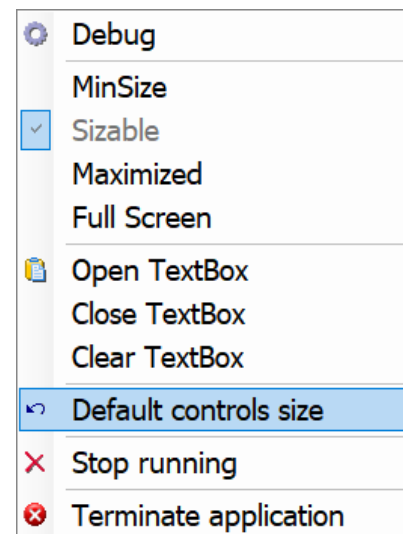
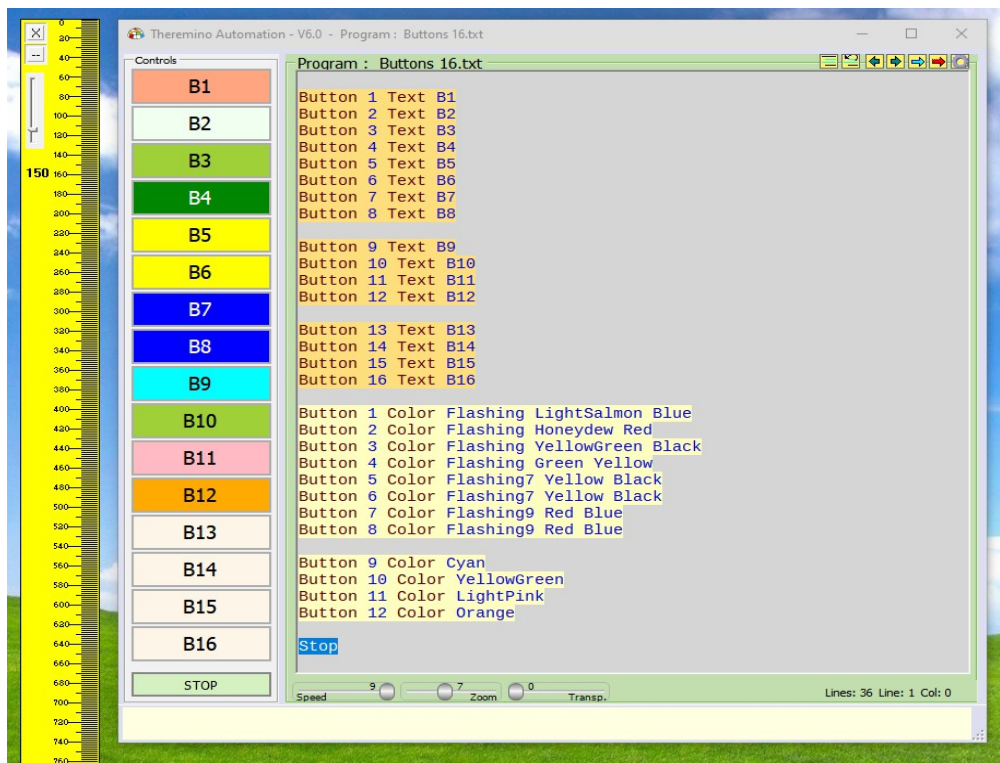
In the MainLoop, on the other hand, pauses that are too long (over 100 milliseconds) should not be used and slow procedures should not be called, otherwise the operations performed by the MainLoop are slowed down.

The events (Button, Keys and others) interrupt the MainLoop and therefore they too must be executed in the shortest possible time.

# Resize the application controls

By resizing the controls, up to 16 or 32 buttons can be displayed vertically, even on 1280 x 768 screens.

In other cases, the buttons could be enlarged horizontally to accommodate more characters, or you could enlarge or tighten all the controls to obtain a good readability depending on the resolution of the screen used.



To resize the controls:

- Place the cursor anywhere in the middle of the screen.
- Then press and hold down the CTRL and SHIFT keys.
- And finally, without pressing the mouse buttons, you move the Mouse in the four directions.
- Or you press the four arrows on the keyboard to enlarge or reduce the size of the controls, vertically or horizontally.

Not being able to use CTRL and SHIFT (for example on a tablet without keyboard), we recommend to close the Automation app. and change the "ControlsRatio\_W" and "ControlsRatio\_H" values in the "Theremino\_Automation\_INI.txt" file.

To restore the original dimensions of the controls, open the application menu and click on "Default controls size".



# Keywords

The keywords are only about twenty, easy to remember and easy to use.

Keywords: Beep Button Controls For Next Exit GoSub GoTo If Else EndIf Key Label Load Option PressKeys SendKeys Print Return Save Select Case CaseElse EndSelect Slot Stop End Variable Wait Window

The bottom bar of the application displays the keywords that can be used.

*Instructions can be quickly written by clicking with the mouse on the keywords of the lower bar.*

**Keywords: Please write a valid numerical expression, or the keyword: Input**

While writing, the bottom bar shows suggestions to complete the instructions properly.

**ERROR: The label "WaitMotorAndRecovery" is already declared.**

**Select next error**

Selecting error lines the "Select next error" button appears, then clicking on this button you select the next error lines.

- - - - -

Not all the errors are identified and explained, we are not Microsoft, we are few and our time is limited. However, the bottom bar is a good help for beginners, and each new version works a bit better.

- - - - -

On the following pages the keywords will be explained in detail.

Note that they are listed in order "almost" alphabet (some words are grouped together because they like each other).



# Keywords one by one

## Array

Arrays are available starting from Automation version 7.8 and simplify operations that would have been difficult to implement in previous versions.

They are numeric arrays, with predefined names, from Array1 to Array9.

Apart from these limitations, the potential of these Arrays is remarkable.

- Each Array can store up to ten million values.
- Very large integers can be used (from -9007199254740991 to 9007199254740991) without loss of precision.
- "Double precision" floating point numbers can be used that are even larger (from  $-1.7 * 10^{308}$  to  $+1.7 * 10^{308}$ ).

Furthermore, the sizing of Arrays is automatic and no errors are ever produced even if the index is wrong (less than zero or greater than the size of the array).

For example, if you write the following line, Array2 is automatically resized to 1000 places (from 0 to 999) and place 999 is updated with the value 1234.

```
Array2(999) = 1234
```

If Array2 previously contained 1000 places or more, then its size is not changed.

During resizing, the value of the elements present in the Array is not changed.

## ArrayClear

The size of Arrays can only increase automatically, but if necessary they can be emptied. For example, to clear Array1 you write: `ArrayClear(1)`

Or you can initialize it writing values separated with commas or spaces.

```
Array1 = 111, 222, 333, 444, 555 or else Array1 = 111, 222 333 444 , 555
```

In this case is the number of elements that determines the size of the array.

## ArrayLength function

This function provides the number of elements in an Array. For example, to print how many elements Array1 contains you write: `Print ArrayLength(1)`

## ArrayToString function

This function provides a string containing all the array elements separated by spaces. For example `Print ArrayToString(1)` could write `111 222 333 444 555`

## Beep

This statement gives the basic sound established by the operating system.

## Beep Frequency Duration

This command produces a pure sinusoidal sound.

- The valid frequencies are from 37 Hz to 32700 Hz
- The valid durations are from 1 millisecond to 2 000 000 000 of milliseconds.

Example that plays a 440 Hz for 1/10 second: `Beep 440 100`

## Beep "String of chars"

Example: `Beep "220 500, 440 50, 0 900, 440 50, 0 900, 440 50"`

Valid separators are: spaces, commas, minus sign, semicolon and newline.

Notice that pauses are specified setting to zero the frequency value.

You could also read a string from a file like in this example:

`Load s1 "HappyBirthday.txt"` and in the next line `Beep s1`

The Beep is executed asynchronously, to stop the sound before the end of the duration time, use another `Beep` command or a `Beep 0 0` command.

The system Beep is pure sinusoidal but it produces "ticks" if the duration is not a multiple of the cycles. It is possible to reduce these "ticks" changing experimentally the duration values, eventually using the provided example application.

- - - - -

To issue specific sounds, such as a siren or the click of the button, you can use the LOAD instruction (see how you are using the following pages).

To emit complex and adjustable sounds, you could use other applications (from theremino collection, for example [Audio Generator](#) or [Sound Player](#) or else [Theremin Synth](#), and controlled them by Automation with the SLOT statement.

To control external applications with automation you write the check values in the slot. For example for Audio Generator you could use the Slot one for the waveform, the two for the frequency, the three for the amplitude and the four for the Pan (position in stereo).

See also the examples located in the folder "Demo Programs / Demo Audio"

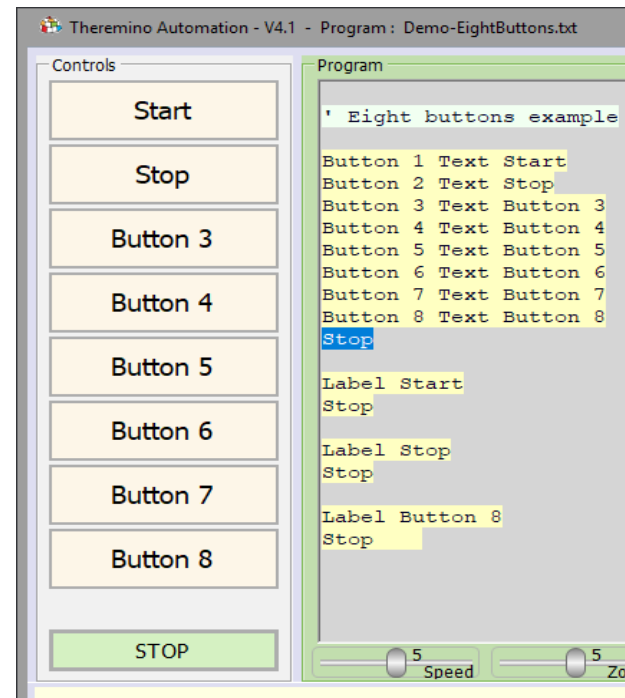
# Button

Buttons appear on the left side of the window and you can press them with the left mouse button, or a finger when using a touch screen.

You can bring up to 120 buttons, on multiple columns, and determine, for each of them, the text to display.

If there is a LABEL instruction with the corresponding text, then press the button, the execution will continue from the LABEL program online.

If there is no corresponding LABEL then the line is marked with an orange color and an error message appears in the lower bar.



Sometimes it might be useful to have buttons not associated with a function, for example to use them as labels.

In these cases, you can enclose the text with double quotes to eliminate error checking.

The jump to LABEL is GOSUB type. So, if you encounter a RETURN, the execution will resume from where it left off. Otherwise it will continue without returning.

If the text is more than one word then for the corresponding LABEL you can also use only the first word.

## Button - Identify the corresponding labels

Pressing a Button when the program is stopped, the program text scrolls to highlight the corresponding LABEL.

When the program is started you can press CTRL and then the button (even those disabled). To use this possibility, the **scroll must be disabled** (Scroll-enabled button).

## Button Text

Normally the word written after **Text** is used to associate the button with a function, but it can also be used as a label.

If the text is very long, it will use a smaller font and, if necessary, the text will be written on several lines. Anyhow, It is recommended to not use too long words or phrases in order to avoid issues while displaying text on buttons.

You could force the text to display over two rows with a CRLF, like in this example:  
**Button B1 Text "First row" + CRLF + "Second row"**

To make an invisible button write a blank text (only two double quotes). To make it visible but without text, you write a space between double quotes.

To view all the buttons the window should be high enough, otherwise late down buttons are not displayed.

## Button Disabled / Enabled

To disable a button you write: **Button <identifier> Disabled**

To enable a button you write: **Button <identifier> Enabled**

To view all the buttons the window should be high enough, otherwise late down buttons are not displayed.

## Button Slot

The SLOT statement connects the button at a Slot.

So, in this example: **Button <identifier> slot 6**, if the Slot value exceeds 500, then the button is pressed.

- - - - -

See also the examples in the folder  
"Demo Programs / BUTTONS"

# Button Color

The "Button Color" instruction colors the buttons to make them more meaningful and visible.

The colors can be chosen by writing their name, or by writing the initial letter and then clicking on the names that are proposed in the lower bar. But you can also press the "Color selector" button, and choose them by clicking with the mouse.

If you press the "Color selector" button with the right mouse button, a different selection panel opens.

Theremino Automation - V6.4 - Program : Buttons with colors.txt

Controls

Change Colors	B17
B2	B18
B3	B19
B4	B20
B5	B21
B6	B22
	B23
	B24

Program : Buttons with colors.txt

```
Button 1 Color Flashing Yellow9 Red7
Button 2 Color Honeydew
Button 3 Color Yellow
Button 4 Color Cyan
Button 5 Color YellowGreen
Button 6 Color LightPink
Button 7 Color Orange
Button 8 Color Light
Button 17 Color Lavender
Button 18 Color LavenderBlush
Button 19 Color LawnGreen
Button 20 Color LemonChiffon
Button 21 Color LightBlue
Button 22 Color LightCoral
Button 23 Color LightCyan
Button 24 Color LightGoldenrodYellow
Stop
```

Speed 9 Zoom 7 Transp. 0

Keywords: LightSeaGreen / LightSlateGray / LightSteelBlue / LightBlue / LightCoral / LightSalmon / LightPink / LightGreen / LightSkyBlue / LightCyan / LightGray / LightYellow /

Color selector

Color

Basic colors:

Red	Yellow	Green	Cyan	Blue	Pink	Gray
DarkRed	DarkYellow	DarkGreen	DarkCyan	DarkBlue	DarkPink	DarkGray
LightGray	White	Black	DarkBlue	DarkCyan	DarkMagenta	DarkRed

Custom colors:


Define Custom Colors >>

OK Cancel Add to Custom Colors

Color selector

mediumBlue	Blue	DarkSlateG...
purple	DarkMagenta	BlueViolet
mediumSla...	SlateBlue	RoyalBlue
adetBlue	LightSeaGr...	DarkCyan
eaGreen	DarkOlive...	Olive
addleBrown	Brown	Firebrick
ichsia	Magenta	IndianRed
mGray	Gray	SlateGray
ghtBlue	DarkSeaGr...	MediumAq...
arkKhaki	RosyBrown	PaleVioletR...
ghtCoral	Salmon	DarkSalmon
ghtSalmon	BurlyWood	Tan
ellow	Lime	SpringGreen

mediumSpr...	YellowGreen	Chartreuse	LawnGreen	GreenYellow	PaleGreen	LightGreen
LightSkyBlue	SkyBlue	Turquoise	Aqua	Cyan	Aquamarine	PaleTurqu...
PowderBlue	LightCyan	LightGray	Gainsboro	Lavender	Khaki	PaleGolden...
PeachPuff	NavajoWhite	Wheat	Bisque	Moccasin	MistyRose	BlanchedA...
PapayaWhip	AntiqueWh...	LavenderBl...	SeaShell	Linen	OldLace	Beige
Cornsilk	LightYellow	LightGolde...	LemonChif...	FloralWhite	Ivory	Snow
Azure	AliceBlue	Honeydew	MintCream	WhiteSmoke	GhostWhite	White
Red1	Green1	Blue1	Cyan1	Magenta1	Yellow1	Gray1
Red2	Green2	Blue2	Cyan2	Magenta2	Yellow2	Gray2
Red3	Green3	Blue3	Cyan3	Magenta3	Yellow3	Gray3
Red4	Green4	Blue4	Cyan4	Magenta4	Yellow4	Gray4
Red5	Green5	Blue5	Cyan5	Magenta5	Yellow5	Gray5
Red6	Green6	Blue6	Cyan6	Magenta6	Yellow6	Gray6
Red7	Green7	Blue7	Cyan7	Magenta7	Yellow7	Gray7
Red8	Green8	Blue8	Cyan8	Magenta8	Yellow8	Gray8
Red9	Green9	Blue9	Cyan9	Magenta9	Yellow9	Gray9

# Button Color Flashing and RGB

## Flashing colors

With the "Button Color Flashing" instruction you can set two colors and make them flash alternately. With Flashing1, Flashing2 ... up to Flashing9, you can also adjust the flashing speed from very slow to very fast.

Here is an example of how to set two colors that alternate quickly.

```
Button B1 Color Flashing7 Yellow9 Red7
```

## RGB colors

Instead of the names of the colors, you can also write the amount of Red, Green and Blue that compose them.

The syntax can be for example: `Button xxx Color R,G,B` where instead of R, G and B you write three numbers from 0 to 255.

Here are some examples:

```
Button B1 Color 255,0,0 ' This is a pure red
```

```
Button B1 Color 255,255,0 ' This is a yellow
```

```
Button B1 Color 200,200,200 ' This is a light yellow
```

## ARGB colors

If you add a fourth number to the RGB color then the color becomes ARGB type, with the Alpha (transparency) value in first place.

The first of the four values is transparency and ranges from 0 (completely transparent) to 255 (completely opaque) with everything in between.

Here are some examples:

```
Button B1 Color 20, 255, 0, 0 ' This is a very light red
```

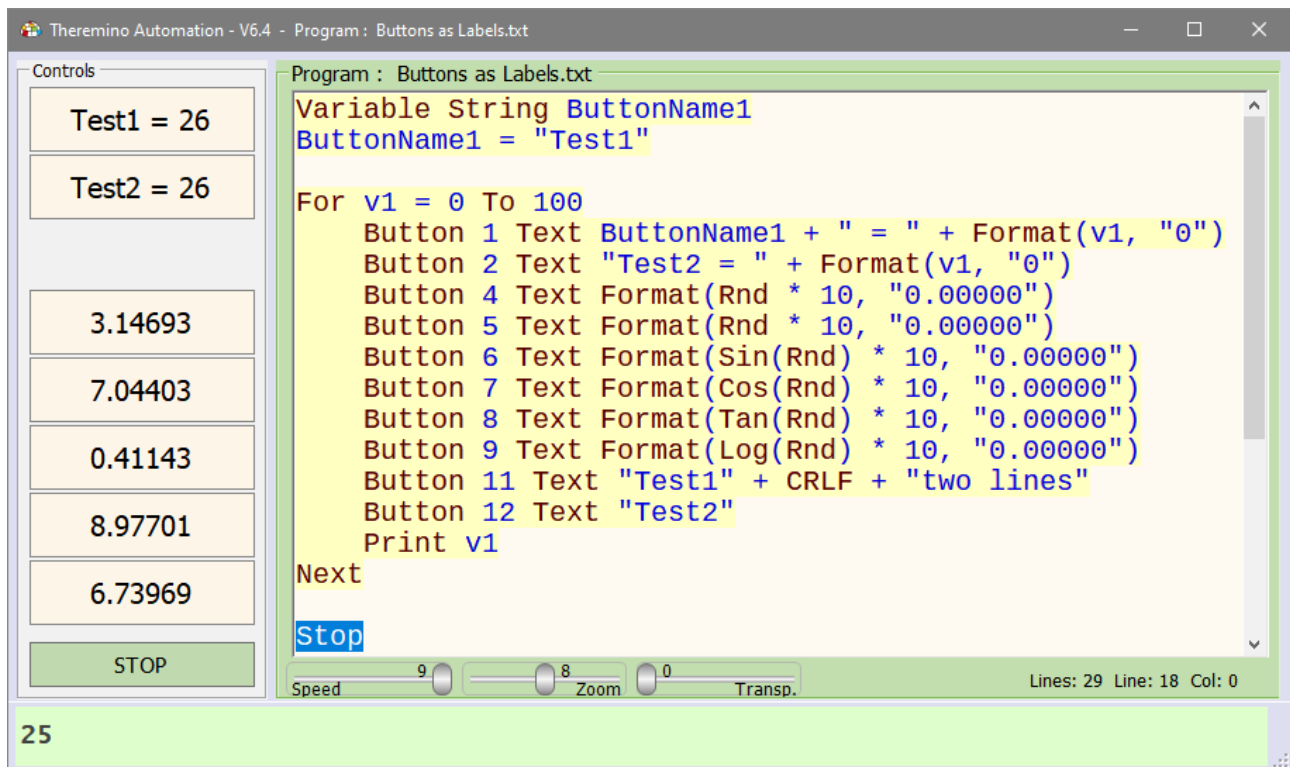
```
Button B1 Color 10, 255, 255, 0 ' This is a very light yellow
```

- - - -

There are examples for Buttons with Images and Flashing, RGB and ARGB colors in the files "Button IMAGES.txt", "Buttons Flashing.txt" and "Button RGB COLORS.txt" found in the "\Demo Programs\BUTTONS" folder

## Button used as Labels

The "Buttons" can also be used to display numerical values or text messages.



The text of the buttons is updated quickly, so you can modify it at various points in the program to show the evolution of numerical values or to change the text when the button is pressed.

It is important to note that the buttons can also be used to call program functions. For this purpose, it is enough that the first word (before the first space) is equal to the word of the Label.

If a Button is not used to call a Label, double quotation marks should be used to prevent the line from being treated as an error.

For example, pressing the first two buttons of the image on this page will call the functions starting with **Label Test1** and **Label Test2**, even if the following text (= 26) is modified.

You can also show colorful and flashing messages, to highlight them.

See also the examples in the folder  
"Demo Programs / BUTTONS"



## Button Text and Identifiers

When using Buttons as labels with a text that can be varied, you must declare the button by an identifier (a text without spaces or quotes) and later change its text with a string.

This is also useful to identify two different buttons sharing the same initial word, otherwise the program would confuse them while choosing which to execute.

Some examples:

```
Button 1 Label Function1
Button Function1 Text "Run function 1"

Button 2 Label Function2
Button Function2 Text " Run function 2"
Button Function2 Color White
Button Function2 Disabled
```

```
Label Function1
...
Return
```

```
Label Function2
...
Return
```

In this example the first row assigns the identifier `Function1` to button 1.

This identifier is then used instead of the button number in order to change its text with a string (as in the second row in the example) or to call a corresponding label that execute a part of the program. By this way there is no danger of ambiguity, even if the text will be changed after.

Even a Button declared by using a string (text closed by double quotation) can have the same role, but can be a source of ambiguity in some cases.

The most important rule to remember is that a Button will be identified in the entire program through its identifier or the first word of a string that are used during the **first** declaration.

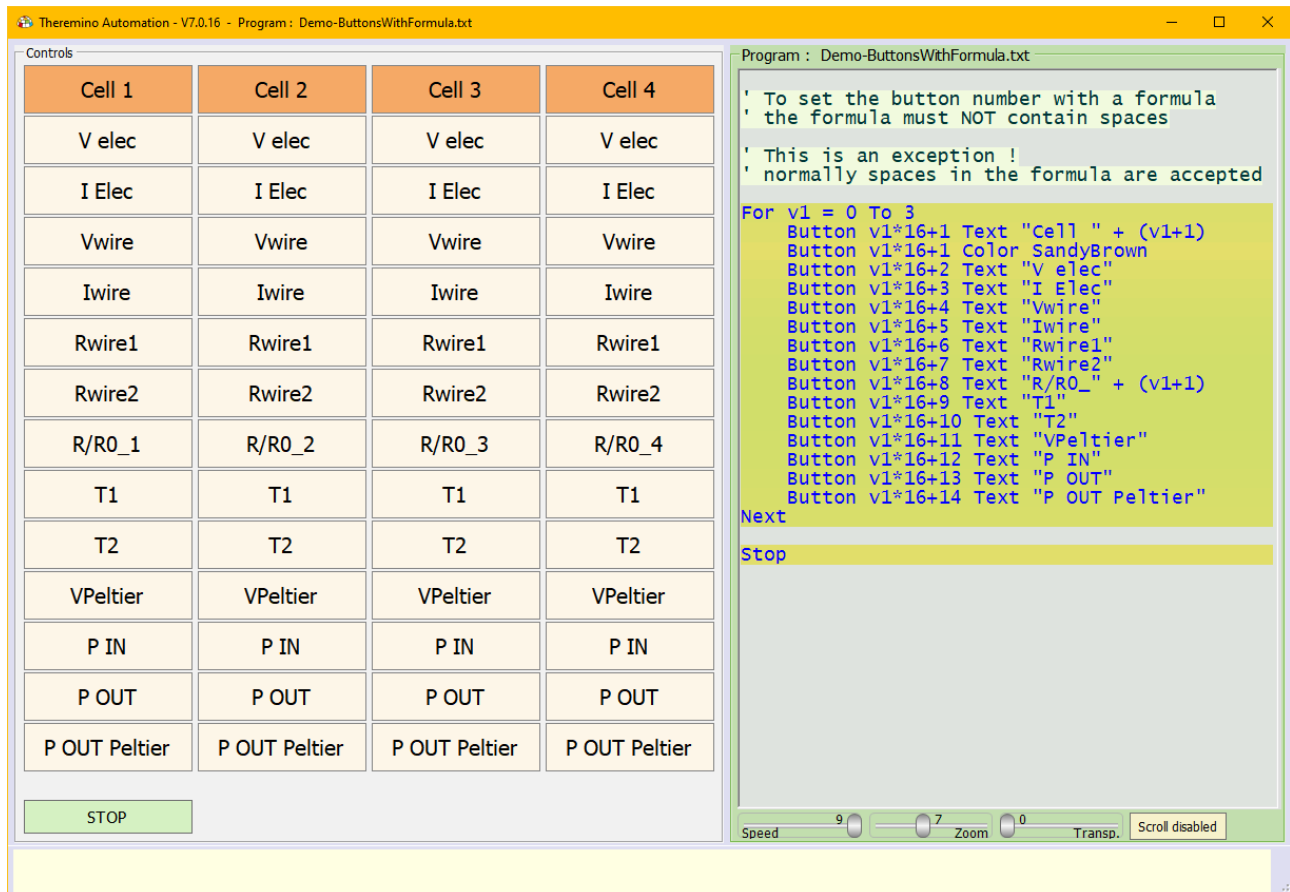
If you write `Button nn Text xxx` the word `Text` will be automatically substituted with `Label` when the program is reloaded.



## Button - Using formula instead of numbers

Normally you write a number for each button and in this way you can move them easily without changing anything else in the program.

In some cases it might be useful to write a formula in place of the number and enclose everything in a FOR loop as seen in the following image.



Note that in this case the formulas must not contain spaces. But this exception only applies to indicating Button numbers. Normally formulas can also contain spaces between terms.

-----

See also the example:

"Demo Programs \ BUTTONS \ ButtonsWithFormula.txt"

## Button that change state

The following example creates a button that turns ON and OFF a LED.

Each time the button is pressed the button text and color is changed, and the Slot 1 value is modified to control the LED luminosity.

```
' ----- INITIALIZATIONS
Variable Numeric LED = 1
Button 1 Text LedToggle
Gosub LedOFF
Stop

' ----- FUNCTIONS
Label LedToggle
    If Slot(LED) = 0
        Gosub LedON
    Else
        Gosub LedOFF
    EndIf
Return

Label LedON
    Slot LED = 1000
    Button LedToggle Text "LED OFF"
    Button LedToggle Color PowderBlue
Return

Label LedOFF
    Slot LED = 0
    Button LedToggle Text "LED ON"
    Button LedToggle Color Cyan
Return
```

With this structure you can also call the three functions from various parts of the program (with `Gosub LedToggle`, `Gosub LedON` and `Gosub LedOFF`) and in all cases the text and the colors of the Button are updated correctly.

If desired, it is also possible to expand this mechanism to a greater number of states. In this case, the "Toggle" function will be called "Change" and inside it, some "Case" statements will be used to pass from one state to the next.

-----

Experiment with the examples "ButtonBistable" and "ButtonMultipleStates"  
in the folder "Demo Programs \ Demo COM Port"

# Buttons with images

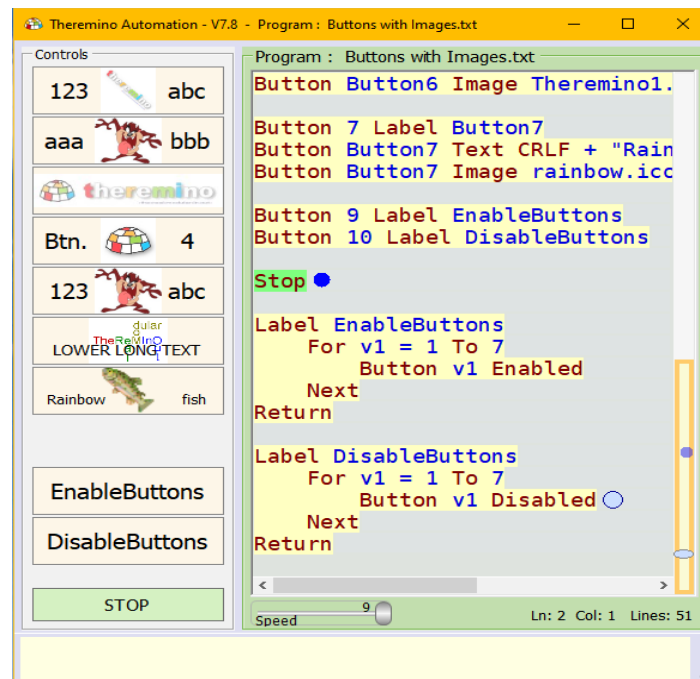
To assign images to buttons, you write, for example:

```
Button 1 Label Button1  
Button Button1 Text "123 abc"  
Button Button1 Image Image1.jpg
```

In this case the image is a JPG but it could also be a PNG, ICO, GIF, BMP, or even other formats that can be recognized by the operating system.

The images must be in the "Media" folder that you find near the "Automation.exe" file.

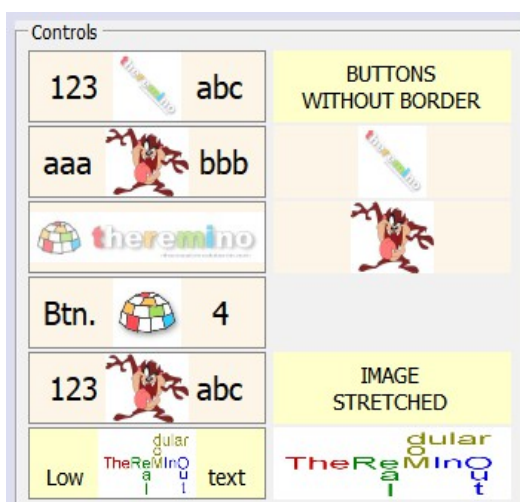
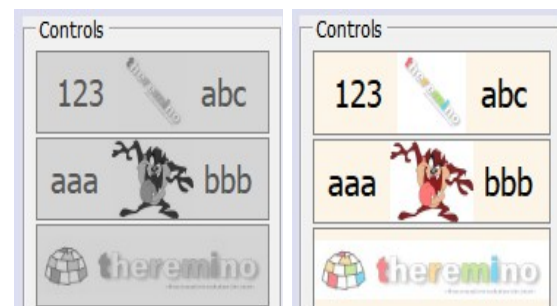
If they are in a sub-folder you must add the missing part of the path.



You can also use rectangular images with the left part white and position the text on the left. Or images with the right part white and use repeated spaces to push the text to the right.

When you disable the buttons the images are automatically transformed into gray-scale images.

This helps to identify the enabled buttons and distinguish them better from the disabled ones.



When setting images you can also remove the black border by writing "Image0":

```
Button Button1 Image0 Image1.jpg
```

Or you can make it thicker by writing **Image2**, **Image3**, **Image4** or **Image5**.

And finally you can also stretch the image to the edges by writing **Image0S** (zero means no border and S means "Stretched")

See the example "Demo Programs \ BUTTONS \ Button IMAGES.txt"

## COM (serial port) - Commands

**COM Open 1 9600** or else **COM Open COM1 9600**

These command opens the serial port "COM1" at the speed of 9600 baud. Instead of 9600, any speed from 1 baud up to many mega baud can be used, up to maximum for the connected hardware.

To find out the names of valid ports, use the **COM\_Portnames** function explained on the next page.

To find out if the port is actually open, use the **COM\_Status** function explained on the next page.

**COM Close**

This command closes the serial port. You can use it even if the door is not open, to make sure it is closed.

**COM WriteString s1**

**COM WriteLine s1**

These two commands send the string s1 to the serial port.

The former only sends the string, while the latter adds a final CRLF.

**COM WriteStringHEX s1** for example "AA AA FF 80 00"

**COM WriteStringDEC s1** for example "170 170 255 128 0"

These two commands send many bytes specified with Hex or Decimal numbers. The numbers must be separated by spaces (not comma or other separators) and the only valid chars are "0" to "9" and "A" to "F".

**COM DiscardOutBuffer**

**COM DiscardInBuffer**

These commands clear the serial port's output and input buffers. Emptying the buffers can be useful in some cases, to clear the past history when starting a new communication session.

**COM EnableDTR**

**COM EnableRTS**

**COM DisableDTR**

**COM DisableRTS**

These controls raise and lower the DTR and RTS outputs.

Some devices only respond if these two outputs are in a certain state.

- - - - -

See also the examples in the folder  
"Demo Programs \ Demo COM Port"

## COM (serial port) - Functions

### `S1 = COM_Status`

In this example the variable S1 is "OPENED" if the "Open" command could open the port, or "CLOSED", if the port has not been opened.

The reasons why you are unable to open the port, could be that the required port number does not exist on the device, or that it is currently used by another application.

### `S1 = COM_Portnames`

In this example the variable S1 is filled with the names of the usable ports separated by spaces.

Here is an example of a string provided by this function `"COM1 COM12 COM13"`

Interestingly, placing the cursor on the line `S1 = COM_Portnames` the ports are displayed, even without running the program. This is a convenient behavior that can be used for all functions that set a variable.

To count the number of ports inside this string and to choose one of the names, you can use the **GetSeparatedStringCount** and **GetSeparatedString** functions which are explained in [this page](#).

### `COM_Dsr`

### `COM_Cts`

These functions read the DSR and CTS signals status. The result is Boolean so you can check them directly with `If COM_Dsr` or with `If COM_Dsr = True` or with `If COM_Dsr = False`

Remember that assigning boolean values to numerical variables then you must test them for ZERO (0 = False) or for NOT ZERO (-1 = True).

-----

See also the examples in the folder  
"Demo Programs \ Demo COM Port"

## COM (serial port) - Receive buffer

### `S1 = COM_Received`

This example read a line of characters arriving from the serial port.

Individual characters are not received, but only complete lines. Reception occurs only when the transmitting device sends an end-of-line character.

The end of the line can be specified by any of the following character combinations: **CR** (13), **LF** (10), **CRLF** (13-10), or **LF CR** (10-13).

The received string does not contain the end of line characters.

### `S1 = COM_ReceiveUntil(string)`

This example reads a number of characters arriving from the serial port.

The end of the line can be specified by the termination characters, which are supplied in the "string" parameter.

A single character or multiple characters can be used as the termination.

The received string also contains the termination characters.

You can use the `Chr ()` function to specify non-printable terminator characters (less than 32 or greater than 127).

For example, to specify the characters zero and 255 as the terminator, you could write: `Chr (0) + Chr (255)`

### `S1 = COM_ReceiveNChars (nn)`

This example reads a number of characters arriving from the serial port.

The number of characters to be received is specified by the numeric value "nn"

If the serial buffer contains no characters or if it contains fewer characters than required then this function returns an empty string.

- - - - -

The received character string can be scanned with the `Mid` function and converted in various ways, for example, the following lines convert the characters to hexadecimal values and print: `01 03 50 80 BF FF`

```
s1 = Chr(1) + Chr(3) + Chr(80) + Chr(128) + Chr(191) + Chr(255)
For v1 = 1 To Len(s1)
    s2 = s2 + Right("0" + Hex(Asc(Mid(s1, v1))), 2) + " "
Next
Print s2
```

## Controls

Probably in future versions we will add other controls, but currently there is only a command to set the mouse cursor position and a TextBox, that can be used to display many lines of text with the Print command.

The "Controls" statements are:

```
Controls SetCursorPos <X Y>
```

```
Controls SetBackColor <color>
```

```
Controls OpenTextBox
```

```
Controls CloseTextBox
```

```
Controls ClearTextBox
```

In the following sections they are individually explained.

### Controls - SetCursorPos <X Y>

This instruction sets the screen mouse cursor position.

The X and Y numerical values are in pixels, therefore, on the lower left the two values are zero, while on the upper right the two values depend on the number of pixels on the screen.

In case of multiple screens the pixels start from the bottom left of the first screen to the top right of the last screen and if the main screen is not the first then the numbers that identify the pixels can also be negative.

You can also read the mouse cursor position with the function MouseXP and MouseYP explained [in this page](#).

See also the examples in the folder  
"Demo Programs \ Demo Mouse"

### Controls - SetBackColor <color>

This instruction sets the buttons box back-color with a selected color or with a color specified by RGB values.

You can also choose the color with the "Color selector" button and if you press it with the right mouse button a different selection panel opens.

# Controls - TextBox

The following commands are used to control the TextBox:

## Controls OpenTextBox

This command opens the TextBox and all subsequent "Print" will be displayed in the TextBox, instead of in the lower status line.

When the TextBox is open, you can resize it by moving the horizontal gray bar that divides it from the program area with the left mouse button (see the image on the next page).

## Controls CloseTextBox

This command closes the TextBox and all subsequent "Print" will be displayed again in the lower status line.

## Controls ClearTextBox

This command deletes all the text in the TextBox.

You can also delete the text by sending a CLS or a Chr (12) with the Print statement, as in the following two examples:

```
Print CLS
```

```
Print Chr(12)
```

The length of the TextBox is limited to ten thousand characters to avoid slowdowns. When this size is exceeded, the characters are automatically deleted from the top of the TextBox and replaced with a warning line.

- - - - -

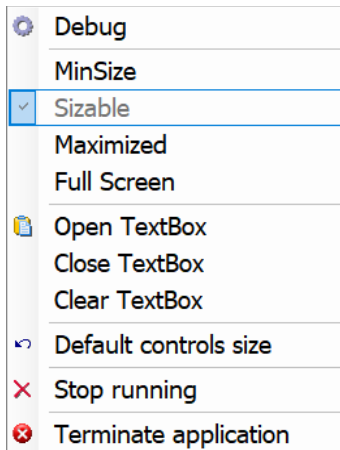
See also the examples in the folder  
"Demo Programs \ Demo TextBox"



## Controls - Close and resize the TextBox

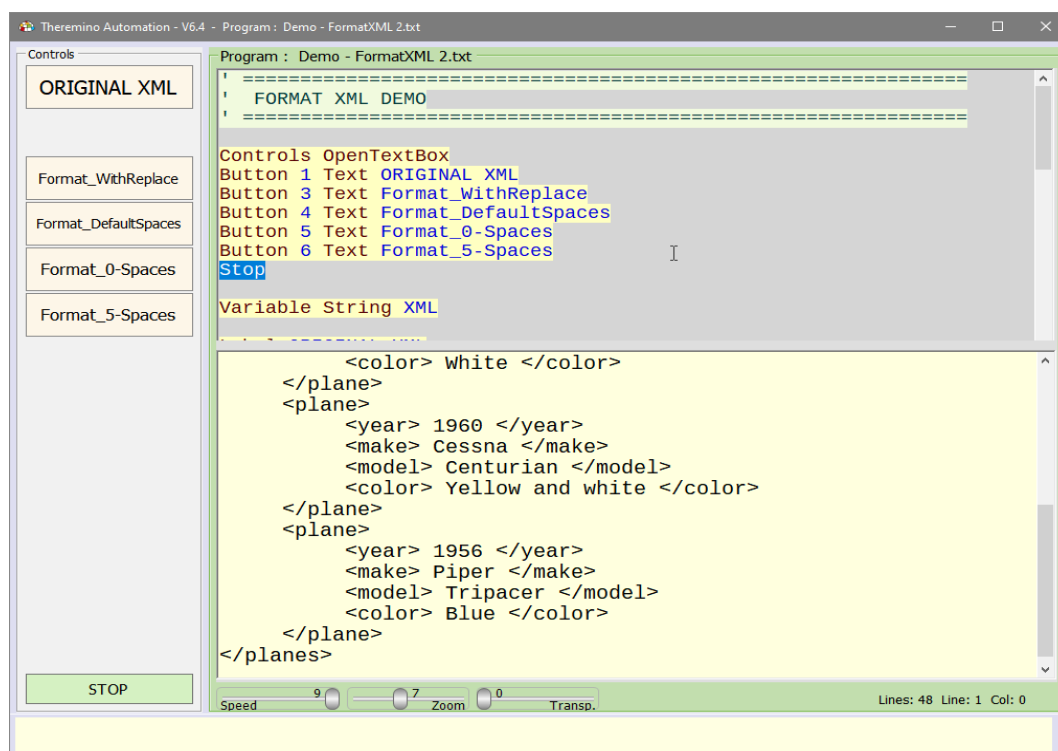
The TextBox opens executing a **Controls OpenTextBox** statement. To close it automatically when the program stops, you could write the statement **Controls CloseTextBox** in the closing event, as you see in the following example:

```
Label EventStop
      Controls CloseTextBox
End
```



Normally you control the TextBox by means of program instructions, but in some cases it might be useful to control it manually.

Pressing the right mouse button on the left side of the application opens a contextual menu that allows you to Open, Close and Clear the TextBox even when the program is stopped.



To resize the TextBox (seen in the lower half of this image), move the mouse with the horizontal gray bar that divides it from the program area.

## For - Next

These two statements repeatedly execute the lines between FOR and NEXT.

The FOR instruction, which must be before the NEXT, is composed of a numeric variable (**counter**), an initial numeric value, and a final numeric value.

```
For counter = initial To Final
---
---
Next
```

When the FOR is executed arriving from the previous instructions, the "**counter**" variable is set with the "**initial**" value.

After having executed the instructions that are on the lines following the FOR, the program comes to NEXT. At this point, if the variable "**counter**" is different from the value "**final**", then the program execution returns to the FOR.

When repeating the FOR the variable "**counter**" is incremented or decremented by one, depending on whether the value "**final**" is greater than or less than it. The increment direction is automatic. To specify the direction you add the word "**Step**", explained in the next page.

If for some reason the variable "**counter**" exceeds the "**final**" value (for example because you used not integer values, or for rounding errors), then it would be set to the value "**final**". In this way it is ensured that the last loop is always executed with the "**final**" value.

- - - - -

The "**initial**" and "**final**" values may be numeric variables, or numeric constant values, or even expressions, as in the following examples:

```
For var1 = var2 To var3
For var1 = 7 To 3
For var1 = var2 + 7 To "var3 - 4 + int (rnd * 6)"
```

*Important to note that the expressions must not contain spaces, otherwise the line turns red and the FOR statement is not executed.*

*You can write complex expressions, which also contain spaces and functions, enclosing them in quotation marks, as shown in the right part of the last example.*

## For - Next with Step

Normally for each FOR and NEXT cycle, the control variable is increased or decreased by one unit.

By adding the word **Step** and a numerical expression it is possible to impose an increase of a fraction of a unit, or even a larger increment, as in the following examples..

```
For v1 = 1 To 9 Step 0.1
  ---
Next
```

```
For v1 = 0 To 32000 Step 1000
  ---
Next
```

In these examples we used **v1** as control variable, and numbers as initial and final values, but of course we can also use user-defined variables and complex expressions.

-----

Using **Step** the direction of the increment is no longer automatic but explicitly specified, as in the following two examples.

```
For v1 = 9 To 1 Step -0.1
  ---
Next
```

```
For v1 = 32000 To 0 Step 1000
  ---
Next
```

If you specify start and end values with variables, the **For** may increment in the unexpected direction. In the example below which produces three short sounds, we would expect that changing **"abc"** to **" "** would cause the **For** not to be executed, but unexpectedly it plays the values one and zero and then emit two sounds.

```
s1 = "abc"
For v1 = 1 To Len(s1)
  Beep 1000 500
Next
```

In these cases it is good to specify the **Step** and establish the increment direction.

## For - Next instead of While

In other languages, many types of loops are used, for example While-Wend, Do-While, Do-Loop, Loop-Until, Loop-While, etc.

These multiple ways of writing loops all do exactly the same thing, so we'll only use one, the For-Next.

The For-Next is undoubtedly the most versatile and complete method to build a repetition cycle and can also produce infinite cycles if you set as final value the number 9e99 (or "Infinity"), as in the following examples:

```
For v1 = 1 To 9e99
  ---
  ---
Next
```

To create infinite loops you can use the "Infinity" function instead of 9e99. To terminate the loop, the control variable can be set to the value "Infinity", or the word "Exit" could be used, as explained on the following pages.

```
For v1 = 1 To Infinity
  If Slot(1) > 500
    v1 = Infinity
  EndIf
Next
```

The For-Next can also be nested and you can easily exit even the innermost For-Next cycles with the Exit statement, as in the following example:

```
For v8 = 1 To 10
  For v7 = 1 To 30
    For v6 = 1 To 50
      ---
      If Slot(1) > 500
        Exit 3
      EndIf
    Next
  Next
Next
```

The next two pages explain how to use the Exit statement

## Exit

This instruction must be placed in a For-Next loop and causes the execution of the program to jump after the **Next** instruction.

Here is an example of premature exit from a For-Next

```
For v1 = 1 To 1000
  If v1 >= 10
    Exit
  EndIf
Next
```

Without the **Exit** instruction this cycle would repeat itself a thousand times, but the If instruction ends the cycle when **v1** reaches 10.

In other cases, instead of checking **v1**, you could exit when a key is pressed, for example "ESC", or if too much time has passed.

-----

This example shows that For-Next loops can also be nested.

```
For v1 = 0 To 1000
  If v1 >= 10
    Exit
  EndIf
  For v2 = 0 To 1000
    If v2 >= 20
      Exit
    EndIf
  Next
Next
```

In the case of nested For cycles, each Exit instruction acts on the innermost cycle.

So in this example:

- ◆ The first **Exit** refers to the **For v1** and therefore exits the final **Next**.
- ◆ The second **Exit** refers to the **For v2** and so only exits from its **Next**.

-----

*The Exit statement only takes effect if inserted into a For-Next loop. If it is positioned outside, the statement is highlighted in orange and is not performed.*

*See also the examples in the folder: "Demo Programs \ Demo FOR NEXT"*

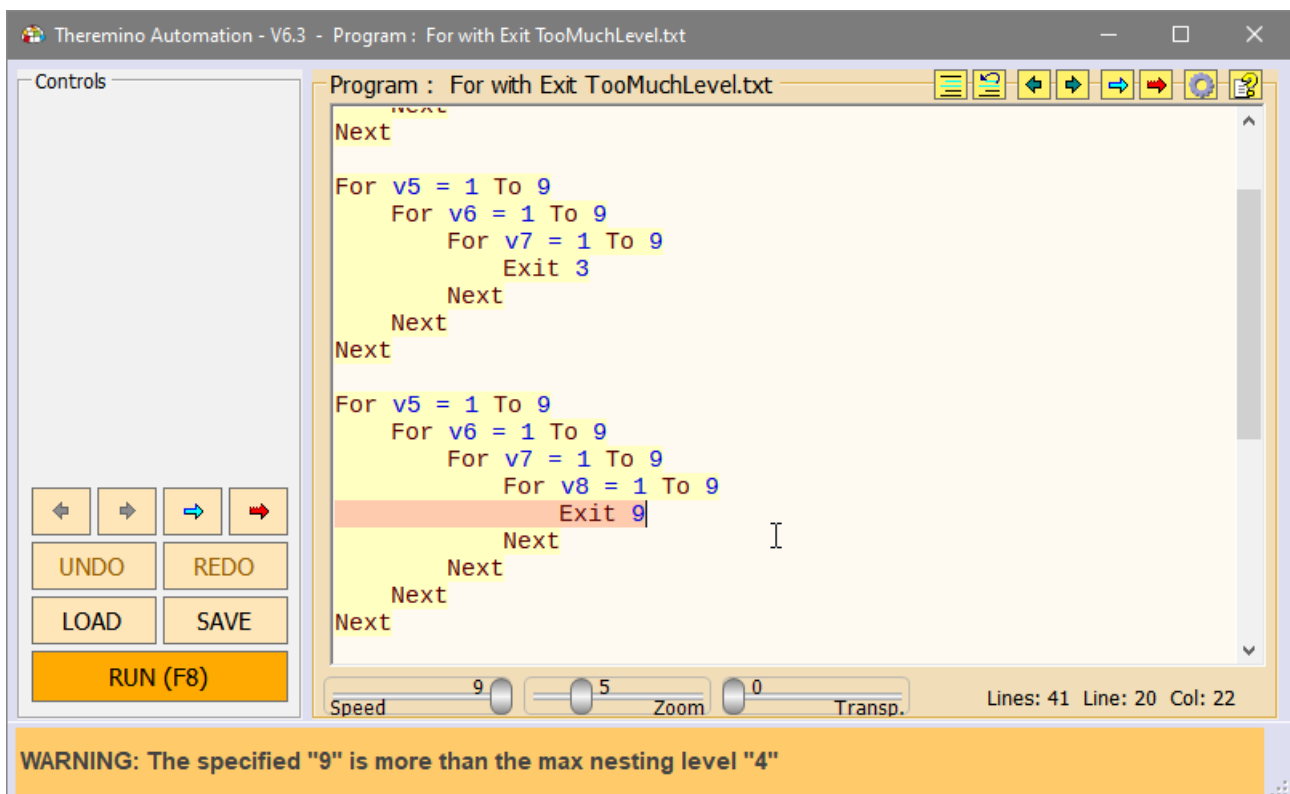
## Exit "n"

This instruction allows you to exit multiple For-Next, nested one inside the other.

```
For v5 = 0 To 9
  For v6 = 0 To 9
    Exit 2
  Next
Next
```

```
For v5 = 1 To 9
  For v6 = 1 To 9
    For v7 = 1 To 9
      For v8 = 1 To 9
        Exit 4
      Next
    Next
  Next
Next
Next
```

If you specify a number greater than the For-Next maximum nesting level, then the Exit statement is not executed and you are warned of the error with a message as in the following image:



## Goto - Gosub - Return

The GOTO and GOSUB statements blasting the execution of the program to the mark with a LABEL (label) and with an identifying text, composed of alphanumeric characters.

- So the instruction `Goto xxx` jumps to the line indicated by `Label xxx`.
- And the instruction `Gosub yyy` jumps to the line indicated by `Label yyy`.

If there is no corresponding label, then the GOTO and GOSUB statements have no effect and the execution continues as if there were.

The identifier can also be composed of several words, and these words can also be enclosed in double quotes (for clarity), for example, you could write `Label xxx yyy zzz`, or `Label "xxx yyy zzz"`.

### Difference between Goto and Gosub

To put it in simple words, the GOTO is like a one-way trip, while the GOSUB also provides for the return.

So if you use a GOTO, the program jumps and continues from that position. While if you use a GOSUB, the program jumps, performs some instructions and, when it encounters a RETURN statement, returns to the statement after the GOSUB. In this case the Stop instruction.

```
Gosub Identifier1  
Stop
```

```
Label Identifier1  
...  
Return
```

See also the examples in the folder  
"Demo Programs \ GOSUB and RETURN"

## Eliminate Gosubs and send parameters to Functions

Starting from Automation version 7.x, you can also omit all the Gosubs and send parameters to function too.

SEE PAGES : [Calling functions](#) and [Function parameters](#)

## If - Else - Endif

The "IF" and "ENDIF" instructions enclose an area to be executed only if the condition is true.

Each "IF" must end its "ENDIF" otherwise does not run and the execution continues as if there was not.

```
If Slot (1)> 500
    Instructions to execute
    only if the value of the slots 1
    is greater than 500
endif
```

The instructions "IF", "ELSE" and "ENDIF" contain two zones, one to run if the condition is true, and the other to run if the condition is false.

```
If Slot (1)> 500
    Instructions to execute
    only if the value of the slots 1
    is greater than 500
else
    Instructions to execute
    if it is less or equal than 500
endif
```

The IF statements (and all the other Automation structures) can be "nested" that is placed one inside the other. For example in the following example the BEEP is performed if all three slots are larger than 500.

```
If Slot (1)> 500
    If Slot (2)> 500
        If Slot (3)> 500
            Beep
        EndIf
    EndIf
EndIf
```

But the same result you could get even with AND instructions

```
If Slot (1)> 500 And Slot (2)> 500 And Slot (3)> 500
    Beep
EndIf
```

-----

See also the examples in the folder "Demo IF ELSE ENDIF" folder



# Key

With the keyword "Key" is obtained by the execution of program parts, pressing keys on the keyboard.

The usable keys are as follows:

1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num0, Num1, Num2, Num3, Num4, Num5, Num6, Num7, Num8, Num9, Left, Right, Up, Down, Space, Esc, PageUp, PageDown, Home, End, Canc, Back, Enter, Ins, Shift, Ctrl, Alt, Print, Scroll, Pause

The KEY statements usually are written at the beginning of the program and can be of two types GOTO or GOSUB. With GOTO The jump and never comes back. With GOSUB the program jumps back when encounters a RETURN.

```
Key 1 Goto Identifier1
Key 2 Gosub Identifier1
Stop
```

```
Label Identifier1
...
Stop
```

Starting from Automation version 7.x, you can also omit all the Gosub word.

When you press a key on the keyboard, the program is stopped, whatever he was doing, and jumps to the corresponding LABEL.

If you have set a KEY with RETURN, then the program is interrupted, jumps to executing lines after the LABEL, then meets the RETURN, and go back to doing what he was doing when he was interrupted.

To avoid accidentally trigger it while you write something, the Key do not work when you browse pages on the web, or when you are writing in an input box, or when you are writing in another application. Or in all cases when the application Automation is not selected, or is invisible, or is minimized.

If you want the Key statement to act always, you can use the "Option GlobalKeys Enabled" statement.

- - - - -

See also the examples in the "Demo Keys" and "Demo Option" folders

## Label

The LABEL instruction (label) is a marker, used to give a unique name to a program line.

The label may consist of any number of alphanumeric characters, but not spaces or double quotes, as in the following examples:

```
Label 1
Label 123
Start Label
Label Stop
Label MyFunction
Label My_function
```

The line labeled with the LABEL instruction, may be the target of the jump instructions: GOTO, GOSUB, and KEY BUTTON, as in the following examples:

```
GoTo MyFunction
GoSub MyFunction
Key 1 GoTo MyFunction
Key 1 GoSub MyFunction
Button 1 Text MyFunction
```

See also the examples in folders

"Demo GOSUB and RETURN" and "Demo Keys"

- - - - -

## Eliminate Gosubs and send parameters to Functions

Starting from Automation version 7.x, you can also omit all the Gosubs and send parameters to function too.

SEE PAGES : [Calling functions](#) and [Function parameters](#)

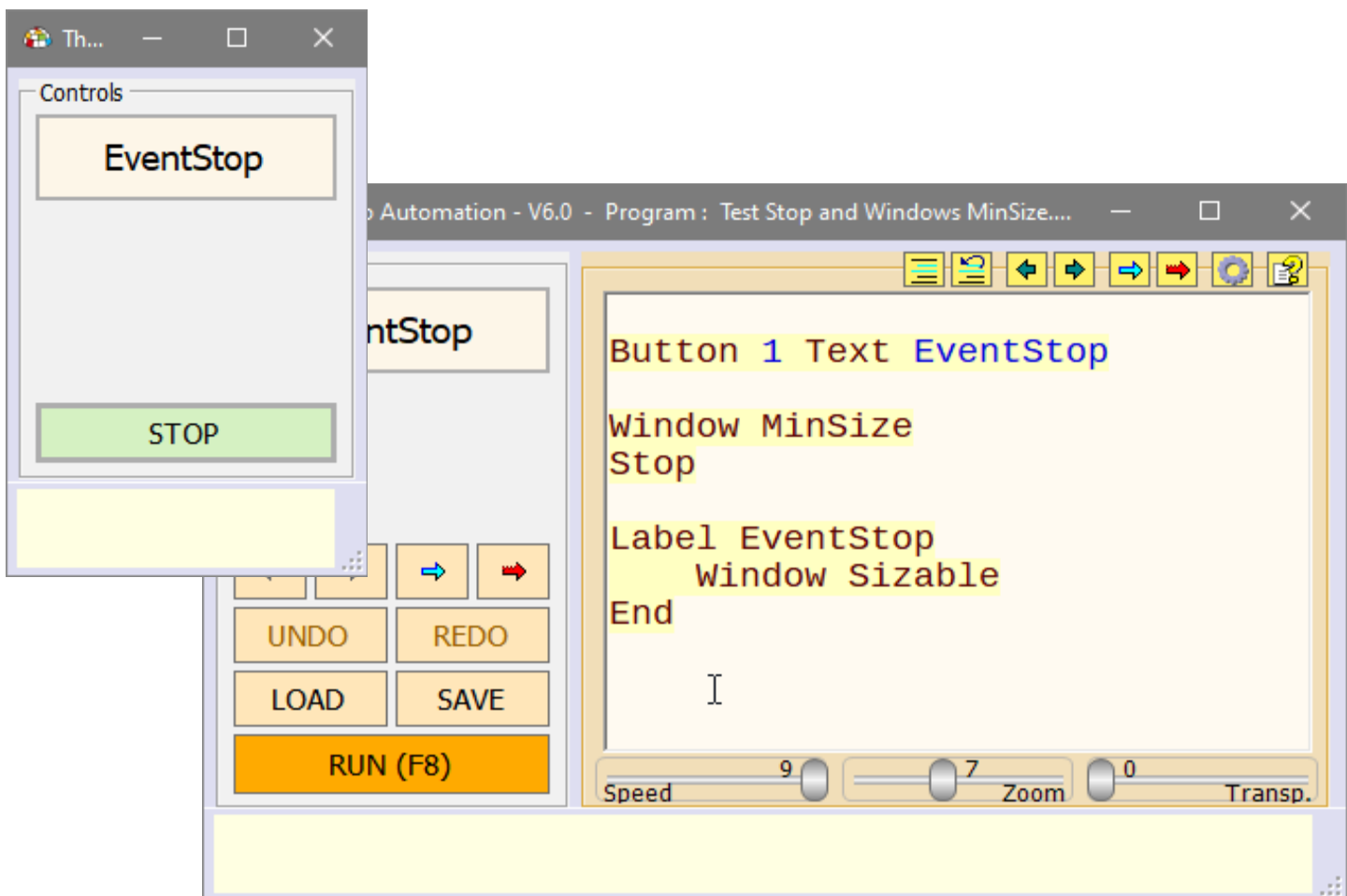
## Label EventStop

The instruction **Label EventStop** is called automatically each time the program is stopped (either manually, by pressing the Stop button, or with the **End** instruction).

This Label is a special event, but you can also call it with **Gosub EventStop**, **Button EventStop** or **Key EventStop**, as you would any other Label. In this case the content of the Label is executed and then the program stops because the EventStop Label always ends with **End**.

This event allows you to complete important actions every time the program stops, for example cutting power to the motors, but you can use it usefully in many ways.

In the following example, every time the program is started, the window is minimized with the **Window MinSize** command. Then, when the program is stopped, the **Label EventStop** is called which returns the window to its normal size, with the **Window Sizable** command.



*See the examples found in the "Demo Programs \ Demo EventStop" folder*

# Label EventTimer

## CAUTION

The **Label EventTimer** is executed by a timer, and randomly interrupts the program in many different places. Unwanted interactions can therefore occur, which are difficult to understand and correct.

Also, if running the EventTimer takes too long, some parts of the main program may be significantly slowed down.

So use this instruction very carefully and carefully checking that it does not produce side effects on the rest of the program.

- - - - -

## INTERACTIONS WITH DEBUG

The constant interruptions make it difficult to follow the progress of the program. So we recommend commenting the entire function, from **EventTimer** to **Return**, when using Debug functions.

- - - - -

The **Label EventTimer** is automatically called about ten times per second.

The instructions found between the **EventTimer** and its final **Return** are executed at maximum speed, regardless of the speed set with the Speed slider. This minimizes side effects on the rest of the program.

However, it is recommended to observe the following rules for everything between the **Label EventTimer** and its final **Return**:

- ◆ Minimize the number of instructions.
- ◆ Possibly do not use slow instructions such as Print or Button Text.
- ◆ Never use very slow instructions such as Load.
- ◆ Do not use any type of waiting (Wait Seconds / Wait Button ...).
- ◆ Do not call other labels (especially if they contain a lot of code).
- ◆ Set Speed at maximum (9) to speed up execution of the main program. This prevents execution from slowing down too much or stopping due to continuous interruptions.

# Label Event\_DroppedFile

The `Label Event_DroppedFile` is used to receive the name and content of a file that is dragged and dropped onto the command pane with the mouse, with an operation called Drag-And-Drop.

For this event to work, you must have defined the variables to read, `DroppedFilename` e `DroppedString`.

If you do not define at least one of these two variables, then the line that must receive the `Label Event_DroppedFile` turns red and gives an error.

If you want to read both variables, you define both of them, otherwise you just define the one you want to use.

After dragging the file, the `Label Event_DroppedFile` is called and the two variables contain the following text strings:

- `DroppedFilename` contains the full path to the file plus the name and also the extension.
- `DroppedString` contains the full text of the file. If the text is not readable, you should find a leading word that explains that the file is of a special type, for example a PNG or JPG image.

- - - - -

Here is a short example of a program that receives and prints the file name and its contents.

```
String DroppedFilename
String DroppedString
Stop

Label Event_DroppedFile
    Print DroppedFilename + CRLF + DroppedString
Return
```

You can try this example by loading it from the Demo Programs \ Files folder.

Then just RUN the example and drag any file from a Windows folder to the "Controls" area.

## Label Event\_ExternalCommands

The `Label Event_ExternalCommands` is used to receive commands from the `Theremino_COBOT` application, but could also receive commands from any other application capable of writing in the "Text Slots".

When an external application writes a string containing a command in the `Slot_ExternalCommands` the event is raised by calling this `Label`.

The Automation program is interrupted, whatever it was doing, and the instructions after the `Label` are executed until the `Return`.

The command text is read with the `"CommandText"` function and then decoded with a `"Select"` structure, as in the following example:

```
Variable Numeric Slot_ExternalCommands = 53
Stop

Label Event_ExternalCommands
  Select CommandText
    Case "Beep"
      ExecBeep
      ClearCommand
    ,
    Case "Beep multiple"
      ClearCommand
      ExecBeepMultiple
    ,
    CaseElse
      Print "Unrecognized command: " + CommandText
      ClearCommand
  EndSelect
Return

Label ExecBeep
  Beep 440 300
  Wait Seconds 0.5
Return

Label ExecBeepMultiple
  For v1 = 1 To 3
    Beep 880 400
    Wait Seconds 0.5
  Next
Return

Label ClearCommand
  SlotText(Slot_ExternalCommands) = ""
Return
```

Notice that a `ClearCommand` statement was added in the `Case "Beep multiple"` **before** the command execution. This statement tells the COBOT application to proceed immediately, without waiting for the command execution to finish.

On the next page there is an example of a sequence that sends these commands.

## Commands from COBOT to Automation

This chapter explains how to send commands from the COBOT application to the Automation application.

The following example is a sequence that moves three motors and between one movement and the next sends the commands "Beep" and "Beep multiple".

```
MoveTolerance 1
MoveTime 0.5
MoveTo +00137.000 +00300.000 +00022.000
SendCommand Beep
MoveTo +00153.000 +00303.000 +00025.000
SendCommand Beep multiple
```

The file that contains this sequence is run by the Theremino\_COBOT application and must be in its "Sequences" folder.

The commands sent to the Automation application can be any text string, even separated by spaces, and are decoded regardless of case.

Any multiple TAB or Space characters are transformed into single spaces by the COBOT application before sending them, and the leading and trailing TABs and spaces are deleted.

The user himself can write his commands, remembering that:

- The commands must be written in the sequence of the COBOT application.
- Instructions for decoding them in Automation's ExternalCommands event.

In the Cobot application, the text slots to be used for the commands must be set. Open the options panel (with the gear at the top right) and locate the "Slots for Text-Commands" section located at the bottom.

The text Slots that are set in the Cobot application must be the same that are declared in the Automation variables with the names: **Slot\_CommandsToCobot**, **Slot\_CobotResponses** e **Slot\_ExternalCommands**

See also the examples in the folder  
"Demo Programs \ SlotText Commands"

- - - - -

In the next page we will see that commands can also be sent from the Automation application to the Cobot application.

# Commands from Automation to COBOT

A slightly different method is used to send commands to the Cobot application. You set the **Slot\_CommandsToCobot** and **Slot\_CobotResponses** variables, and then use them with the commands to write and read in the text Slots.

## Commands from Automation to COBOT ( text strings in the Slot\_CommandsToCobot )

- **ExitFromEdit** ( exit from editing state )
- **StartExecution** ( auto-exit from editing state )
- **StopExecution** ( auto-exit from editing state )
- **GotoHome** ( auto-exit from editing state )
- **GotoCenter** ( auto-exit from editing state )
- **EnableRepeat**
- **DisableRepeat**
- **SelectLine nnn**
- **ExecuteSelectedLine**
- **EnableCollaborative**
- **DisableCollaborative**
- **AddNewPosition**
- **SetHomePosition**
- **SetCenterPosition**
- **SetHoldingTorque nnn**
- **SetSafeTorqueLimit nnn**
- **SetTorque nnn**
- **SetAcceleration nnn**
- **SetSpeed nnn**
- **LoadSequence SeqName.txt** ( the file name "xxxxx.seq" must not contain spaces )

## Responses from Cobot to Automation ( text strings in the Slot\_ResponsesFromCobot )

- **Ready**
- **Execution running at line nnn**
- **Motors disconnected**
- **Human detected - Speed reduced**
- **Human too near - Execution stopped**
- **Too much torque - Execution stopped**
- **Motors disconnected - Execution stopped**

See the example "Commands\_to\_COBOT.txt" in the folder  
"Demo Programs \ SlotText Commands"



## External commands to Automation Buttons

You can also execute Automation Buttons with commands from external applications.

This method is easier to use than the method explained in the previous pages which expects the Label **Event\_ExternalCommands**

You simply send the text that appears on a button (or its identifier) and the button executes as if you pressed it with the mouse.

- - - - -

To use this method you need to set the command slot using, for example, a line like this: **Variable Numeric Slot\_ExternalCommands = 1**

But **you shouldn't** set the **Label Event\_ExternalCommands**

- - - - -

### OPERATION

All the commands that will be sent in the **Slot\_ExternalCommands** will be compared, case insensitive, with the text present on the buttons and also with the identifiers that were used in the first creation of the buttons, and if they match then the button is pressed.

We also recommend using single spaces between words, in the text that appears on the Buttons to be executed and also to use identifiers without spaces, because some applications such as "VoiceCommands" always send single spaces.

- - - - -

**IMPORTANT** - To direct external commands to the Buttons  
You must NOT set the **Label Event\_ExternalCommands**

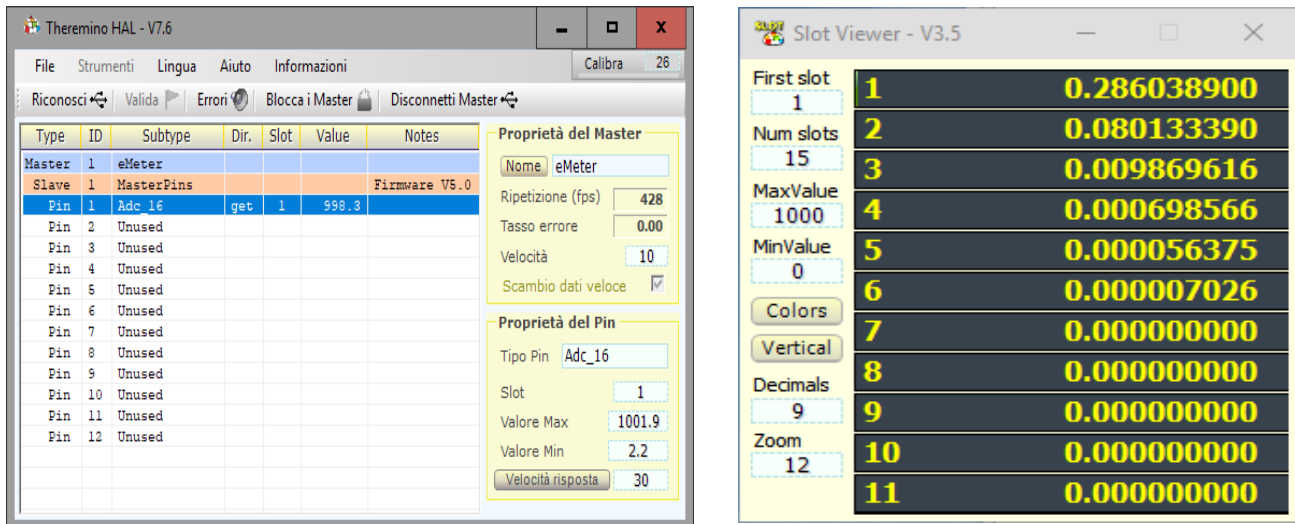
If you set the **Label Event\_ExternalCommands**  
then the execution of external commands must be performed inside it  
and the Buttons will no longer be executed.

Try the **SlotText-ExternalCommands\_TO-BUTTONS.txt** example  
sending him the commands on SlotText 1 with Theremino\_Terminal  
or with a second copy of Automation.

## Load (Applications)

In some cases it may be useful to start an application as many times as you run Automation. Often this application is Theremino\_HAL or SlotViewer, or even both.

If the application is already active it is not launched a second time.



The Load instruction stops the program execution until the application is fully started.

The applications "xxxx.exe" file should be in the folder "Apps". You could also give the full path, but should be in a fixed place in the system (not in the Automation folder), or by moving the Load Automation stop working.

If the application to launch is located in a sub-folder, you can add the part of the path you need, such as:

**Load Apps\Theremino\_HAL.exe**

If there are spaces in the path, you must add quotes

**Load "Apps\Theremino\_HAL.exe"**

If the application to launch is in a parent folder, you can go up a folder with the colon. Eg:

**Load "..\Theremino\_HAL\Theremino\_HAL.exe"**

- - - - -

If you specify only the file name (for example "Theremino\_SlotViewer.exe") then the application is searched in the folder "Apps" and also in all its sub-folders.

## Load OpenApps

This instruction starts all applications starting with "Theremino\_" and ending with ".exe".

Applications are launched if they are in the "Apps" folder and in all its sub-folders.

## Load CloseApps

After having opened many applications, you may need to close them. The Load CloseApps function closes all the applications that was opened with "Load".

### NOTICE

When closing Automation by the red cross  
all the opened apps are automatically closed  
(unless you are working on the source in "Debug" mode)

## Load CloseAll

*This command closes all the apps, as the precedent command, but then closes Theremino Automation too.*

### SUGGESTION

*To avoid closing Automation  
hold down CTRL while executing the "Load CloseAll" statement.*

### NOTICE

When closing Automation by the red cross  
all the opened apps are automatically closed  
(unless you are working on the source in "Debug" mode)

## Load CloseApps file-name

To close a single application after the "Load CloseApps" statement you also write the filename, as in this example:

```
Load CloseApps C:\Theremino_HAL\Theremino_HAL.exe
```

If you do not specify the complete path then the path is rebuilt starting from the position where the executable "Theremino\_Automation.exe" is located, you can then write its name as in the following line and if the file is identified then its process will be closed.

```
Load CloseApps Theremino_HAL\Theremino_HAL.exe
```

## Load CloseApps process-name

*To close an application you could also write the name of the "process" which is the name of the executable file of the application not preceded by the path. You write the name with or without the final exe extension, as in the following examples:*

```
Load CloseApps Theremino_Editor.exe
```

```
Load CloseApps Theremino_Editor
```

*Closing applications using the name of the process can be useful when you do not know where the executable resides, such as in the case of Theremino\_Editor which is never started directly, but is started by the Windows system when you open a file that has been associated with him.*

***The process name must have exact uppercase and lowercase letters.***

## Load CloseApps Windows

### WARNING

**This command turns off your PC !**

This example shuts down the Windows operating system:

```
Load CloseApps Windows
```

## Load Slots / Save Slots

The **Load Slots** instruction loads all the slot values (from slot 0 to slot 999), by reading them from the "\_Slots.txt" file (numeric Slots only, not the SlotText).

The "\_Slots.txt" file contains the values of all the Slots, which had been previously saved with the instruction **Save Slots**.

With **Load Slots n1 n2** you load only from Slot n1 to Slot n2, included.

Instead of n1 and n2 you can also write variables, or formulas, also complex. Note that the formulas of Load Slots instructions must not contain spaces, or give an error. To use spaces in these formulas you might enclose them with double quotes. Here are some examples:

<b>Load Slots 20 30</b>	slots from 20 to 30 inclusive
<b>Load Slots v1 v2</b>	slots from 10 to 20 inclusive
<b>Load Slots 2*3 2+v2</b>	slots from 6 to 22 inclusive
<b>Load Slots 2 Math_PI*4</b>	slots from 2 to 13 inclusive

In the examples it is assumed that v1 is equal to 10, and v2 which is equal to 20.

In the last example the PI (Math\_PI function with value 3.14159...), is multiplied by 4, and then rounded to the nearest integer (13).

## Load Vars / Save Vars

With the **Save Vars** statement you save the variables used in the "\_Vars.txt" file.

With **Load Vars** you reload the variables that are used and present in the file.

Using **Save Vars** in the event **EventStop** and **Load Vars** in the initial part of the program, you can restore the total state of the application that will restart exactly as it was closed.

### WARNING

When using **Load Vars** you need to be careful where you initialize your variables and how you initialize them.

Read the next page that explains how to use **Load Vars**

# Load Vars - Beware of Initializations

Executing the **Load Vars** command could overwrite your initializations, creating unexpected, strange, and counterintuitive behavior.

To avoid problems, it is best to place the Load Vars statement in the first lines of the application and write all the initializations immediately after.

Initializations that must not be modified must be in the initial area, after **Load Vars** and before **Stop**.

```
' -----  
Load Vars  
' ----- Not affected by Load Vars  
Array1 = 0.5,1,2,5,10,20,50,100,200  
Array2 = 0,12,20,50,100,200,500,1000  
InitLog  
Numeric SoundFrequency = 1000  
Numeric SoundMillisec = 10  
' ----- LOAD APPS  
Load Theremino_SlotViewer.exe  
' ----- BUTTONS  
Button 1 Label TestOnOff  
Button 2 Label TestFrequency  
' ----- START ALL  
OpenComPort  
UpdateButtons  
Loop  
Stop  
  
Label InitLog  
    String LogFile = ".\LOGS\LOG_" + Format(Now, "yyyy_MM_dd_HH_mm") + ".txt"  
    Numeric LogOldTime = -Infinity  
Return
```

In this example, the areas highlighted in yellow are the initializations that should not be covered by Load Vars.

Note that the initializations placed in the InitLog function are also valid since the InitLog call is in the valid area.

This way, if necessary, you can change the initializations and they will not be covered by what is reloaded by Load Vars.

-----

See also the examples in the "Demo Programs\Vars\" folder

The "SaveVars\_Test.txt" example tests Save and Load with all types of variables.

While the "SaveAndRestoreVars.txt" example shows a more complex method that is used to save the variables and restore them using a file of your choice.

## Load (Images, Videos and Sounds)

With the LOAD instruction you can upload images, videos and sounds, they must be in the "Media" folder. If they are in a sub-folder you must add the sub-folder name, as in this example: `Load "Sound\Ringer.mp3"`

Image files can have the following extensions:  
`jpg jpeg bmp png exif tiff svg`

Video files can have the following extensions:  
`avi wmv mpg mpeg m2ts m3u mp4 m4v mp4v 3g2 3gp2 3gp 3gpp mov gif`

The sound files (and music) may have the following extensions:  
`mp3 wav m4a aif aifc aiff au snd aac adt adts flac mid midi rmi`  
*(if the file is missing you will hear a short "beep" in its place)*

Here are some examples of statements that load these files

`Load Image1.jpg`

`Load Video1.avi`

`Load "Science Picnic 2014.mp3"`

Before to load the audio files, you could set the volume with the instruction:

`Load Volume nn` (with nn from 0 to 100)

The `Volume` instruction must be before the `Load` instruction and has effect on all the audio files, excluding the .mid e .midi files.

- - - - -

After loading the video and audio, you could use the following instructions:

`Load Hide` become invisible images and videos (the program reappears).

`Load Stop` stops the execution of the video and audio.

`Load Pause` pause the playing of the video and audio files.

`Load Position Sec` changes the position (in seconds) of video and audio files.

`Load Play` restarts the video and audio files that where paused.

- - - - -

See also the ["Media" functions](#) page, and the examples:  
"Demo-ImagesVideoSounds.txt" in the "Demo LOAD and MEDIA functions" folder.

And the two examples showing file names composed with strings and numbers:

"Demo - BM\_Foundation1.txt" and "Demo - BM\_Foundation2.txt"

## Load (txt, pdf, doc, etc...)

With the Load statement instead of loading and viewing files with automation, you can instruct the Windows system to open any type of file. In this case Windows will open the file with the program that has been predefined for its extension.

So usually if it is a ".txt" it will be opened with Notepad, if it is a ".rtf" with WordPad, if it is a ".doc" with Word or with OpenOffice and if it is a ".pdf" with Google Chrome or with a PDF-Reader.

### Warning

Do not use Load to load a file from disk into a string!

To do this you must use the GetTextFile() function

as in this example `String str = GetTextFile("MyFile.txt")`

- - - - -

If you Load a ".txt" file located in the Programs folder. then the file is loaded as an automation program. If it is in another folder, or starts with "Files \", or has an extension other than txt, then it is loaded with the default Windows program for its extension.

If the file does not have a full path then it is loaded from the "Files" folder.

To use a relative path it must be taken into account that the starting folder is always "Programs". The following example shows how it is possible to indicate that the "Example1.pdf" file is in the "Files" folder:

`Load "..\Files\Esempio1.pdf"`

In this example the initial colon means: "go up one folder".

- - - - -

Image, video and music files (with extensions .jpg .bmp .png .gif .avi .wmv .mpg .mp4 .mov .mp3 and .wav), are loaded by automation only if they are located in the "Media" folder".

If they are not in the "Media" folder, then they will be opened by the Windows operating system with the default program for their extension.



## Load (Program)

With the Load statement you can load an automation program by choosing it among those who are present in the automation folder "Programs".

The name of the program to be loaded must be enclosed in double quotes.

The program file to load must be located in subfolders of the application Automation, starting from the "Programs" folder.

The program file to upload must end with the extension ".txt"

Here's an example:

```
Load "Examples\Demo LOAD PROGRAM\Demo-LoadProgram_2.txt"
```

The program that is uploaded will replace the current program and will continue running immediately, starting with the first line of the newly loaded program.

See the examples in the folder  
"Demo Programs \ Demo LOAD PROGRAM"

## Load (Variable from File)

With the Load instruction you can load an entire text file into a variable, which can be one of the predefined ones (from s1 to s99) or an user-defined "string" variable.

The name of the file to be loaded in the variable, could be enclosed in double quotes, or even without quotes. The file must have ".txt" extension and must be in the same folder as the program, or in the "Files" folder.

Here's an example: `Load s1 SetupFile.txt`

After loading from file, the variable can be separated into lines and fields, using the GetSeparatedStringCount and GetSeparatedString functions.

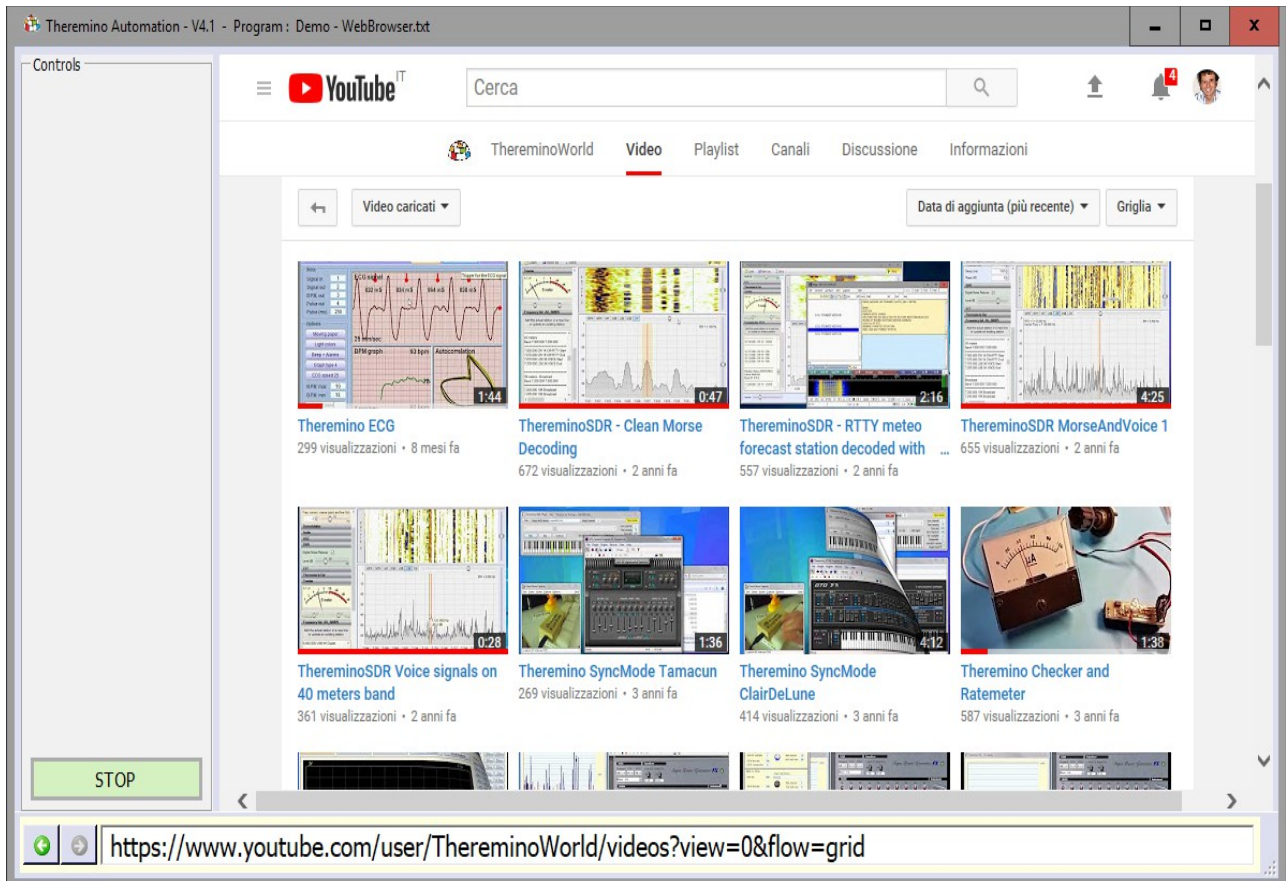
See the examples in the folder  
"Demo Programs \ Demo LOAD TXT files"

## Load (web address)

The LOAD instruction, followed by a web address, opens a web page.

Example: `Load http://www.google.com`

When writing the addresses in the LOAD instruction, the initial part `http://` it's necessary.



In this image you see a YouTube page, with videos of Theremino system.

Once you open a page it is possible to navigate anywhere on the web, using links found on web pages.

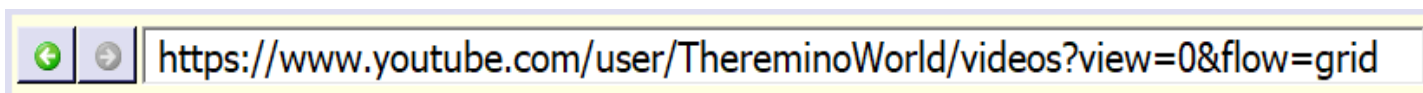
Alternatively, you can write a new address in the bar at the bottom, as explained on the next page.

# Load (options for web pages)

## The address bar

Writing an address in this bar, you can navigate to a new web site. When writing in this bar, you can also omit the `http://` that precedes the address.

After writing the address, to begin the navigation to the new site, you must press ENTER.



The two buttons on the left are used to open the previous or the next page.

## Zoom the page in and out



In some cases it may be useful to magnify the page, for easier reading of the characters.

In other cases it may be useful to reduce its size, to avoid having to use the sliders too often.

To zoom the web page, you can use keyboard shortcuts, or the mouse wheel.

First of all you must **click on the web page to be sure it is selected**, otherwise the following commands will not work.

- ◆ Press Ctrl and "+" key, to enlarge the page.
- ◆ Press CTRL and the "-" button, to reduce the size of the page.
- ◆ Press CTRL and the ZERO button, to reset the normal magnification.
- ◆ Press CTRL and scroll the mouse wheel, to change the magnification.

# The "Option" statements

Instructions starting with the word "Option" change the overall behavior of the program. Currently there are only three instructions, but we will probably add more in future versions. To take effect, they must be executed at each program start, so they are usually written in the initial area, before the "Stop" instruction. However, you can also change them later, during program execution.

## Option GlobalKeys Enable

Normally the Key statement and the Key () function only work if the Automation application is selected and is not minimized. But if you execute this instruction then the keyboard keys always act.

If you enable this option you will have to be careful not to use the keyboard for other applications, otherwise you could run commands by mistake.

## Option GlobalKeys Disable

After enabling the GlobalKeys option you may want to disable it while the program is running.

This instruction restores normal operation with the keyboard keys that act only if the Automation application is selected and is not minimized.

## Option Speed n

Normally you adjust the program execution speed with the "Speed" slider, explained in [This Page](#). But in some cases you can write the Speed instruction in the same program, and in this case the Speed slider is set from program too.

The number "n" can vary from 1 to 9 and is the same number indicated by the Speed slider.

## Option Transparency n

Instead of using the "Transparency" slider, explained on [This Page](#), you can program the transparency by setting an "n" number from 0 to 9.

See also the examples in the "Demo Programs \ Demo Option" folder

# PressKeys

This instruction sends characters and commands, as pressing the keyboard keys or pressing and releasing the Mouse buttons.

All letters, numbers and commands sent with PressKeys must be separated with a space, as in this example:

```
PressKeys A B C D Enter
```

To send spaces, use the "Space" command, as in this example:

```
PressKeys A B Space C D Enter
```

Spaces separate the letters from each other and also allow you to send commands.

Here is the complete command and symbol list:

```
Tab Esc Space Enter Return Left Right Up Down Backspace Pause
CapsLock PageUp PageDown End Home PrintScreen Insert Delete
ScrollLock NumMul NumAdd NumSub NumDiv NumLock NumDelete Num0 Num1
Num2 Num3 Num4 Num5 Num6 Num7 Num8 Num9 f1 f2 f3 f4 f5 f6 f7 f8 f9
f10 f11 f12 : ; = , < > _ . ? * - / ~ ` [ { \ | ] } ' a b c d e f
g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 LButtonDOWN
LButtonUP MButtonDOWN MButtonUP RButtonDOWN RButtonUP
```

The following commands are modifiers that are executed before any other character. All characters in the line will be executed with the modifier key pressed.

```
SHIFT CTRL ALT WINDOWS
```

Here are two examples that run the C and V keys while holding CONTROL

```
PressKeys CTRL C
```

```
PressKeys CTRL V
```

Between a PressKeys and the next it may be necessary to add a pause (Wait) to allow time for Windows to execute the previous command.

Here is an example that opens the Windows-x menu, and then the command window and finally executes a DIR command.

```
PressKeys WINDOWS x
```

```
Wait Seconds 0.5
```

```
PressKeys i
```

```
Wait Seconds 0.5
```

```
PressKeys d i r Enter
```

Note that this example depends on system language and on menu containing "Windows PowerShell". Also note that the x after WINDOWS must be lowercase, otherwise it would be sent as SHIFT-X and the menu would not open.

The PressKeys command is not very reliable, it can depend on the system language and execution times. For more information see [this page](#) and [this page](#)

PressKeys can be used with a variable containing the characters, for example you could write: `s1 = "d i r Enter"` and then: `SendKeys s1`

## SendKeys

This instruction is similar to the previous "PressKeys" but has some advantages and also some disadvantages.

The main advantage is that you can write the text to be sent, without separating the letters with spaces. On the other hand, the commands must be enclosed with braces, as in this example: `SendKeys ABCD{ENTER}`

There are also other subtle differences that make this command more comfortable in some cases but more uncomfortable in others. For example with SendKeys you cannot open the Windows-X menu of the example on the previous page.

The commands are also slightly different, here is a complete list:

```
{TAB} {SHIFT} {CTRL} {ALT} {WINDOWS} {ESC} {ENTER} {LEFT} {RIGHT}
{UP} {DOWN} {BACKSPACE} {BREAK} {CAPSLOCK} {PGUP} {PGDN} {END}
{HOME} {PRTSC} {INSERT} {DELETE} {SCROLLLOCK} {NUMLOCK} {HELP}
{ADD} {SUBTRACT} {MULTIPLY} {DIVIDE} {F1} {f2} {f3} {f4} {f5} {f6}
{f7} {f8} {f9} {f10} {f11} {f12} : ; = , < > _ . ? * - / ~ `
[ { \ | ] } ' a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6
7 8 9
```

To specify that any combination of SHIFT, CTRL, and ALT should be held down while several other keys are pressed, enclose the code for those keys in parentheses.

For example, to specify to hold down SHIFT while E and C are pressed, use "+(EC)". To specify to hold down SHIFT while E is pressed, followed by C without SHIFT, use "+EC".

Instead of the three control characters (+ ^ %) you could use the {SHIFT}{CTRL}{ALT} modifiers. Anyway the three characters (+ ^ %) are reserved as modifiers and you can not use them. So to send a "+" you have to write {ADD}

- - - - -

To specify repeating keys, use the form {key number}. You must put a space between key and number. For example, {LEFT 42} means press the LEFT ARROW key 42 times; {h 10} means press H 10 times.

- - - - -

The SendKeys command is not very reliable, it can depend on the system language and execution times. For more information see [this page](#).

SendKeys can be used with a variable containing the characters, for example you could write: `s1 = ABCD{ENTER}` and then: `SendKeys s1`



# Print

This statement prints, on the application bottom line, a numerical value or a character string.

After the word PRINT you can write any formula, even complex, composed with numerical values and character strings.

Examples:

`Print "Donald Duck"` result: `Donald Duck`

`S1 = "PI ="`

`S2 = Format (Math.PI, "")`

`Print S1 + S2` result: `PI = 3.14159265358979`

`Print "Duck" = "Cat"` result: `False`

`Print Mid("Duck", 4, 1) = "k"` result: `True`

`Print CLS` result: All the text cleared

Usually you use this instruction to check the the program operation, to know the intermediate values of the calculations and to highlight that certain lines have been executed. *To have a larger area for the text to be printed, you could use the [Controls OpenTextBox](#) statement.*

## Insert spaces in the strings

Sometimes it may be useful to insert many spaces in a string. Other times you could add special spaces that do not allow display and printing devices to wrap at that point.

To insert spaces in the strings, the following methods can be used:

- ◆ Enclose the string with double quotes and insert the spaces in the string.
- ◆ Use the `PadLeft`, `PadRight` and `Chars` functions.
- ◆ Use the `NBSP` constant. (Non Breaking Spaces)
- ◆ Insert `NBSP` spaces in the string itself with **ALT 255** (press ALT and the key sequence 2 5 5 on the numeric keypad, and finally release the ALT key).

- - - - -

To experiment with the Print instruction use the example "Demo TextBox"

# Save

## Save slot and variables

The instruction **Save Slots** saves the values of all slots (from 0 to 999), in the "\_Slots.txt" file (valid for numeric Slots only, not the SlotText).

The instruction **Save Vars** saves all the predefined variables values (v1..v99 and s1..s99), in the file "\_Vars.txt"

More info in the pages [Load/Save Slots](#) and [Load/Save Vars](#)

## Append some text to a file

The instruction **Save StringToFile s1 s2** appends the text contained in the variable s1, to the file whose name is specified by s2.

The file name must also include the extension, usually ".txt". Instead of s1 and s2 you can use user-defined variables or even write directly text strings (enclosing the text with double quotes).

If the path is missing then the path becomes the "Files" folder, which is next to the "Automation.exe" file.

Files are normally written as Unicode (two bytes per character) so they can also contain Chinese characters. To write a file in ASCII (one byte for each character) you must precede each string you write with the text "<ASCII>".

Also be careful not to "append" ASCII lines to a pre-existing file of the Unicode type, in which case delete it beforehand with the DeleteFile instruction.

See: "Programs\Demo Programs\Files\WriteFile\_ReadFile\_SelectFile.txt"

## Delete a file

The instruction **Save DeleteFile s1** deletes the file specified by s1.

The file name must also include the extension, usually ".txt". Instead of s1 you can use an user-defined variable or even write directly a text string (enclosing the text with double quotes).

If the file is not found then the path becomes the "Files" folder, which is next to the "Automation.exe" file.

## Copy a file

To copy a file you use StringToFile and GetTextFile as in this example:

```
Save StringToFile GetTextFile("File.txt") "FileCopy.txt"
```

As always you can use variables and if you do not specify the path then the "Files" folder is used, which is next to the "Automation.exe" file.



## Select - Case - CaseElse - EndSelect

The Select-Case construct allows you to choose between a number of cases. In these examples could be only two "Case" but it could be dozens.

Obtaining the same operation with a series of "IF" would be more complex and the operation would be less noticeable.

```
Select v1
  Case 1
    Print 1
  Case 2
    Print 2
  CaseElse
    Print "Not 1 and not 2"
EndSelect
```

In this example, if the variable v1 is equal to 1 then the first "Case" is selected. If is equal to "2" then the second "Case" is selected.

The "CaseElse" is selected if all the preceding "Case" are not matched.

Our Select-Case implementation is particularly flexible. Almost all languages require that after the "Cases" there is a constant value, instead in Automation you can also write complex expressions.

In the next image you can see a simple example, only functions that read slot, but as addition and multiplication, and after the "Select" after the "Case", you can write expressions of any complexity.

```
v1 = 3
Select Slot(1)
  Case v1 * 2
    Print "This is executed when Slot(1) = 6"
  Case Slot(2) + 2
    Print "This is executed when Slot(1) = Slot(2) + 2"
EndSelect
```

WARNING: a terminal "EndSelect" is always required.

- - - - -

See also the examples in the folder  
"Demo Programs \ Demo Select-Case"

## Select - Case (special features)

Our Select-Case implementation allows unusual comparisons, impossible in almost all programming languages, but very useful.

The "Case" statements compares strings regardless of upper and lower case letters. Also you can compare everything, strings, numbers, Boolean operators, and expressions with true or false results, permuting them as you like, in the Select and Case statements.

For example, you could write "Select True", and the first "Case" with an expression that is "True", would be executed .

```
Select True
    Case Slot (1) <> 0 And s1 = "Is OK" And v1 = 3
        ...
EndSelect
```

If instead you wrote "Select False", the first "Case" with an expression that is "False", would be executed.

Alternatively, you could write "Select Slot (1) = 3" that would execute "Case True" if the Slot (1) is 3. Otherwise would execute the "Case False" (or the "CaseElse").

```
Select Slot (1) = 3
    Case True
        ...
    Case False
        ...
EndSelect
```

Could also write "Case Slot (2) = 5", and this would behave like a "Case True", only if the Slot (2) was worth 5. Otherwise it would behave like a "Case False".

You can therefore make all kinds of statements and choices, writing them in the way you prefer, and that seems more natural.

- - - - -

See in the folder "Demo Programs \ Demo Select-Case",  
the example "Demo – Select-Comparations"  
and the "Demo - Select-True-False"

# Slot

This instruction reads and writes the numeric Slots.

The slots are the center of the theremino-system communication, and who reads these instructions **it should** already know what they are.

Otherwise, we recommend [read this section](#) and the following about SlotText, and maybe the whole [communications page](#) from start to finish.

Here are some examples of writing in a Slot

```
Slot(1) = 12
```

The value 12 is written in Slot 1

```
Slot(2) = Rnd * 1000
```

In the Slot 2 is written a random value from 0 to 1000

```
Slot(3) = Sin(1 / 3)
```

In the slot 3 is written the number 0.3333333333333333

Examples reading from a Slot

```
v1 = Slot(1)
```

The value of slot 1 is read and assigned to the variable v1

```
If Slot(2) < 500  
    Beep  
EndIf
```

If Slot 2 is less than 500, than a sound is played

More complex expressions are also possible, the following expression writes the value 16 into Slot 3.

```
Slot (1) = 12  
Slot (5) = 3  
Slot(Slot(5)) = Slot(Sin(2)) + Slot(1) / 3
```

- - - - -

See also the examples in the "Demo Programs \ Demo Slots" folder

To view the values of variables and Slots,  
you may want to open the Debug window, with the right mouse button.

# The Command Slot

To communicate with HAL applications a special Slot is used, the "Command Slot". Through the Command Slot (by means of special numbers called NAN) you send instructions to the HAL applications and you can also read information on the number of connected modules and errors, here are some examples:

## Slot 0 = Recognize

HAL applications must recognize connected modules (as if pressing the "Recognize" button on the HAL).

## Slot 0 = Calibrate

HAL applications must calibrate all modules that can be calibrated, usually the CapSensors (as if you pressed the "Calibrate" button on the HAL).

There are also commands that are not sent to the Command Slot, but directly to the Slots to which actuators are connected, such as Servo-motors or Stepper motors. Here are some examples:

## Slot 1 = Sleep

It cuts power to the Servo-motor connected to Slot 1

## Slot 3 = Reset

The Stepper motor connected to Slot 3 is reset with a new value (for details see the HAL instructions).

From the Command Slot you can also read information about the connected devices. After a "Recognize" command or after starting the HAL application, the Command Slot contains the number of connected devices and can be read for example with:

## v1 = Slot (0)

In v1 you get the number of devices (or -1 during recognition)

The Command Slot can contain NAN numbers corresponding to the following strings: "Sleep", "Reset", "Recognize", "Calibrate", "Master disconnected ERROR", "No Masters"

## S1 = DecodeCommandSlot (0)

This function gets the command and error strings.

- - - - -

*Normally the Command Slot is the Zero Slot but, when using multiple HAL applications on the same PC, a different Slot could be used for each of them. In these cases, you open the INI file of the HAL application with the Notepad and modify the "CommandSlot = 0" line with a number other than zero.*

See also the examples in the folder  
"Demo Programs \ Demo SlotZeroCommands"

## SlotText

This instruction reads and writes text strings in the SlotText.

The Slots are the center of the theremino-system communication, and who reads these instructions **it should** already know what they are.

Otherwise, we recommend [read this section](#) and the following about SlotText, and maybe the whole [communications page](#) from start to finish.

Be careful not to confuse Slots with SlotText, have similar addresses (from 0 to 999), but they write and read in different memory areas.

In addition, the Slots contain numbers (integers or floating point), while SlotText contain character strings (up to 100000 characters).

With SlotText you can control various applications, for example: HAL, ArduHAL, IoT HAL, QRdecoder, Cobot, Blockly, CNC, Dictation, GPS, Loggers, Motors, OpenAI, Player, SlotTextToSpeech, Spectrometer and SlotViewer.

In the folder "Programs\Demo Programs\SlotText Commands" there are many examples to try to control various applications with SlotText.

- - - - -

Here are some examples of writing text strings in a SlotText

`SlotText(1) = "House " + "on the tree"` The text is written in SlotText 1

`SlotText(2) = Rnd` A random value is transformed to text and written.

`SlotText(3) = 2.5 + 3` In the SlotText 3 is written the text "5.5"

Examples reading text strings from a SlotText

`s1 = SlotText(1)` A string is read from the SlotText and assigned to the variable v1

`If SlotText(2) = "OK"` If the SlotText 2 contains "OK", then a sound is played  
`Beep`  
`EndIf`

## Speed

The "Speed n" instruction is changed to "Option Speed n". See [this page](#).

## Stop, End

With STOP the program stops, but continues to answer the PC keyboard (instruction **Key n Goto/Gosub xx**), Buttons (instruction **Button n Text xx**), and also to Slots (instruction **Button n Slot nn**).

Instead, the statement END totally terminates the program execution. To start it again you will have to press the RUN button.

## TTS (Text to speech)

These instructions translate the text into speech. Windows 10 usually has at least your local language and the English. To add other languages do "Settings" , "Date, time and language", "Language" and finally with "Add a language".

**TTS SelectVoice "xxx gender"** Select the voice among installed.

Examples: "ENG", "ENGMALE", "ENGM", "ENGF", "ENGN", "ENG Neutral", "FRA", "ITA FEM.", "CHI", "CHI F", "ChiMale", "ZHO M", "ZHO", "CHI neutral"

<b>TTS SetVolume n</b>	Volume adjustment (values from 0 to 100)
<b>TTS SetSpeed n</b>	Speed adjustment (values from -10 to 10)
<b>TTS Speak "xxx"</b>	Speak the "xxx" text
<b>TTS SpeakWait "xxx"</b>	Speak the "xxx" text and Wait until finished
<b>TTS Stop</b>	Stop talking immediately
<b>TTS Pause</b>	Pause the pronunciation temporarily
<b>TTS Resume</b>	Resume speaking the text that was paused
<b>TTSvoices</b>	Returns the list of the installed voices
<b>TTSlanguage</b>	Returns the voice "ISO" name (ENG, CHI..)
<b>TTSready</b>	Returns "True" if TTS is inactive
<b>TTSpaused</b>	Returns "True" if TTS is paused
<b>TTSspeaking</b>	Returns "True" if TTS is speaking

Setting Volume to zero the TTS Speak function is disabled and TTSready returns immediately "True".

To pronounce numbers and strings there are also the applications [SlotsToSpeech](#) and [SlotTextToSpeech](#). And there is also the [Voice](#) application which, in addition to speaking, recognizes words from a list of words and phrases. These applications also work alone, but you could make them interact with Automation through Slots.

See also the examples in the folder " Demo Programs \ Demo TTS "

# Variable

The Automation language contains some predefined variables.

`v1 v2 v3 ... v98 v99`      <- Predefined numerical variables

`s1 s2 s3 ... s98 s99`      <- Predefined string variables

In addition to the predefined variables, other variables can be defined

`Variable Numeric Pluto`      <- User defined numeric variable

`Variable String str1`      <- User defined string variable

You can also define variables by omitting the word "Variable"

`Numeric Pluto`      <- User defined numeric variable

`String str1`      <- User defined string variable

All the variables must start with a letter (upper or lowercase)  
and be at least two characters long.

- - - - -

The Automation language must be simple to understand so there are no local variables. All variables, no matter where and how they are defined, have the same value at every point and every function of the program.

So be very careful to use different variables in each function  
(unless you want to use them to communicate values between functions)

All uninitialized variables contain the value zero or an empty string. To give a value to the variables, write as in these examples:

`v1 = 12`

`s1 = "Pluto"`

`value1 = 123`

`str1 = "This is a string"`

These lines only act if they are executed.  
Otherwise the variables remain uninitialized.

## Variable - Immediate assignments

When defining new variables you can also assign them an immediate value, as in these examples:

```
Numeric Pluto = 12 * 44
```

```
String str1 = "This is a string" + Str(Pluto)
```

Immediate values are assigned to all the variables **when the program is started** (with RUN), **even if the assignment lines are not executed**.

If the program later executes these lines then the values are assigned another time, eventually with different values...

If the assignments contain formulas and other variables (such as the "+ Str (Pluto)" in this example), then the actual value you assign to the variable depends on what was in "Pluto" previously.

Pay attention to these behaviors because in some cases what happens may be unintuitive.

For example, if variables are used during the assignment of values (such as the "Pluto" variable in the example on this page) these may already be initialized when the program is started, or they may not be.

In other words, the value of the variables can depend on the order in which the assignments are written, and later also on the order in which both the immediate assignment lines and those containing normal assignments will be executed.

Everything becomes complicated if the assignments contain formulas and references to other variables which in turn could depend on other variables yet ...

If in doubt, write an initialization function and call it before using variables. Something similar to this example, but to be studied carefully from time to time:

```
Label InitAllVariables
```

```
String str1 = "This is a string"
```

```
Numeric Pluto = 12 * 44
```

```
String str2 = str1 + Str(Pluto)
```

```
Return
```

Experiment with the examples in the folder  
"Demo Programs \ Demo Immediate Vars"



## VarsFromFile

In some cases it may be useful to load variable values from a configuration file. So you could, for example, make the same program work on different machines, with different dimensions and different limits for the movements.

This command could also be useful on other occasions, because it is a generic command, which can also be used several times in the same program and also by reading different files at various points in the program.

### How the VarsFromFile command works

Each time the command `VarsFromFile FileName.txt` is run, all the variables indicated in the file are updated with the values defined in the file.

The file must exist and must not contain errors, otherwise an error message appears and the program stops.

In the file, the variables are written one per line, followed by an equal and the value to be assigned to the variable, as in the following example.

```
Slot_LedR   = 907
Slot_LedG   = 908
Slot_LedB   = 909
Array1      = 11 22 33
Array2      = 11,22,33
```

To avoid errors all the variables indicated in the file must be declared in the program in execution.

The variable names must match, but it does not matter if the characters are lowercase or uppercase, and you can also use spaces or tabs before and after the equal.

In the case of string variables, it is possible to include spaces in the string by enclosing it between two double quotes. as shown in these examples:

```
StrVar1 = "   ABCD   " ' string with spaces
StrVar2 =   ABCD      ' leading and trailing spaces removed
```

You can place the file in the "Files" folder or in the main folder of the application (which contains Automation.exe). You can also place it in sub-folders, but in this case you must prefix the file name with the part of the path necessary to go down to the sub-folder.

Experiment with the "VarsFromFile" example located in the "Demo Programs \ Demo Vars" folder.

The "VarsFromFile" example reads the variables from the "VFF\_Example.txt" file located in the "Files" folder.

## Wait

The program stops on the line of the WAIT instruction, and will continue in the following line, upon the occurrence of certain conditions.

Here are some examples:

<code>Wait Seconds 1:53</code>	Waits for a second and 53 cents
<code>Wait Button "Button 1"</code>	Waits for the pressure of the Button 1 (if initialized)
<code>Wait Slot (4) &gt; 500</code>	Waits for the Slot 4 value becoming greater than 500
<code>Wait AppRunning("xx.exe")</code>	Waits until the application is running
<code>Wait Not AppRunning("xx.exe")</code>	Waits until the application is not running
<code>Wait Application xx.exe</code>	(deprecated) Waits until the application is closed

If you specify only the application file name  
(for example **"Theremino\_SlotViewer.exe"**)  
then the application is searched in the folder **"Apps"**  
and also in all its sub-folders.

See also the examples in the folder "Demo Programs \ Demo Wait"

## Window commands

`Window MinSize`

`Window Sizable`

`Window Maximized`

`Window FullScreen`

With these statements you can change the size of the window, during program execution.

The images on the next page show how the program window appears in various sizes.

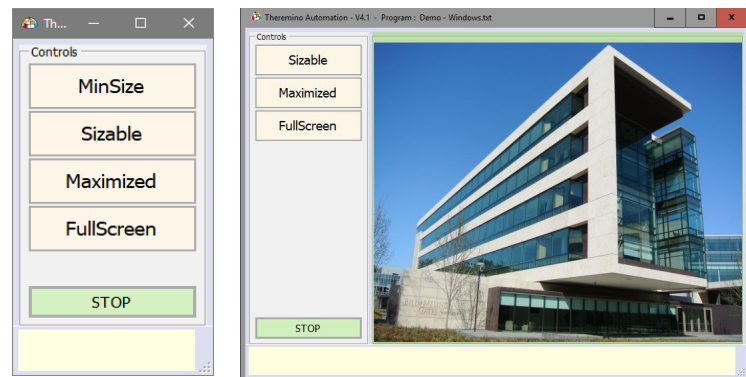
`Window SetFocus`

With "SetFocus" you activate the Automation window and send it the Windows system "focus".

This command can be useful when you want to make sure you are receiving events from keystrokes on the keyboard.

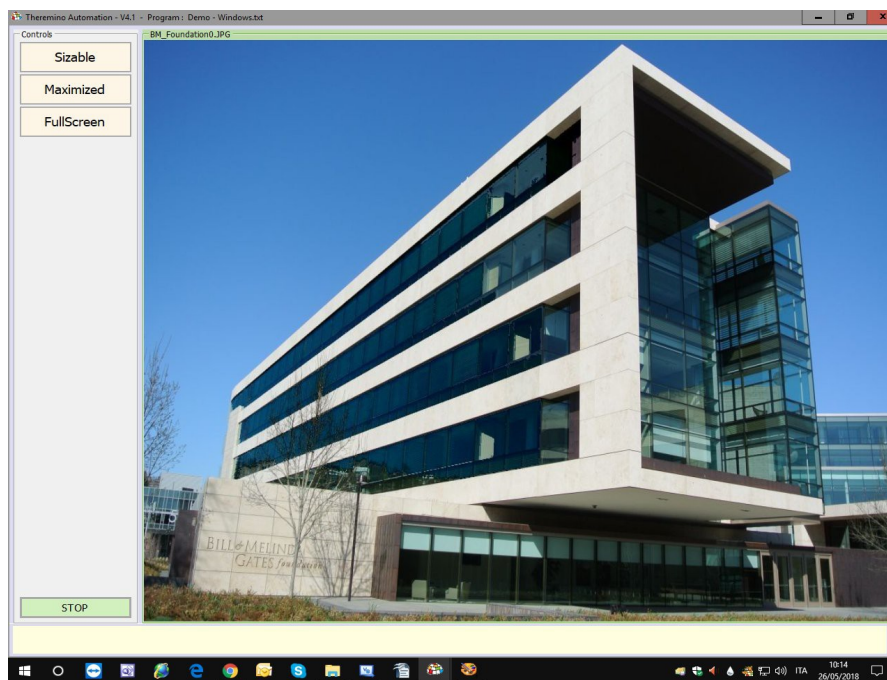
# Window

You can also choose manually the window dimension options, using the [application menu](#) commands Which opens with a right-mouse click on the left side of the application.



	Debug
	MinSize
<input checked="" type="checkbox"/>	Sizable
	Maximized
	Full Screen
	Open TextBox
	Close TextBox
	Clear TextBox
	Default controls size
	Stop running
	Terminate application

See also the example  
"Demo - Windows.txt"



# Calling functions

Eliminating Gosubs and passing parameters to functions simplifies programs and makes code more readable.

The function to which you pass the parameters starts, for example, like this:

```
Label Function1
    Variable Numeric Param1
    Variable Numeric Param2
    Variable String Param3
    ....
```

If you wrote something like this in previous versions of Automation:

```
Param1 = 12
Param2 = Rnd
Param3 = Now
Gosub Function1
Param1 = 567
Param2 = 89
Param3 = "23/07/2022"
Gosub Function1
Param1 = 10
Param2 = 11
Param3 = Now
Gosub Function1
```

Now the same sequence is simplified with:

```
Function1(12) (Rnd) (Now)
Function1(567) (89) ("23/07/2022")
Function1(10) (11) (Now)
```

- - - - -

Function parameters are written in round brackets.

Parameters of type string must be enclosed in double quotes.

The details of passing the parameters  
are explained on the next page.

# Function parameters

The function you pass parameters to must start with one or more rows of variables, that must come all before any other line containing statements.

Parameter variables can be any number and can be all numeric or all string or mixed in any way.

Beware that unlike other languages, variables are NOT local to the function in which they are written, but readable and writable by all other areas of the program. This forces you to write many more variables, all with different names, but makes programming easier for those with little experience.

A good way to generate definitely different variables is to use the initials of the function name as in the following example.

```
Label FunctionWithParams
  Variable Numeric FWP1 '<-- FWP are the name initials
  Variable Numeric FWP2
  Variable String FWP3
  ....
```

When calling the function, you write parameters in round brackets as in following examples (for each parameter the initial and final Spaces and TABs do not count).

```
FunctionWithParams (Rnd) ( -77.234) ( "The quick brown fox")
```

```
FunctionWithParams ( Sin(x) ) ( Cos(x) ) ("The day is: " + Now)
```

Parameters of type string must be enclosed in double quotes, as in the previous two examples, otherwise you will get errors.

Each parameter will then be available in the corresponding variable. If you write more parameters in parentheses than there are variables, then the excess parameters will have no effect. And if there are more variables than the parameters in brackets, then the excess variables will not be set.

If you want to leave a variable unchanged and write the following ones, use the word (Null) as in the following example.

```
FunctionWithParams (Sin(x)) (Null) ("The day is: " + Now)
```

See also the examples in the folder  
"Demo Programs \ SlotText Commands"

# Expressions and Functions

Instead to write a number or a character string, you can write a complex expression. Expressions may contain functions, operators, variables and constants. The IF and Select-Case constructs, can also contain boolean conditions.

## The functions

```
Slot(n) Sin(v) Cos(v) Asin(v) Acos(v) Tan(v) Atan(v) Atan2(v, v)
Sqrt(v) Abs(v) Sign(v) Pow(v, v) Exp(v) Exp2(v) Exp10(v) Log(v)
Log2(v) Log10(v) Int(v) Round(v) Mid(str, index) Mid(str, index,
len) Len(str) Trim(str) PadLeft(str, places) PadRight(str, places)
UCase(str) LCase(str) WCase(str) Format(v, style) Format(v) Str(v,
style) Str(v) Chars(str, number) Replace(str, oldStr, newStr)
RemoveComments(str) Chr(n) Asc(str) Bin(n) FromBin(str) Hex(n)
FromHex(str) Now() Today() Date(year, month, day) ElapsedTime Rnd
Rad(v) ConvertToIEEE754(Value, ConvertSingle, ReverseBytes)
COM_Status COM_Received COM_Portnames MouseX MouseY MouseXP
MouseYP MouseButtons GetSeparatedStringCount(str) CommandText()
GetSeparatedStringCount(str, separator) GetSeparatedString(str)
GetSeparatedString(str, separator) Key(str) DecodeXML(str)
FormatXML(str) FormatXML(str,n) Limit(n, Min, Max) LimitReached
```

## The operators

And Or Xor + - \* /

And, Or and Xor can be boolean and numeric

## The user defined variables (examples)

Variable Numeric Pluto

User defined numeric variable

Variable String str1

User defined string variable

## The predefined variables

v1 v2 v3 ... v97 v98 v99

Predefined numerical variables

s1 s2 s3 ... s97 s98 s99

Predefined string variables

## The constants

False True CRLF NBSP TAB Math\_PI Math\_E Infinity

Infinity = 9e99

## The conditions

Conditions are expressions that end with True or False.

True or False is determined by comparisons (> <=> = <= <>).

## The numerical values

The decimal separator is always the dot. Example: 3.1416

The exponential notation is valid too.

For example: 3e5 that means 3 with five zero, ie. 300000

or: -2.5e2 that means -250

or: 7.3e-5 that means 0.000073



# Expressions results

This language automatically converts Booleans, Numbers and Strings when needed, so in some cases the ZERO could be considered False, and any other number considered True.

## Here are some examples of expressions and conditions

`2 * 3 + 4 * Sqrt(16)` This is an expression, with result 22

`PI * 2` This is an expression, with result 6.2831853 ....

`"Duck" = "Cat"` This is a condition, with result False

For other examples of functions open the file "Demo - Print.txt"  
and the files in the folder "Demo Functions and Errors"

## The Slot () function

This function reads a numeric value from a Theremino System Slot.

For example, the following line reads the Slot 2 value and puts it in the variable v1:

```
v1 = Slot (2)
```

There is also an expression for writing to the Slots, for example the following line writes a numeric value in the Slot 12:

```
Slot (12) = 500.33
```

You can also compose complex expressions and use variables as Slot indices. For example, the following three lines copy the contents of Slots 0 to 9, into Slots 100 to 109, increasing the numerical values by 500.

```
For v1 = 0 To 9  
    Slot (v1 + 100) = Slot (v1) + 500  
Next
```

There are also special instructions for the Command Slot (usually the Zero Slot). See the [Zero Slot - Special Commands](#) page. Here are some examples:

```
S1 = DecodeCommandSlot (0)  
Slot (0) = Sleep
```

# Numerical functions

Please note that all functions in Automation have parentheses, so do not use % for modulus or ^ for power.

**Pi** - Returns the value of Pi Greek (3.14159265358979)

**Sin (number)** - Returns the sine of the number

**Cos (number)** - Returns the cosine of the number

**Asin (number)** - Return the arc sine of the number

**Acos (number)** - Return the arc cosine of the number

**Tan (number)** - Return the tangent of the number

**Atan (number)** - Return the arc tangent of the number

**Atan2 (number, number)** - Arctangent extended to four quadrants

**Sqrt (number)** - Return the square root of the number

**Abs (number)** - Returns the absolute value of the number (always positive)

**Sign (number)** - Returns the sign of the number (-1, +1, or zero)

**Min (number, number)** - Returns the lesser of the two numbers

**Max (number, number)** - Returns the greater of the two numbers

**Mod (number, module)** - Returns the first number limited by the module

**Int (number)** - Returns the integer, rounded down

**Rad (numero)** - Ritorna il numero convertito in radianti (  $n * \text{PI} / 180$  )

**Round (number)** - Returns the number rounded to the nearest

**Rnd** - Returns a random number between zero (inclusive), and one (not included). The number is totally random because the Randomize function is used.

**Pow (number, number)** - Returns the first number, elevated to the second

**Exp (number)** - Returns the natural logarithm base, elevated to "number"

**Exp2 (number)** - Returns "2" elevated to "number"

**Exp10 (number)** - Returns "10" elevated to "number"

**Log (number)** - Returns the natural logarithm (in base "e") of the "number"

**Log2 (number)** - Returns the logarithm in base "2" of the "number"

**Log10 (number)** - Returns the logarithm in base "10" of the "number"

**Limit (number, Min, Max)** - Returns "number" limited between Min and Max

**LimitReached** - Returns True if the Limit function has limited a "number"

See also the examples in the folder "Demo Programs \ Demo Math Functions"



# String functions

**Left (string, n)** - Returns the first "n" characters of the string

**Right (string, n)** - Returns the last "n" characters of the string

**PadLeft (string, length)** - Returns the string padded with spaces up to "length"

**PadRight (string, length)** - Returns the string padded with spaces up to "length"

**Len (string)** - Returns the string length (number of characters)

**Trim (string)** - Returns the string without leading and trailing spaces and TABs

**Ucase (string)** - Returns the string with all uppercase characters

**Lcase (string)** - Returns the string with all lowercase characters

**Wcase (string)** - The first letter capitalized and the other tiny

**Chars (string, number)** - Returns the string repeated number times.

**Replace (string, s1, s2)** - Returns a string with all the occurrences of the string s1 replaced with the string s2.

**RemoveComments(string)** - Return a string without the final part containing the comments. Comments begin with a single quote ('), which is considered only if it is outside a string, i.e. not enclosed in double quotes.

**Mid (string, start, len)** - Returns a string starting from "start", and long "len"

**Mid (string, start)** - Returns a string starting from "start", up to the end.

The "start" parameter of the Mid () functions begins from "1". If start is "0" or a number less than zero, the function always returns an empty string.

**GetSeparatedStringCount (string, optional separator = space)**

**GetSeparatedString (string, index, optional separator = space)**

These two functions count and extract sub-strings from a string, by breaking it at a separator. The separator can be a single character or a string of characters. If the separator is omitted then a space is used as a separator. Any repetitions of the separator count as a single separator.

The GetSeparatedString functions returns the sub-string indicated by the "index" number. The first sub-string has the zero index.

**StringContains (string, "text-to-search")** - Returns "TRUE" if the string "text to search" is contained in the string "string". This function is not case sensitive, uppercase and lowercase characters are equivalent.

# Conversion functions

**Chr (number)** - Returns the char corresponding to the number (from 0 to 65535)

**Asc (string)** - Returns the ASCII value of the first character of the string

**Bin (number)** - Returns a string with the number converted to binary

**Hex (number)** - Returns a string with the number converted to hexadecimal

**FromBin (string)** - Returns a number corresponding to the binary string

**FromHex (string)** - Returns a number corresponding to the hexadecimal string

**Val (string)** - Transform to number a numeric value contained in a string

## **ConvertToIEEE754(Number, ConvertSingle, ReverseBytes)**

Returns a string containing four or eight hexadecimal values, with the number converted according to the IEEE754 format.

If ConvertSingle is written True, four values are returned, otherwise eight.

If ReverseBytes is written True, the bytes are reversed in LittleEndian format.

Example: ConvertToIEEE754(1.234, True, True) = "B6 F3 9D 3F"

- - - -

The following four functions do all the same conversion from number to string using a great number of different options.

**Format (number, style)** - Convert a number to String

**Format (number)** - Convert a number to String

**Str (number, style)** - Convert a number to String

**Str (number)** - Convert a number to String

[Read the details in the next page](#)

# The Format() and Str() functions

The Format function (number, style) converts a number to a string.

In the "style" parameter you must provide a string that specifies how it should look the converted number.

<code>Format(1234.56, "000000,000")</code>	Result: "001234.560"
<code>Format(1234.56, "0.0")</code>	Result: "1234.6"
<code>S1 = Format(1234.56, "0")</code>	Result: "1235"
<code>S1 = Format(Math_PI, "")</code>	Result: "3.14159265358979"
<code>Format(1234.56, "+0.000;-0.000")</code>	Result: "+1234.6" or "-1234.6"

If you prefer a shortest name, you can use the Str (number, style) function which is perfectly identical to the Format (number, style) function.

<code>Str(1234.56, "000000,000")</code>	Result: "001234.560"
<code>Str(1234.56, "0.0")</code>	Result: "1234.6"

Both functions can also be used by omitting the style parameter. In this case the number will be converted with the max precision.

<code>Format(Math_PI)</code>	Result: "3.14159265358979"
<code>Str(Math_PI)</code>	Result: "3.14159265358979"

There are many other style options, such as the next two styles, that convert into scientific notation, with one or two-digit exponent.

<code>"0.000000E+0; -0.000000E+0"</code>
<code>"0.000000E+00; -0.000000E+00"</code>

and the following notation convert "Now" to a date.

<code>Str(Now, "yyyy/MM/dd HH:mm:ss")</code>	Result: 2024/11/30 09:50:14
--	-----------------------------

For a full explanation of all possible formats read [This Page](#).

In this application (and in the whole scientific world),  
the point is always used for decimals,  
even in nations and operating systems that use the comma.

# The MouseX, Y, XP and YP functions

**MouseX** and **MouseY** returns the normalized position of the mouse cursor on the screen from zero (left or bottom) to one (right or top). Normalized values are useful for making adjustments as if you had a Joystick or two potentiometers.

In case of multiple screens this function uses only the screen on which the Automation window is positioned.

**MouseXP** and **MouseYP** returns the position of the mouse cursor on the screen with pixel values. Therefore, on the lower left the two values are zero, while on the upper right the two values depend on the number of pixels on the screen.

In case of multiple screens the pixels start from the bottom left of the first screen to the top right of the last screen and if the main screen is not the first then the numbers that identify the pixels can also be negative.

## The MouseButton function

With **MouseButton** you get a number indicating the mouse button pressed.

1 = left button / 2 = right button / 4 = central button

The combinations of buttons pressed simultaneously produce numbers which are the sums of the individual buttons. So the numbers produced by this function can go from 0 (no button pressed), up to 7 (all three buttons pressed simultaneously).

## The MouseWheel function

This function returns a negative or positive number that tells how many clicks the wheel has been rotated since the last time it was read.

To use it, for example to new an axis of a robotic arm you should:

- Read MouseWheel frequently and add it to the previous value.
- Limit the value between min and max of that axis.
- Give the limited value to the axis.

The Automation application does not need to be selected, but the cursor must be over the "Controls" box that contains the "Buttons". This is to avoid moving the motor by mistake, if you use the wheel on other applications.

See the example Test\_MouseWheel.txt  
in the folder Programs\Demo Programs\Mouse

# The functions that read the pixel colors

These two functions read the color of the screen pixels.

**GetCursorPixelColor** Returns the color of the pixel pointed to by the cursor.

**GetPixelColor(X, Y)** Returns the color of the pixel specified with X and Y.

The color is returned as a 24-bit integer that contains red in the upper eight bits, green in the middle eight bits, and blue in the lower eight bits. If you convert this number with the HEX function you can distinguish the three colors well.

- - - - -

The following functions extract the characteristics from the numeric variable "color" and return a numeric value.

**GetColorRed(color)** Returns the amount of red (a number from 0 to 255)

**GetColorGreen(color)** Returns the amount of green (a number from 0 to 255)

**GetColorBlue(color)** Returns the amount of blue (a number from 0 to 255)

**GetColorHue(color)** Returns the hue (a number from 0 to 360)

**GetColorSaturation(color)** Returns the saturation (a number from 0 to 255)

**GetColorLightness(color)** Returns the luminosity (a number from 0 to 255)

- - - - -

**GetColorIndex(color)** Returns the color index (from 0 to 8) chosen from the nine following colors: Black, Red, Orange, Yellow, Green, Cyan, Blue, Purple, White

- - - - -

The following two functions extract the name of the color from the numeric variable "color" and return a value of type string.

**GetSimpleColorName(color)** Returns the name of the color chosen from the nine following colors: Black, Red, Orange, Yellow, Green, Cyan, Blue, Purple, White

**GetNearestColorName(color)** Returns the name of the nearest known color.

- - - - -

See also the examples in the folder:  
" Programs \ Demo Programs \ Demo GetColors "

## The Key() function

**Key()** returns to the Boolean value **True** if the indicated key is pressed and the Automation application is selected.

If you want the Key statement to act even when the Automation application is not selected or, for example, when it is minimized, you can use the "Option GlobalKeys Enabled" statement.

See the examples "Wait\_Key.txt" and "Option GlobalKeys.txt" that are in the folder "Examples\Demo Keys".

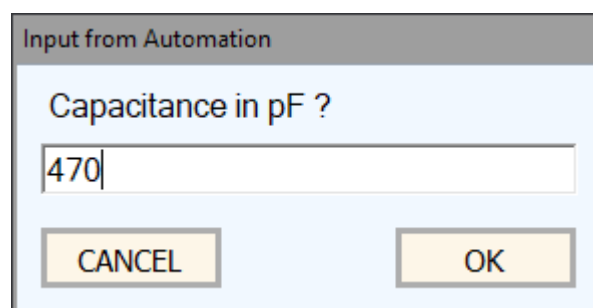
Note that the **Wait** instruction can wait for a condition to occur, so:

- ◆ **Wait Key ("Space")** waits for the space bar to be pressed.
- ◆ **Wait Not Key ("Space")** waits for the space bar to be released.

## The “Input” function

This function pauses the program and waits for the user to input a value.

The value can be a string of characters, or a number, depending on whether you use on the left of the "=", a string or numeric variable.



If the user presses CANCEL the variable is not changed.

Examples:

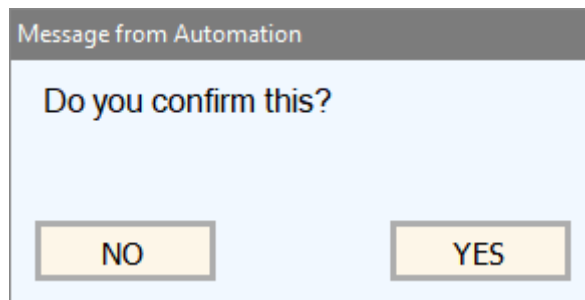
```
S1 = Input "Write your input here"
```

```
V1 = Input "Write a number"
```

See also the examples in the folder  
"Demo Programs \ Demo Input"

# The “Message” function

This function present a message and waits the user to select "YES" or "NO"



Examples:

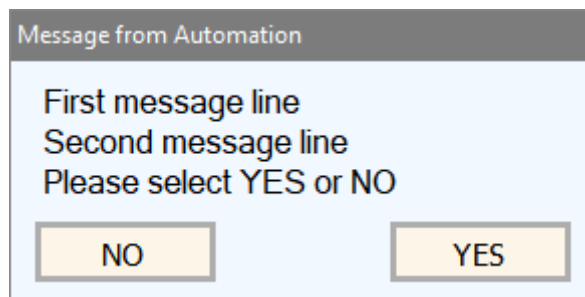
```
S1 = Message Do you confirm this?
```

```
V1 = Message Do you confirm this?
```

If a numeric variable is used to the left of the equal, then its numeric value will be "0" for "NO" and "1" for "YES".

If instead the variable is a string it will directly contain "NO" or "YES".

If desired, you can write the text on several lines, using the CRLF characters, as in the following example:



```
s1 = "First message line" + CRLF
```

```
s1 = s1 + "Second message line" + CRLF
```

```
s1 = s1 + "Please select YES or NO"
```

```
v1 = Message s1
```

See also the examples in the folder  
"Demo Programs \ Demo Input and Message" folder

# The date and time functions

**Date (Year, Month, Day)** - Year, month and day converted to date

**Now** - Returns the date and time of the present moment

**Today** - Returns the date of the present moment

The functions that return a date (Date, Now and Today) may be used in formulas. You can add or subtract one from the other and you can also add or subtract with numbers. In this case the numbers represent days and the decimal part of the number represents fractions of days (which are not minutes or seconds but "fractions of days" ie, for example, "0.5" days are worth 12 hours).

You have to be careful that these functions do not return strings but numbers so in some cases they will not work, for example **Print Left(Now, 2)** does not print the first two digits of the current year but prints "No".

In these cases you will have to transform the date into a string and then perform the operations on it, for example with: **Print Left( Str( Now ), 4)** that prints "2024".

## The “ElapsedTime” function

**ElapsedTime** - Runners of the program.

This feature provides the seconds, since the program was started. The seconds include decimals to the tenths of a microsecond.

To measure a time interval (stopwatch), you can set the starting time in a variable (from v1 to v99 or another user-defined numerical variable) and then subtract it from the final time. As in the following example:

```
v1 = ElapsedTime
...
...
Print ElapsedTime - v1
```

If you hold up the program for a long time, the ElapsedTime value grows a lot, but not have to worry about that. Numeric rounding do not cause loss of precision, and microsecond continue to be valid, even if you keep running the program for many years.

To memorize time values it is best to use v1..v99 or the user-defined numerical variables or the Arrays, which are double-precision (16 significant digits), and not the Slots, that contain single-precision numbers (8 significant digits).



# The filtering functions

These functions are used to filter out noisy numerical data, just as you would with classic electronic resistor and capacitor filters (HP = high pass / LP = low pass).

◆ **v2 = FilterHP (V1, Frequency, NumPoles)**

◆ **v3 = FilterLP (V2, Frequency, NumPoles)**

ATTENTION: these functions must be performed with a frequency at least twice the set cut-off frequency and a maximum of one can be written for each program line.

By setting NumPoles = 1 you get a filter made up of a single resistor and a single capacitor that provides a slope of 6 decibels per octave (or in other words a halving of the voltage for every doubling of the frequency (low pass filters) or for every halving of the frequency (high pass filters).

By increasing NumPoles (up to a maximum of 99) you get filters composed of many cells in cascade, which have a considerably greater slope and therefore a filtering effect (up to almost 600 dB per octave). Filters of such high order would be impossible to make with electronic components, not only due to the exaggerated size they would get, but also because they would not work at all, due to the load produced by the subsequent cells on the previous ones.

Normally a not very high number of poles is used (from 1 to 10) but if necessary you can increase it and obtain more filtering. The only side effect of increasing the poles is an increase in the transit time of the signals.

## Examples

**v1 = FilterLP(v1, 1.5, 1)** This is a high pass filter, with a cutoff frequency of 2.5 Hz and a single pole.

**v3 = FilterHP(v2, 1.5, 1)** This is a low pass filter, with a cutoff frequency of 1 Hz and fifteen poles.

Due to the fifteen poles this latter filter has a transit time of half a second, and with sixty poles the transit time would increase to one second. But the transit time also varies with the inverse of the frequency. So if the frequency were ten hertz these times would drop to twentieths and tenths of a second.

# The “Media” functions

After starting sounds and videos with the "Load" instruction it may be useful to wait for the execution to finish, or to know the current length and position in seconds.

The following functions facilitate these operations.

## MediaLength

The total length of the running file replies (in seconds)

## MediaPosition

Replies the current position of the running file (in seconds)

## MediaPlaying

It returns TRUE if the file is running, or FALSE if the run is finished.

Here is an example that starts a video and then waits the end execution:

```
Load Video1.avi
```

```
Wait Not MediaPlaying
```

- - - - -

The examples are in the folder "Demo LOAD and MEDIA functions".

See also the commands `Load Hide` `Load Stop` `Load Pause` `Load Position`  
e `Load Play`, which are used to control the execution of files,  
and which are explained on [this page](#).

# The XML file functions

## DecodeXML

The `XML = DecodeXML(XML, Section)` function, accepts a string containing XML text and returns a string that contains only the section of text that begins with `<Section>` and ends with `</Section>`

You can use multiple successive DecodeXML lines to isolate a value. The first row will isolate a block, the second row a sub-block and so on, until the single value is obtained.

Some XML files contain lists consisting of multiple sections with the same name. In these cases a FOR loop is used which contains a DecodeXML with an additional parameter. The additional parameter, which is also the increasing variable of the FOR loop, specifies which section to extract. The values of this parameter start at one and grow until the DecodeXML replies an empty string, then an `Exit` is used to end the cycle.

See the example "DecodeXML Plants"  
and the other examples in the "Demo Programs \ Demo XML" folder

## FormatXML

The `XML = FormatXML(XML, Number-initial-spaces)` function, accepts a string containing XML text and returns a string with the leading spaces and the line-changing characters, which make it human-readable.

In the following example, five leading spaces have been used. If in the function `FormatXML` the second parameter is omitted, then two spaces are used.

```
<planes><plane><year> 1977 </year><make> Cessna </make><model> Skyhawk </model><color> Light  
blue and white </color></plane><plane><year> 1975 </year><make> Piper </make><model> Apache  
</model><color> White </color></plane><plane><year> 1960 </year><make> Cessna </make><model>  
Centurian </model><color> Yellow and white </color></plane></planes>
```

```
<planes>  
  <plane>  
    <year> 1977 </year>  
    <make> Cessna </make>  
    <model> Skyhawk </model>  
    <color> Light blue and white </color>  
  </plane>  
  <plane>  
    <year> 1975 </year>  
    <make> Piper </make>  
    <model> Apache </model>  
    <color> White </color>  
  </plane>  
  <plane>  
    <year> 1960 </year>  
    <make> Cessna </make>  
    <model> Centurian </model>  
    <color> Yellow and white </color>  
  </plane>  
</planes>
```

# The File and Folder functions

## **SelectFile (FolderName)**

Opens a dialog and returns a String with the file path and name.

## **GetFileNames (FolderName)**

Returns a String with all the path and file names, separated with CRLF.

## **GetFileNames (FolderName, Sorted, Extension)**

The same of the precedent but eventually Sorted and selected by extension.

## **GetFolderNames (FolderName)**

Returns a String with all the folder names, separated with CRLF.

## **GetFolderNames (FolderName, Sorted)**

The same of the precedent but eventually Sorted.

## **FileExists (FileName)**

Returns TRUE if the file exists.

## **AppRunning (ApplicationNameAndExtension)**

Returns TRUE if the application is running.

## **GetTextFile (FileName)**

Returns a string containing the file text.

## **GetApplicationPath**

Returns a String with the application path.

In alternative you could use ".\xxx" to indicate the Automation.exe path.

## **GetFilePath (FilePathAndName)**

Returns a String with the file path.

## **GetFileNameWithoutPath (FilePathAndName)**

Returns a String with the file name only.

## **REMARKS**

All the returned strings containing a path do not have the ending backslash.

File and Folder Names could be a string variable or a string like: "xxx.exe"

"Sorted" is a boolean variable, as immediate values you write: True or False.

- - - - -

See also the example file: "Demo Files\SelectAndGetFilesAndFolders.txt"

# Auto indentation

```
Label 1
  If Slot(1) < 500
    Gosub 2
  Else
    Gosub 3
  EndIf
Goto 1
```

The indentation improves the visibility of the program, highlights the beginning and end of the structures, and makes it easier to write error-free software.

The Automation application automatically indents programs, while writing them.

Get used to see the indented software is an important teaching aid.

It will become easier then indenting manually, also using programming environments that do not have the automatic indentation.

```
Label testAllSounds
  Wait Seconds 0.01
  If Slot(11) < 100
    If Slot(12) < 100
      If Slot(13) < 100
        If Slot(14) < 100
          Goto loop
        EndIf
      EndIf
    EndIf
  EndIf
Goto testAllSounds
```

## Constructs auto completing

Pressing ENTER on a line of type **For / If / Label or Select**, the structure is automatically completed with the corresponding instructions **Next / EndIf / Return, Case and EndSelect**.

Before pressing ENTER, the **For / If / Label or Select** line must be complete and valid.

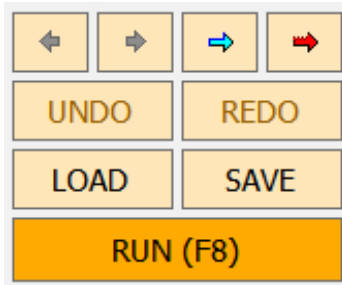
-----

See also the examples in the folder  
"Demo Programs \ Demo INDENT"

# Windows and Menus

## The control buttons

The main functions are always available on the buttons, which can be pressed with the mouse, or your finger on a touch screen.



The most important is the RUN to start and stop the program.

Then there are the LOAD and SAVE buttons, which are used to load and save programs (see also [This Page](#) that explains how to edit and save programs).

The UNDO button is used to go back, if you have made changes to the program and want to eliminate them. The REDO button reconstructs the changes eliminated by UNDO.

The two dark ARROWS move the cursor, and also the visible page, on the program sections previously visited.

The blue ARROW searches for all occurrences of functions, variables or even simple words.

The red ARROW searches in the declarations only (Button, Key, Label e Variable)

*The search functions are very convenient, just select a word or just position the cursor on it and then press the arrow repeatedly.*

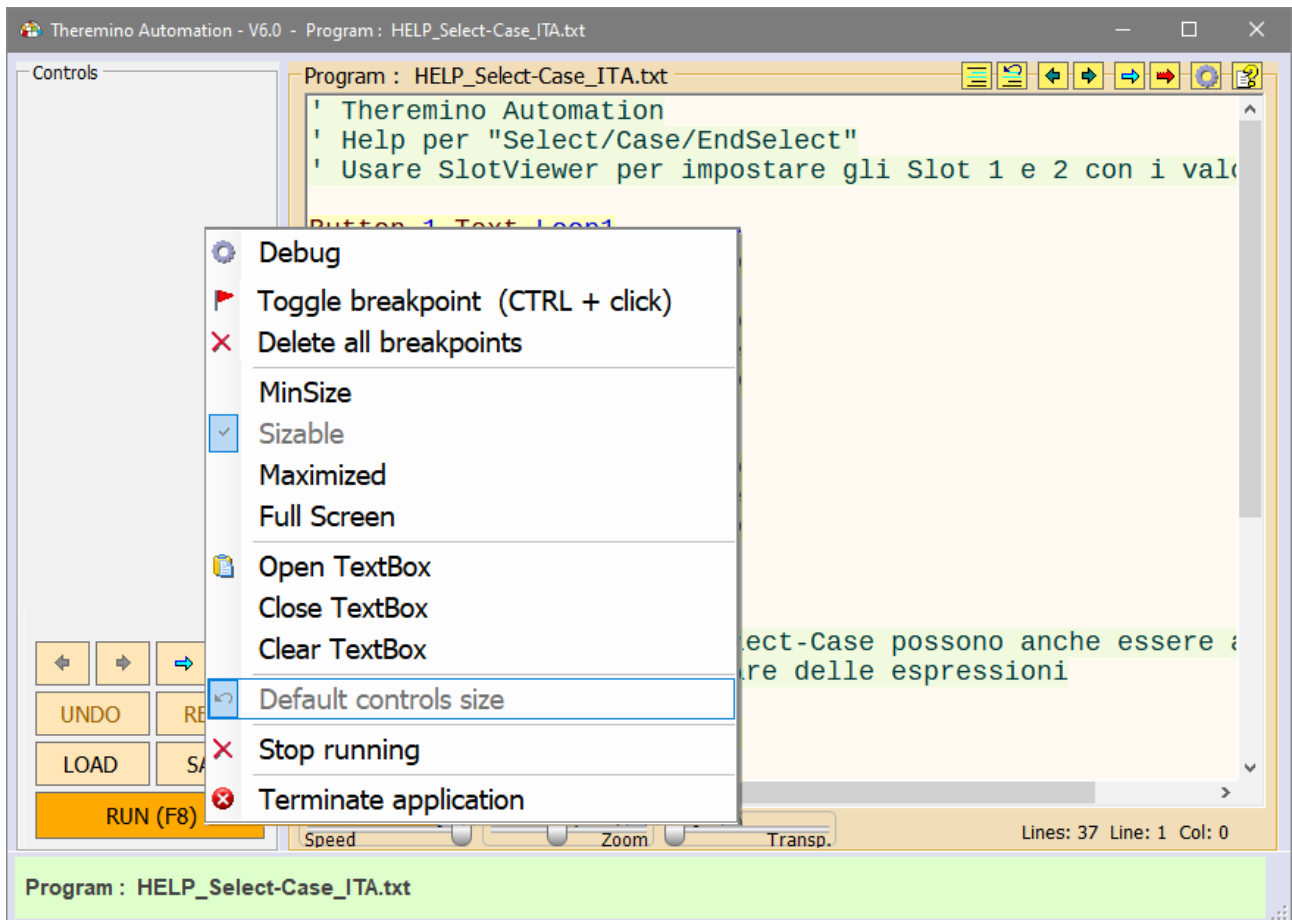
If you are viewing a video or an image in full screen, these buttons are not visible.

On those occasions, to stop the program, you can use the following methods:

- ◆ You can stop running the program at any time, with the key **SHIFT-ESC**, or with **ALT-E**.
- ◆ You can use the right mouse button and open the application menu (shown on the next page).
- ◆ You can start and stop the execution with **F8** or with **ALT-R**
- ◆ If the Debug window is open, the **F8** and **ALT-R** keys do a “Run from cursor”, otherwise the do a normal “RUN” from the program start.

# The application menu

By clicking the right mouse button (or by tapping the touch screen without lifting your finger for two seconds), you open this menu.



When the program is stopped, you must click on the left side of the application (control area), and not on the program area.

Instead, when the program is started, this menu appears when you click anywhere, even on the program, as well as on the pictures and videos to full screen.

This menu allows you to open the Debug window, add and remove Breakpoints, choose the size of the main window, Open, Close and Clear the TextBox, Stop the execution of the program, and also totally close the Automation application.

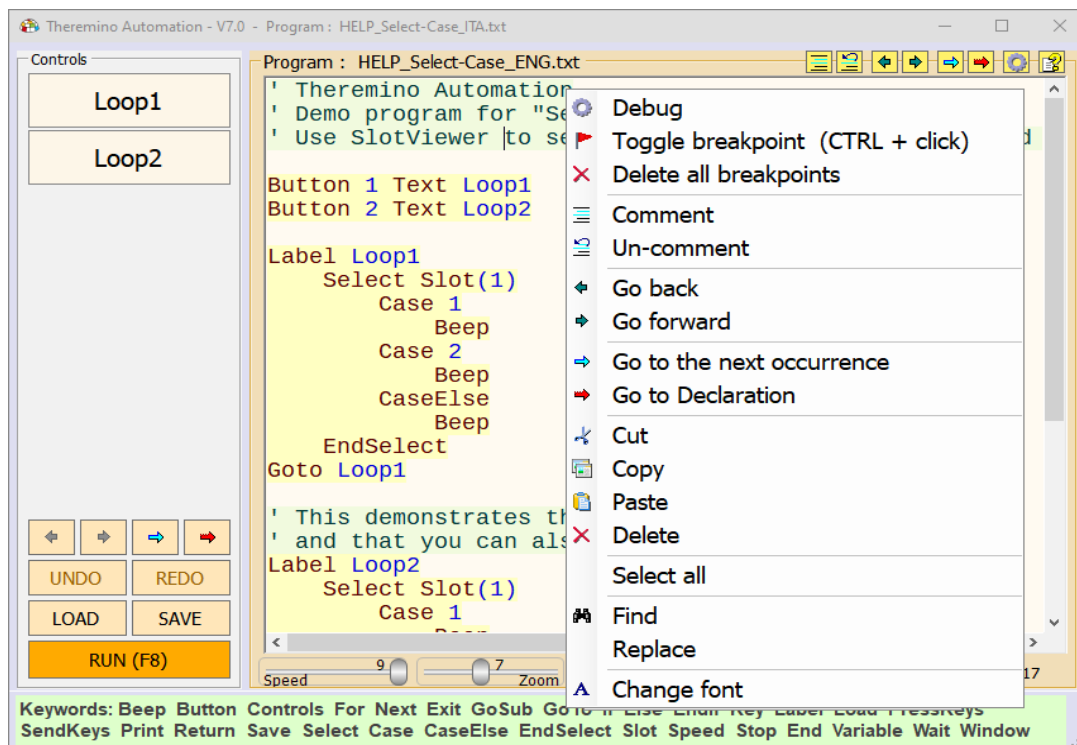
Knowing about this menu is important.

Without him never be able to get out of some situations.

For example, when a full-screen video is active.

# The program menu

By clicking on the program area, with the right mouse button (or by tapping the touch screen without lifting your finger for two seconds), this menu opens.



The program should not be running, otherwise instead of this menu would open the one on the previous page. The first three lines of this menu are used for Debug (software maintenance and monitoring) and will be better explained in [This Page](#).

With "**Comment**" and "**Uncomment**" you comment (add initial quote) to whole areas of the program. Or you delete comments.

"**Go Back**" and "**Go Forward**" move the cursor, and also the visible page, on the program sections previously visited.

"**Go to the next occurrence**" searches other occurrences of the selected word.

"**Go to declaration**" searches the selected word in the declarations rows only.

The commands "**Cut**", "**Copy**", "**Paste**", "**Delete**" and "**Select All**", they copy, paste, delete and sort the parts of the program. In their place, you could also use CTRL-X, CTRL-C, CTRL-V, DELETE and CTRL-A.

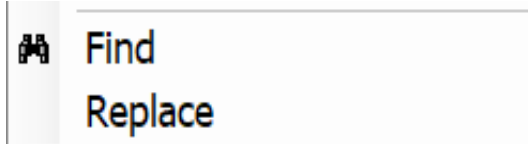
With "**Find**" (or CTRL-F), and "**Replace**", you open the windows to find and replace words and phrases.

The last line "**Font**", is used to choose the font. Clicking on it opens a list of font types, on the left side of the application, and some control buttons.

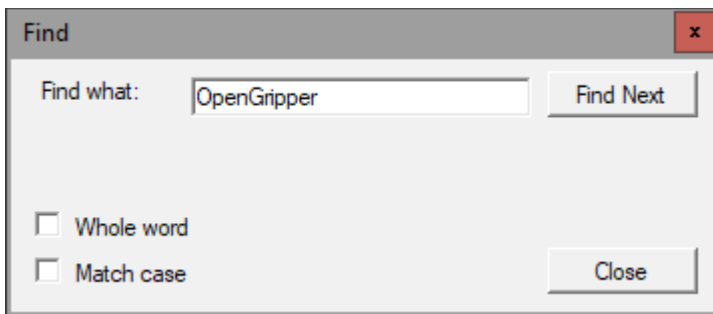


# Find and Replace

With the latest voices at the bottom of the application menu, you open two windows, similar to each other.



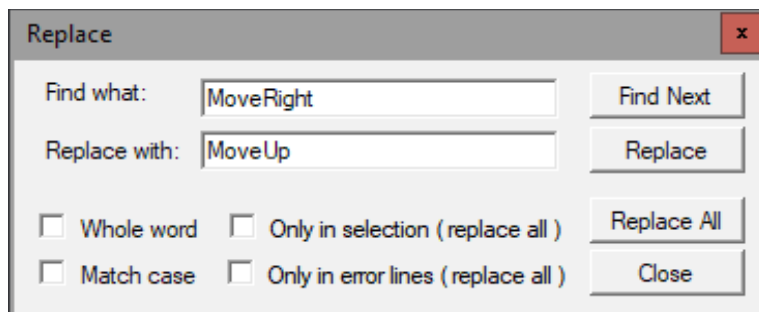
## The "FIND" window



With this window you look for words or phrases in the text of the program.

- ◆ If you enable "Whole Word", the word must be complete.
- ◆ If you enable "Match case", the word should also match as case sensitive.
- ◆ With "Find next" (or F3), you go to the next occurrence of the searched word. If you get to the end of the research program starts from the beginning.

## The "REPLACE" window



This window is the same as the previous, but also lets you replace the word (or phrase) with another.

If you press "Replace" is carried out only one replacement. But with "Replace All" you replace all occurrences.

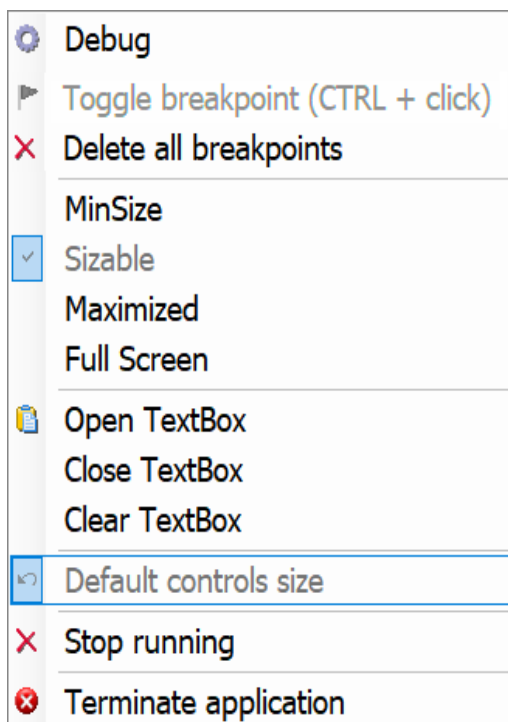
The substitution can happen in all the program or only in the selection, or only in the lines containing errors (or warnings).

# Full Screen operation

"Full screen" means that the display has no visible window edges, and that the desktop bars are hidden.

Automation can view in "Full screen" all the pictures and video.

Even web pages, and the text of the program, are displayed in full screen, but with the buttons vertical bar on the left.



To open the window in full screen, you can use the menu that opens, right-click the mouse, and choose "**Full Screen**".

To return to the "in window" operation, you choose the "**Maximized**", "**Sizable**" or "**MinSize**" options.

Alternatively, to exit from the FullScreen condition, you might interrupt the execution of the program, with the buttons **SHIFT-ESC** or **ALT-E**.

While the program is running you can also use the **Window MinSize**, **Window Fullscreen**, **Window Maximized** and **Window Sizable** instructions which are explained in [this page](#).

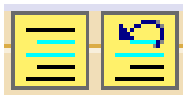
-----

To experiment with the "Windows" instruction, and with the various dimensions of the window, see the "Demo Programs \ Demo – Windows" example.

# The top bar controls



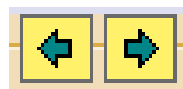
The first buttons adds or removes a bookmark. With the second you can delete all the bookmarks.



These buttons comment and de-comment the selected text.



The blue arrows are used to go back in the changes to the program and to reconstructs the eliminated changes.



The two dark ARROWS move the cursor, and also the visible page, on the program sections previously visited.

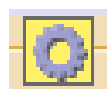


The blue ARROW searches for all occurrences of functions, variables or even simple words.



The red ARROW searches in the declarations only (Button, Key, Label and Variable)

*The search functions are very convenient, just select a word or just position the cursor on it and then press the arrow repeatedly.*



The gear opens the debug window.

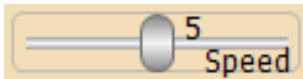


The question mark opens the instruction file (Help) in the chosen language. To make this command work, you need to copy the Help file of your preferred language to the "Automation \ Docs" folder. The latest Help files are downloaded from [this page](#).

If the Help file is not found then a message appears suggesting to open the Docs folder and copy the file into it.

Or you could choose to select a Help file in your preferred language located in the "Docs" folder or any other folder. *To change the selected file click this command with the right mouse button.*

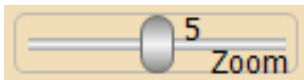
## The bottom bar controls



The SPEED slider sets the program execution speed.

The speeds range from "1" (one instruction per second), until "8" (ten thousand instructions per second), and to "9" (the maximum speed allowed by the system).

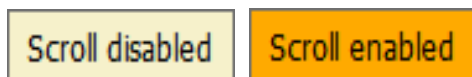
While writing the program it is good to use an average speed. Usually the speed "5" (20 instructions per second), Which is slow enough to be able to visually follow the execution of the program.



The ZOOM slider sets the size of text, both in the program window, or in the Debug window.



This slider adjusts the transparency of the main window and allows you to see also below it.



This button enables the automatic "scroll" of the program during execution. You keep it disabled in order to focus on a function. Instead, it is enabled when you want to follow the general progress of the program.

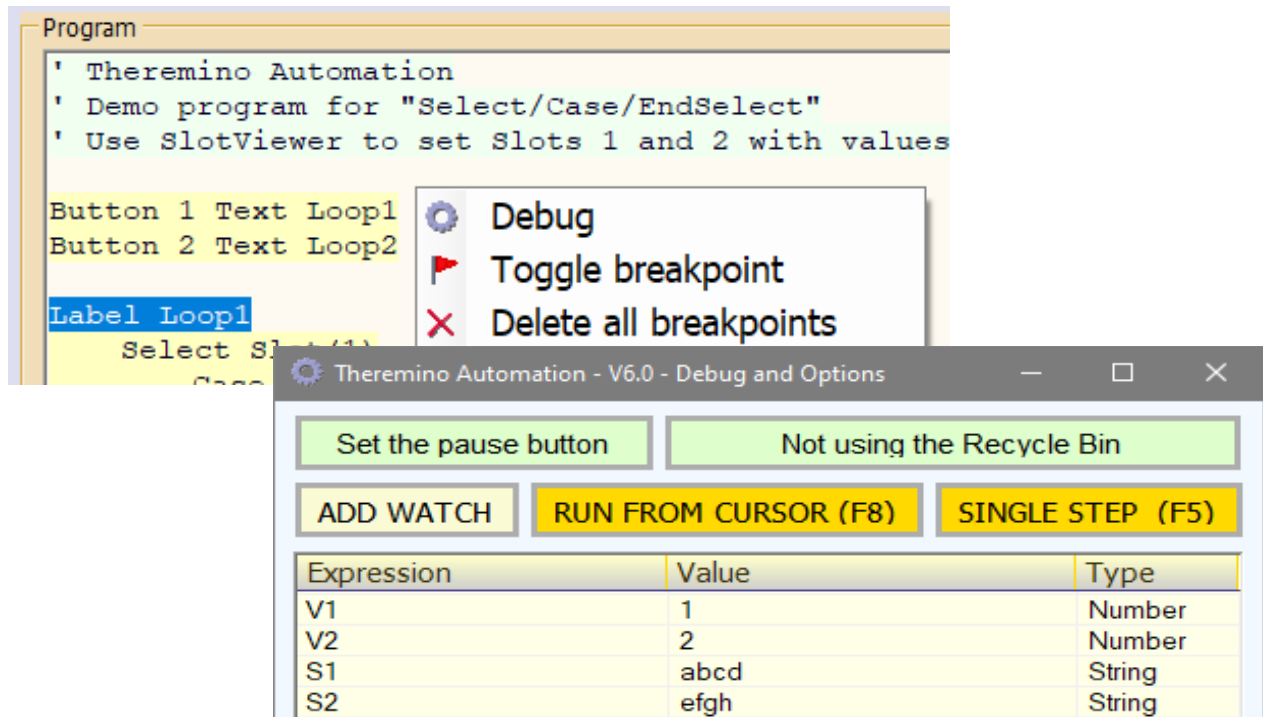
Lines: 29 Line: 17 Col: 16

The right side of the lower bar shows information about the program:

- The total number of lines
- The line where the cursor is (starting from line 1)
- The column where the cursor is (starting from column 1)

# The Debug Window

The "Debug" window facilitates the development of the software. To open it you press the right button of the mouse on the map area, and choose "Debug".



The two main functions are: "RUN FROM CURSOR", which allows you to run the program from anywhere, and "SINGLE STEP", which allows you to run one line at a time.

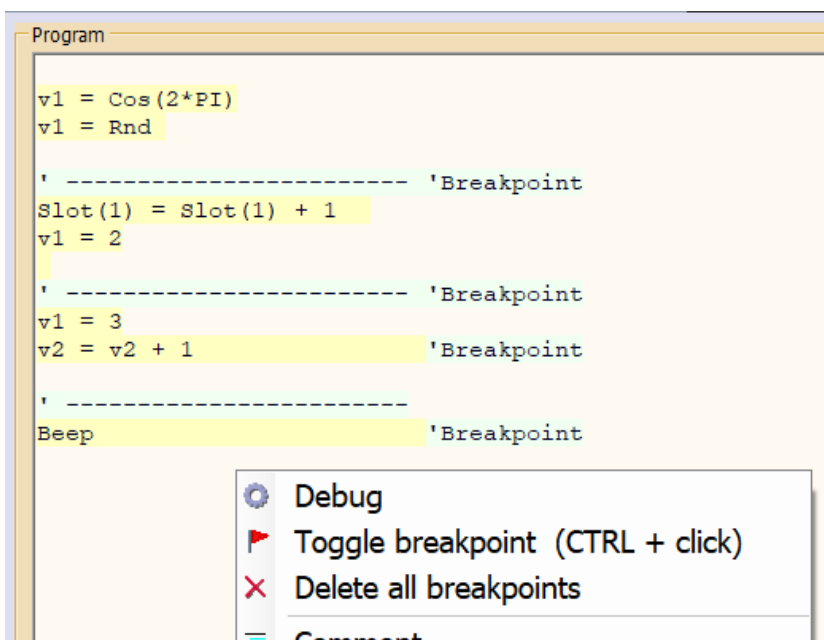
To run the program from an arbitrary point, first of all the stops with the STOP button in the main window. Alternatively, you can stop it by pressing RUN FROM CURSOR (**F8**) or SINGLE STEP (**F5**). Then you place the cursor on the line you want to execute, and you press one of these buttons.

If the Debug window is open, the **F8** key does a "Run from cursor", otherwise it does a normal "RUN" from the program start.

The other functions of this window are: The Watch (watch expressions), the exec line (execute instructions), BreakPoints (interruption points), and two option buttons (green buttons) which will be explained in the following pages.

Experiment with the examples of the "Demo Debug" folder

# The Debug Window (Breakpoints)



The **CONTROL** button + **Left mouse click** adds or removes breakpoints.

You can use this method even while the program is running.

Also when you add a breakpoint the Debug window opens automatically and this is very convenient for intercepting the program while it is running.

When the program is not in execution you can edit the "Breakpoints" with the menu that opens, clicking with the right mouse button on the program lines.

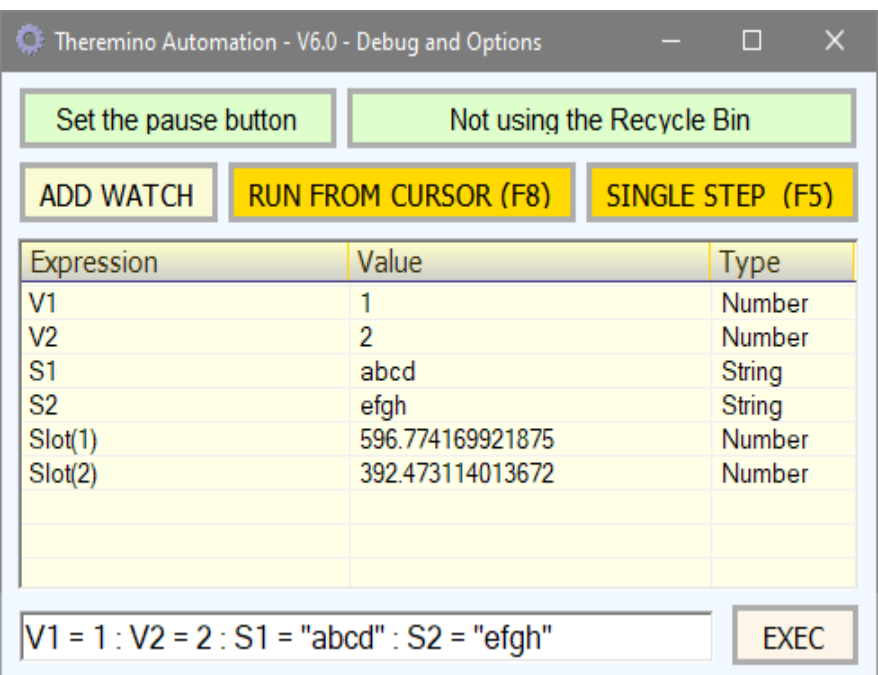
When the Debug window is open, the program stops at every line ending with **'Breakpoint'**.

Instead, If the Debug window is closed, Breakpoints are ignored. So it is possible to leave them in the preferred positions, even in the final program.

When the program is stopped at a breakpoint, you can use all the options in the Debug window.

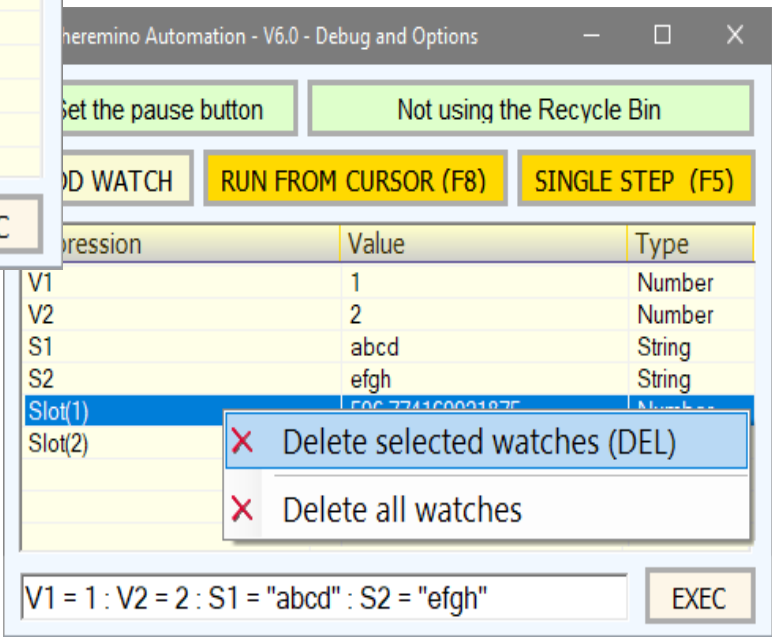
- ◆ You can explore the values of the variables, and the slot (0 to 999), with the watch table.
- ◆ In the Watch you can also write complex expressions and calculations.
- ◆ You can edit the values of variables and Slot, writing assignments in the EXEC line
- ◆ You can execute instructions, writing in the low line and pressing EXEC
- ◆ You can continue to run with RUN FROM CURSOR
- ◆ You can run a single line, with SINGLE STEP
- ◆ You can change the position of running, and then continue with RUN FROM CURSOR or SINGLE STEP

# The Debug Window (Watches)



Each row of this table is a "Watch" (control expression).

With the watch you look at the values of variables and their type, and controls the operation of the program expressions.



The Watch are added by selecting a variable, function, or expression of program, and then pressing the "ADD WATCH" button.

To delete one or more watch, you select the chosen lines, pressing the right mouse button on the table, and you use the menu visible on the right.

To edit a watch you double-click with the left mouse button, on his line, and change the text of the expression with the keyboard.

The Automation application Watches are very powerful, you can write complex expressions, make calculations, check if expressions are true or false, read the values of the slot, add text strings, etc...

Moreover, unlike almost all programming environments, The Watch are updated in real time (ten times per second), even with the program stopped. So they always reflect the actual value of the expressions and Slots. Not even DotNet does this, and it is a very useful feature.

-----

Experiment with the examples of the "Demo Debug" folder



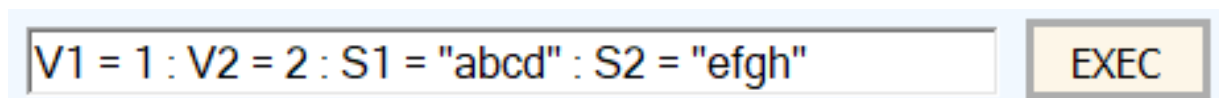
# The Debug Window (Exec)

The bottom line of the Debug window, you can execute instructions and assign values to variables and to Slots.

Using the word Print, you can print complex expressions results, and then have them calculate and know the result.

You can also write the Goto and Gosub, and then press EXEC and jump to the corresponding labels, even while the program is running.

The line also accepts multiple instructions on the same line (separated by a colon, as shown in the example below).



You can use just about any instructions that may be written in the program.

The only keywords not valid are: If, Else, Endif, Select, Case, CaseElse, EndSelect, Return, Stop and Wait. Writing them do not get any effect, they just do not run.

- - - - -

When you write in the EXEC line, the errors and suggestions are shown in the low bar of the main window.



Clicking on the "Select next line" button you jump to the next error line. More info about errors and suggestions in the page "[Keywords](#)".

- - - - -

Experiment with the examples of the folder "Demo Debug"



# The Debug Window (Pause Button)

The new "PAUSE BUTTON SLOT" option, introduced by Automation version 4.5, enables an external mechanical button to pause program execution.

To enable this option you open the debug window, as explained in the previous pages.

Then click on the button called "Set the pause button" and write a Slot number.

This number (from 1 to 999) must correspond to the slot to which an external button is connected.

To enable the external button, write a number from 1 to 999 in the "Input" window, and then press OK. To disable it, write a zero, or clear the box and press OK.

When the external button is connected to a Slot the "Pause Button Slot" button turns dark green and indicates the number chosen for the Slot.

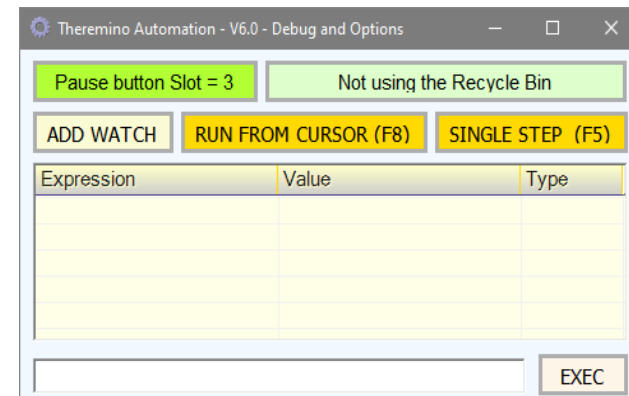
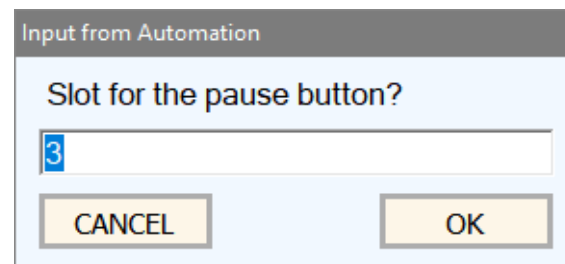
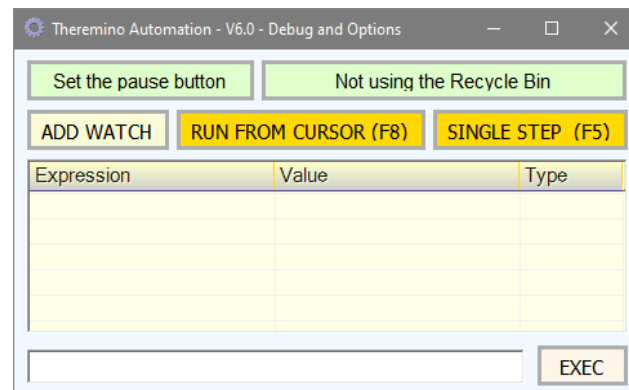
Pressing the external button the program execution stops. Releasing it the execution resumes.

Normally the pause is activated when the Slot value is more than 500, but it is also possible to define a different value, and also to define if the Pause must be activated for values more or less than this value. This is better explained with the following examples:

- 12 The pause is activated when the Slot 12 value is more than 500
- 12>300 The pause is activated when the Slot 12 value is more than 300
- 12<10 The pause is activated when the Slot 12 value is less than 10

During the pause all the application borders colors are orange and you will also be warned by a message in the lower line of the application.

**WARNING : Pause button is pressed (verify the Slot 3)**



# The Debug Window (Pause Button with LEDs)

When large machines are used, sometimes work is done using commands close to the machine (joysticks and buttons) and not always looking at the monitor. In some cases the monitor could also be turned off and the whole interaction could take place by watching how the machine moves and by means of sounds and lights (usually tricolor LEDs or traffic lights) that indicate the various states of the machine (OK, attention, errors, etc. ).



It can therefore happen to have the Pause button pressed and not know it, and consequently imagine defects that do not exist, They therefore asked us to be able to color the LEDs with a color of your choice and also to make them flash when the Pause button is pressed.

First of all, to be able to adjust their brightness, the LEDs must be connected to Slots configured as Pwm16. And it is also recommended to set them as logarithmic to have a more linear adjustment, similar to what the human eye sees.

Then you will have to press the "Set pause button" and set it with more than one number. The first number (12 in this example) indicates the Pause button Slot, while the following numbers indicate the LED Slots.

Slots for the pause button?

12 20 21

If you set the LED Slots, the green button becomes like in this image.

Multiple pause Slots are defined

The example "12 20 21" indicates to turn on the LEDs connected to Slots 20 and 21 to maximum light, when pressing the button, and turn them off when it is released.

The number of LEDs to turn on depends on how many numbers are written, one, two, three or even more can be turned on.

You can also make the LEDs blink with a frequency from 0.2 to 20 Hertz, by writing at the end "F=" followed by a number. The following example causes the LED connected to Slot 20 to blink at a frequency of 3 Hertz.

12 20 F=3

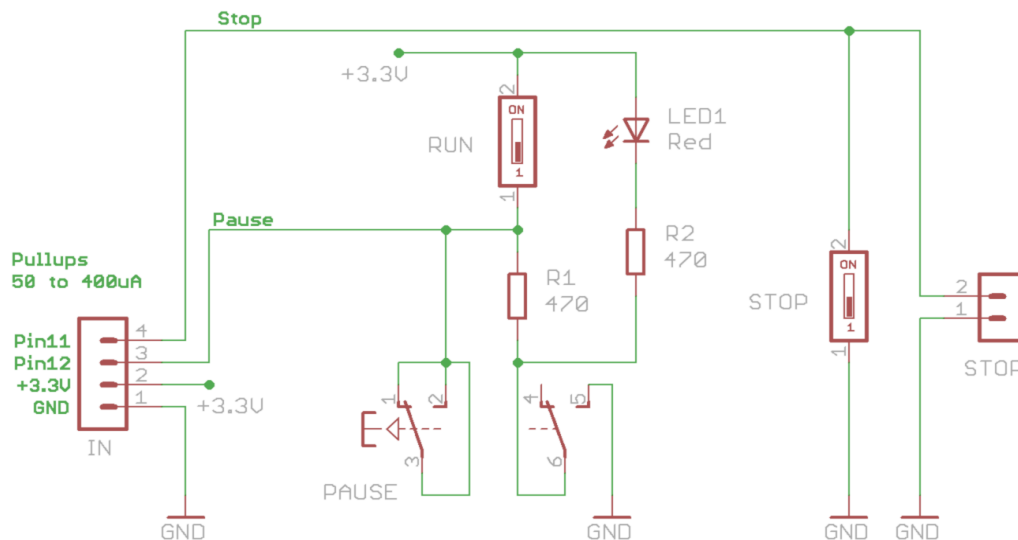
And finally, you can also determine the brightness of each LED. The following example turns on the LED connected to Slot 20 with 800 brightness, 21 with 500 brightness and 22 with 100 brightness and also makes them blink. If a tricolor LED is used, the result is a fast blinking Orange light.

12 20=800 21=500 22=100 F=3

## The Debug Window (Pause Button adapters)

To connect the Pause and Stop buttons we have prepared small adapters in three versions. The first version (positive) is suitable for normally open pause buttons, the second (negative) for normally closed ones and the third is just a series of holes, useful for those wishing to build special adapters.

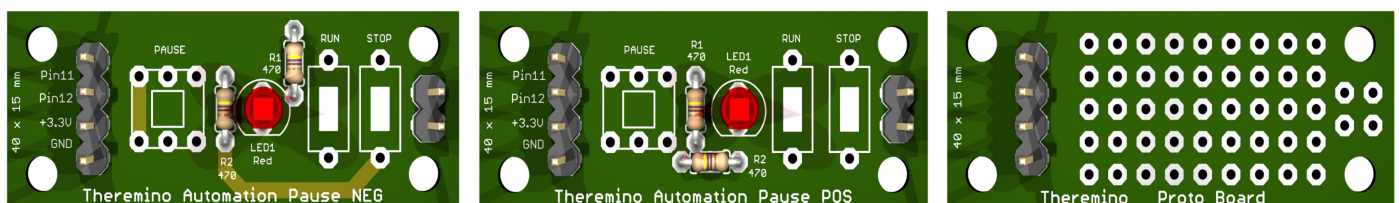
## Theremino Automation Pause POS



The PAUSE button is of the latch type, ie it remains pressed by itself.

The RUN button is used to temporarily unlock the pause.

The STOP button is used to end the program, but note that it is not managed directly by the Automation application. So you will have to write some program lines to read and manage it.



The projects of these adapters with wiring diagrams, 3D images and Eagle files for making printed circuit boards are downloaded with this link: [PauseAdapters](#)

If you want to buy them already assembled, or possibly just PCBs or component KITS, look for them on [www.store-ino.com](http://www.store-ino.com) or on eBay from the seller *maxtheremino*.

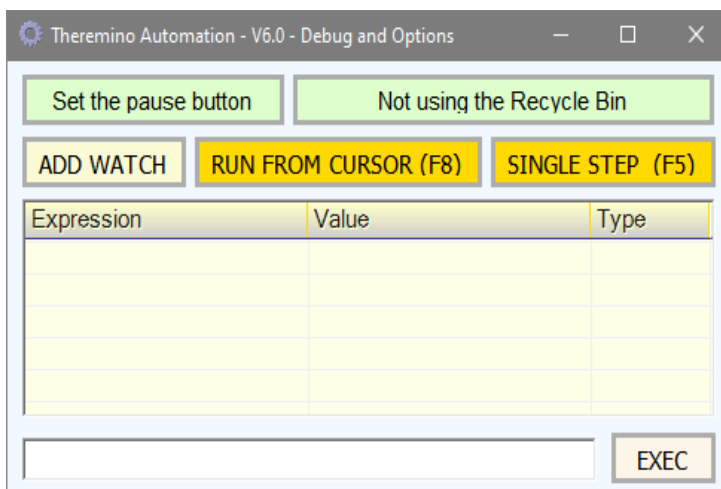
# The Debug Window (Recycle Bin)

Whenever you edit the program text and run it, or load another program or close the Automation application, the current program is saved to disk.

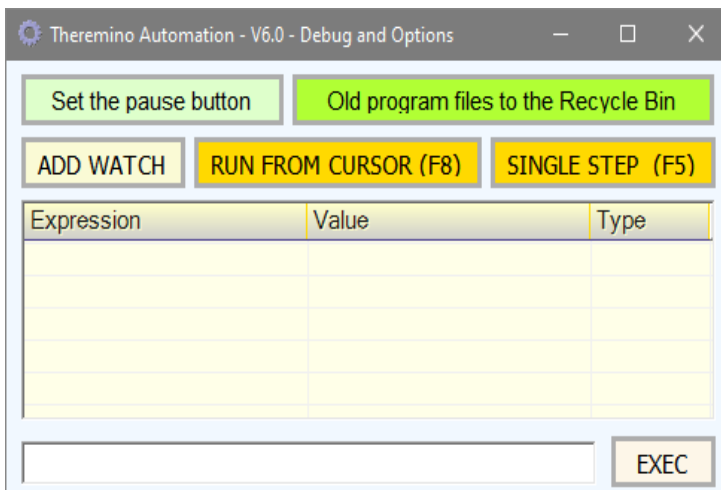
It may happen that you made involuntary changes or made mistakes and only noticed the following day. Or it could happen to cancel a program by mistake.

In these cases we would like to recover the previous program which has however been overwritten by the new version or deleted.

The new "Recycle Bin" option, introduced by version 6 of Automation, saves any previous version in the recycle bin before overwriting it.



To enable the use of the recycle bin, click on the "Not using recycle bin" button.



When this option is enabled the button turns to a darker green and the button text is changed to "Old programs to recycle bin"

It will then be possible to recover all previous versions, by opening the operating system recycle bin, dragging the files (one at a time) into a folder and opening them one by one with NotePad, until you find the desired version.

# Programming techniques

Some Automation language instructions perform operations that would require entire program pages with other languages. To explore all the possibilities of the language we recommend loading, reading and running the examples found in the folders:

## Simple programs

These are programs of a few lines, just to start seeing something moving.

## Demo Programs

This folder contains the most important examples, which explain all the keywords of the language.

## Advanced Programs

Here are the more complex programs that have been created. Programs even with 1000 or 2000 lines, at the limit of Automation's possibilities.

The most complete examples are "**WXM\_CNC\_Vslot\_Automation**" and "**WXM\_CNC\_Vslot\_longrail**" which are in the "Advanced Programs" folder (to run the sequence, and the LOGs, you will also need to use the blue "Select sequence" button and choose the "CNC\_PP\_Cycle" or "LinearTest").

These programs wait for the machine reset signal at start up and every time the sequence is started. If you don't have the hardware you can get past these blocks by **long pressing the CTRL key**. And you could also **comment the lines that test the Master, the RS485 and the TestRAW**.

Many of these programs do not work or work only partially because they would need specific hardware and also a folder and file structure made just for them. But can be useful for studying the techniques and for copying some functions to be used in your own programs.

- - -

The next few pages explain some of the techniques developed when creating complex automation programs. These techniques can be useful in some cases, but to use them you need to have a good level of programming experience.

# Special applications and folders

In the same folder where the main file of this application is located, which is called "Theremino\_Automation.exe", you will also find a folder named "Apps", which contains other applications of the Theremino system.

These applications are used by many Automation examples. Programs often load them on startup and let them close automatically when Automation is closed.

Some of the most often used applications are:

**Theremino\_HAL** to communicate with the Masters via USB.

**Theremino\_SlotViewer** to view the Slots and modify them in the tests.

**Theremino\_SignalScope** to view the Slots as with an oscilloscope.

All these applications must be in the "Apps" folder, in order to share the "SlotNames.txt" file which contains the names and options of the Slots.

Sometimes multiple copies of an application are used, especially the SlotViewer and the SignalScope and in this case a number is added at the end of the name, as in this example: "Theremino\_SlotViewer1.exe".

## **Important**

*Applications located in the APPS folder  
may not be the latest versions.*

*They are fine to try the examples of Automation,  
but to use them in new programs it is advisable to replace them  
with the latest versions that are downloaded from the Theremino site.*

# Open files with Notepad and other apps

These examples open some applications on the Windows system

```
Label OpenNotepad
    Load "C:\Windows\Notepad.exe"
Return
```

```
Label OpenCalculator
    Load "C:\Windows\system32\Calc.exe"
Return
```

```
Label OpenPaint
    Load "C:\Windows\system32\mspaint.exe"
Return
```

Experiment with the "Open Notepad and Apps" example  
located in the "Examples \ Demo Files" folder

Opening applications, for example Notepad, also indicating a file to open is more complex, it is therefore advisable to prepare variables that contain the paths and names of the files and then use them with **Load VariableName**, as in the following example:

```
Label InitNotepadPaths
    Variable String EditorFolder
    Variable String NotepadApp
    Variable String EditSequence1
    EditorFolder = "Apps\Theremino_Editor\"
    NotepadApp = "C:\Windows\Notepad.exe"
    EditSequence1 = NotepadApp + " " + EditorFolder + "Sequences\CircleTest.seq"
Return
```

```
Label OpenNotepad
    Load NotepadApp
Return
```

```
Label OpenSequence1
    Load EditSequence1
Return
```

Experiment with the "Open Notepad With Files" example  
located in the "Demo Programs \ Demo Files" folder

# Open folders

These examples open some folders of the Automation application. Note that the dot + backslash indicates the main folder of the Automation application, that is the folder where the "Automation.exe" file resides.

```
Label OpenMainFolder
  Load ".\"
Return
```

```
Label OpenFilesFolder
  Load ".\Files"
Return
```

```
Label OpenMediaFolder
  Load ".\Media"
Return
```

To indicate folders and sub-folders of the Automation application, the point and the backslash can be omitted, so the three previous examples can simply become:

```
Load ""   Load "Files"   Load "Media"
```

A double period is used to indicate the upper folder. For example to open the folder that contains the folder of the Automation.exe file you can write:

```
Load ".."
```

These examples open the C drive and some Windows System folders.

```
Load "C:\"
```

```
Load "C:\windows"
```

```
Load "C:\windows\system32"
```

- - - - -

Experiment with the "Open Folders" example  
located in the "Demo Programs \ Demo Files" folder

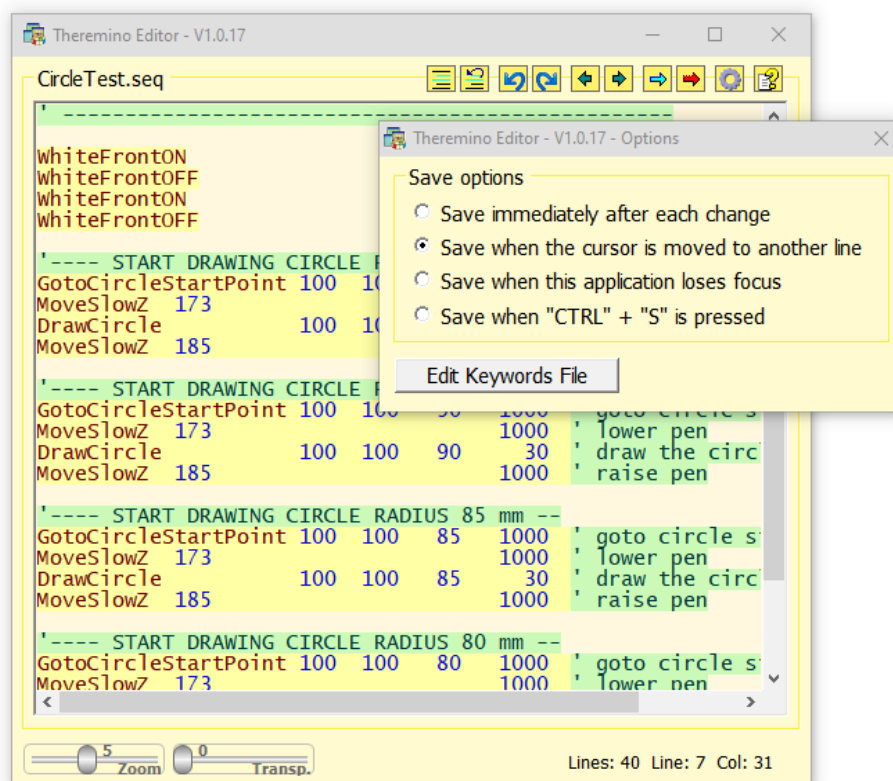


# The "Theremino Editor" application

With this application it is possible to modify command sequences during the execution of an Automation program. See on the next pages how the sequences are used.

The peculiarity of this editor is to modify the sequence file while writing, without the need to use the "Save" command each time. So you can make changes "on the go", without having to stop and restart the execution of the Automation program.

By clicking on the name of the sequence, the folder with the sequence files opens in the first line at the top and can be copied, renamed and deleted.



In the options that open by clicking on the gear you can also choose to save only when you go to a different row, or only when you click on another application, or when you press CTRL + S.

Also in the options that open with the gear is the "Edit Keywords File" button which opens Notepad with a list of keywords.

You can add new words to the list to show errors in a different color and thus make it easier to write sequences.

Some commands can be composed of a single word, others also have parameters.

You will find many example commands, if you don't need them, delete them and write your own.

**ATTENTION:** The commands that are written in the sequences are not executed automatically, but must be interpreted in the "CASE" of the sequence executor that is written in the Automation program.

Experiment with the "Demo OPEN EDITOR" example located in the folder "Demo Programs \ Demo Sequences"

## Start "Theremino Editor" with files

Launch the Theremino\_Editor application passing the name of the sequence as a command line parameter requires a line that contains two paths separated by a space. It is difficult to write everything in one line without making mistakes, so it is advisable to prepare variables with paths, as in this example:

```
Label InitEditorPaths
  Variable String EditorFolder
  Variable String EditorApp
  EditorFolder = "Apps\Theremino_Editor\"
  EditorApp = EditorFolder + "Theremino_Editor.exe"
Return
```

```
Label OpenSequence1
  Load EditorApp + " " + EditorFolder + "Sequences\CircleTest.seq"
Return
```

```
Label OpenSequence2
  Load EditorApp + " " + EditorFolder + "Sequences\TestSeq.seq"
Return
```

Experiment these techniques with the "Demo OPEN EDITOR" example located in the "Examples \ Demo Sequences" folder

## Open sequences with Notepad

You could also open sequence files with Notepad. Using Notepad is easy but compared to Theremino\_Editor you lose the error reporting and also after each change you must remember to save the file.

Here are two examples that open sequences with Notepad:

```
Load "Apps\Theremino_Editor\Sequences\CircleTest.seq"
```

```
Load "Apps\Theremino_Editor\Sequences\TestSeq.seq"
```

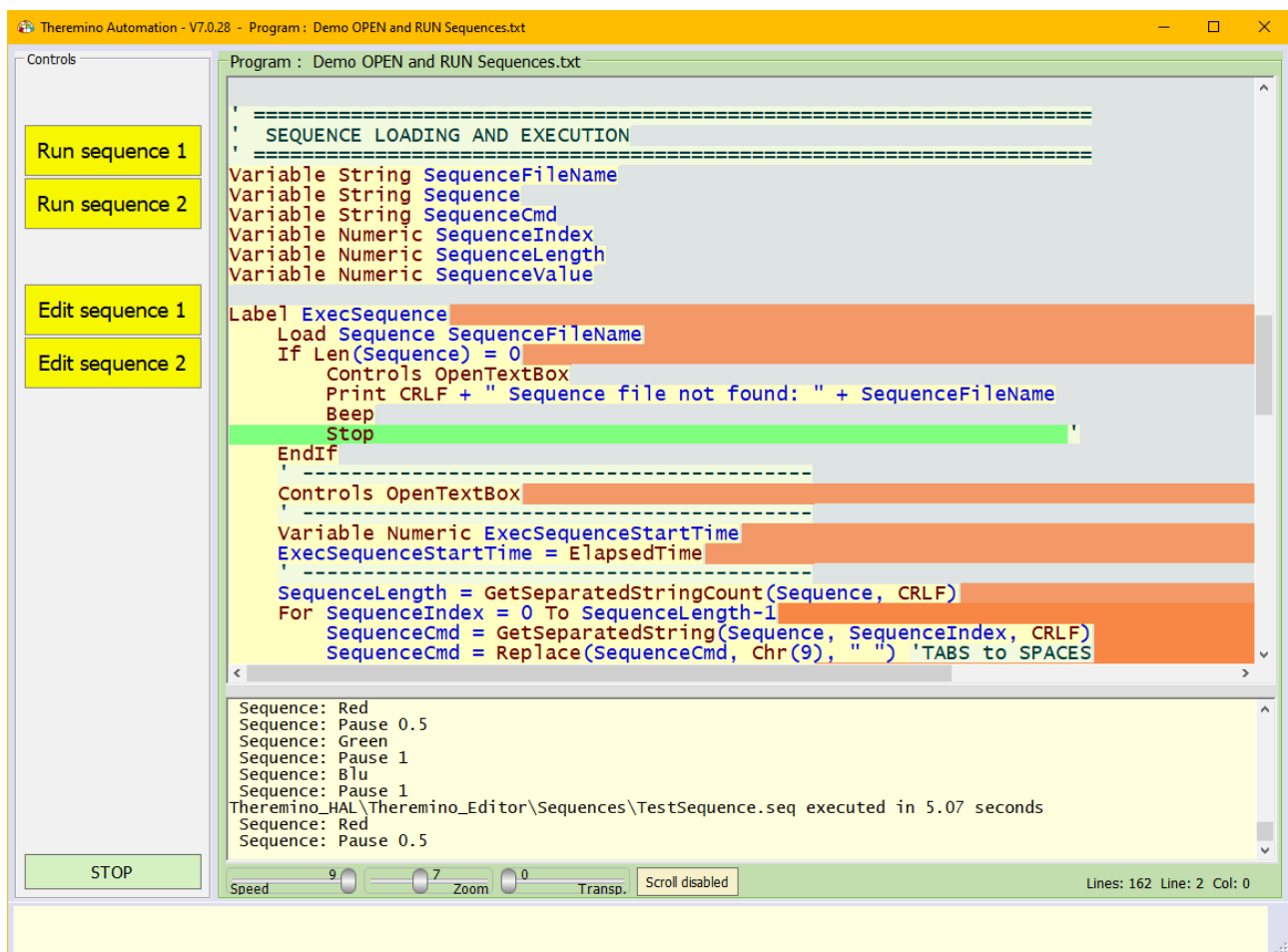
Experiment with the "Demo SEQUENCES" example located in the "Demo Programs \ Demo Sequences" folder.

# Using "Sequences"

Sequences are just a list of commands written in a text file. But it is not a language! You decide the words to execute, write them one by line, and then in the Automation program write what it takes to execute them.

In the Automation program the sequences are executed by a special function that we normally call "ExecSequence". To this function you must add all the "Case" that are needed to execute the commands. Some of the commands may also include parameters.

Using sequences is not easy, you have to decide the names of the commands, write what is necessary to execute them and also add them to the list of valid commands of the sequence editor. The list of valid commands is only used to change color in case of an error, but it is a great help to avoid making mistakes.



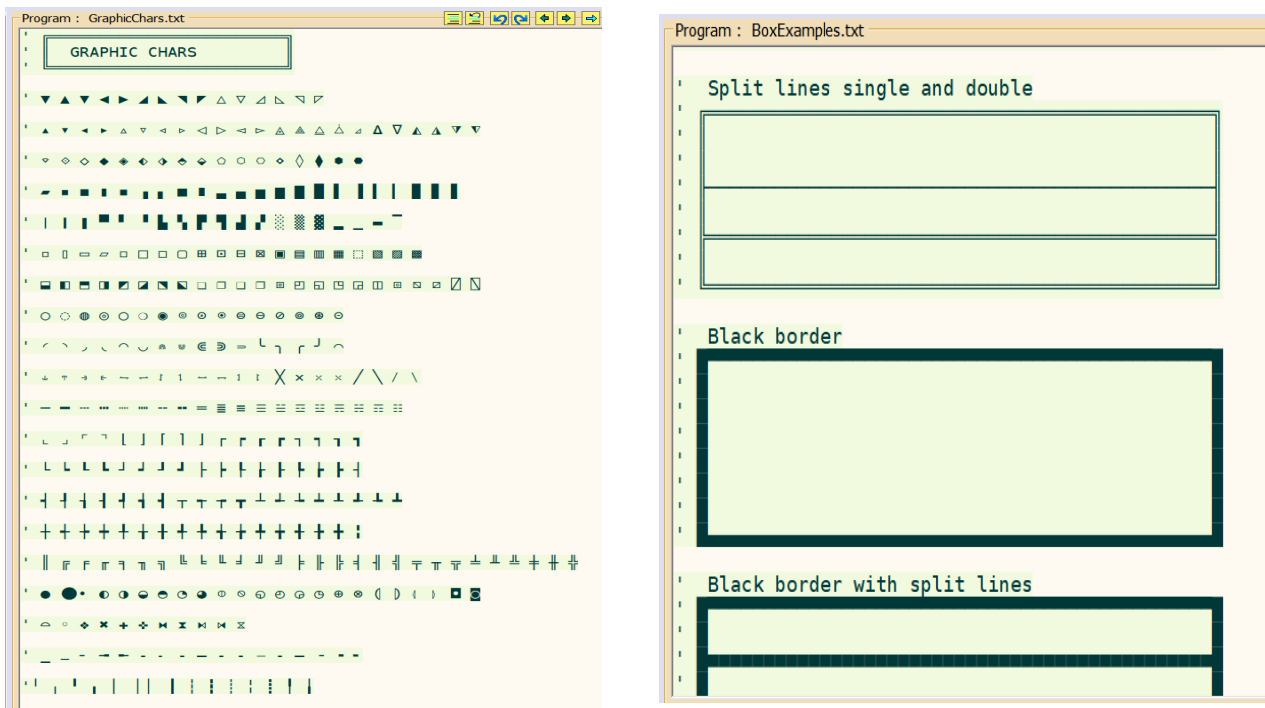
Experiment with the "Demo OPEN and RUN Sequences" example located in the "Demo Programs \ Demo Sequences" folder.

# Using the graphic chars

With the graphic characters you can build rectangles and dividing lines, useful for writing titles and highlighting the various areas that make up the programs.

To write graphic characters in your programs, copy them one by one from the "GraphicChars.txt" file.

To facilitate the composition of the rectangles we have prepared a second file called "BoxExamples.txt", from which to copy the complete rectangles, and then eventually modify them, by adding lines to lengthen them or by adding characters in between to widen them.



Both files are located in the "Demo Programs" \ "Graphic Chars" folder.

Having to copy many characters, you could simplify the copy and paste operations by keeping two copies of Automation open, one with the graphic character file to be copied and the other with your own program on which you will paste the characters. To open a second copy of Automation you must hold down SHIFT while starting it.

To view all the graphic characters, even the strangest, you have to choose the font "DejaVu Sans Mono" or the "Fira Mono".

If you limit yourself to the most common graphic characters, including frames chars, you can also use the fonts: "Courier New", "Cousine" and "Lucida Console".

# Sew Machines functions

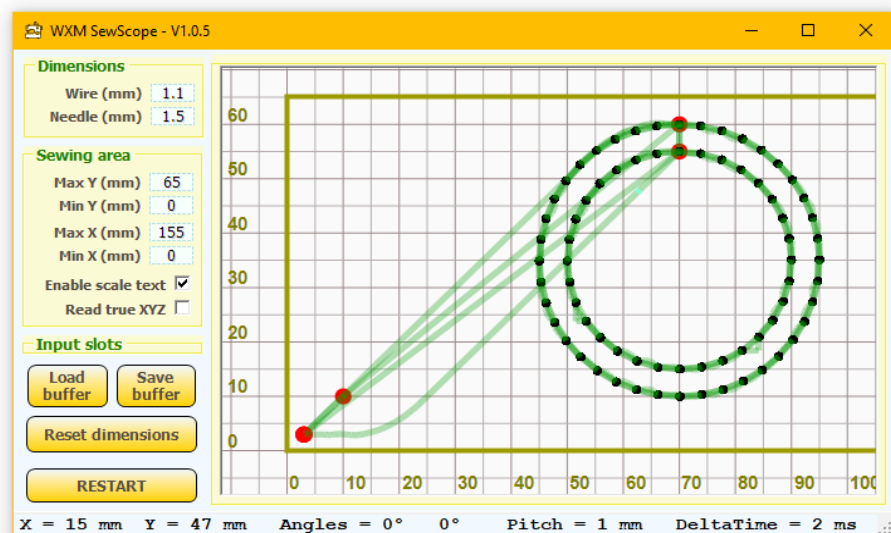
These functions control the position of the fabric and the movement of the needle in the industrial Sewing Machines.



## The SewScope application

This application simulates the movements of a sewing machine and can be useful for fine-tuning sewing programs without using thread and fabric.

The SewScope application is always available in the Apps folder and can be opened by automation programs with the Load command.



## Operating limits for the Sew functions

The Stop\_Angle is the angle of the Z axis when the needle enters the fabric.

The Start\_Angle is the angle of the Z axis when the needle comes out of the fabric.

When the Z axis rotates to Start\_Angle, the X and Y axes are moved to the next destination with the maximum possible acceleration and speed.

To compensate for the acceleration limits of the Z axis and minimize inaccuracies at high speed, the Start\_Angle setting should be increased from a standard value of 270 ° to approximately 350 °

The StopAngle is not used in the "Sew" functions, but is the angle at which the needle enters the fabric. If the Z axis rotates past StopAngle, when X and Y have not yet reached their destination, the needle moves sideways and the seam becomes inaccurate. This limits the maximum linear speed (mm/s) and the maximum sewing speed (stitches/sec).

# Sew functions

These functions (**rarely used and little tested**) drive the three stepper motors of computerized sewing machines.

The X and Y motors move the fabric and the Z motor lowers and raises the needle with each revolution.

The Z motor value 1 represents one 360 degree revolution, so multiples of 1 indicate "needle up" and multiples of 0.5 indicate "needle down".

## ● **SewInit SlotX SlotY SlotZ StopAngle StartAngle**

Initialization parameters (all integer values).

The three Slots are an integer from 1 to 999.

StopAngle is the angle to stop moving XY (about 90 degrees - not used).

StartAngle is the angle to start moving XY (about 350 degrees).

## ● **SewLimits Xmin Xmax Ymin Ymax MaxPerSec**

Movement limits (all float values).

Xmin, Xmax, Ymin and Ymax are in millimeters.

MaxPerSec is the maximum number of points per second.

## ● **SewTo DestX DestY Speed Pitch Options**

Instruction that starts a sewing segment execution.

DestX and DestY are the end-of-segment destinations in millimeters.

Speed is the speed of movement in mm per second.

Pitch is the width of the stitches in millimeters.

Options is a string of options starting and ending with double quotes.

## ● **SewAbort**

The SewTo command is asynchronous and runs on a separate thread, until the segment is completed or until it is stopped with SewAbort..

## ● **Var1 = SewProgress**

The variable Var1 receives a number from 0 to 1.

This number indicates how much of the last SewTo has been executed.

# Sew functions - Examples and Notes

## Examples of options for the SewTo function

	DestX	DestY	Speed	Pitch	Options	
SewTo	70	35	20	4	"Radius=5"	Circle with 5 mm radius
SewTo	70	35	20	2	"ZigZag=2"	ZigZag of 2 mm (+/- 1mm)
SewTo	10	30	20	5	"Begin=2"	Two starting points
SewTo	10	30	20	5	"End=3"	Three final points

The options "Begin" and "End" are not implemented.

You can also combine all the options, as in the following example

	DestX	DestY	Speed	Pitch	Options
SewTo	70	35	20	4	"Radius=5 ZigZag=2 Begin=2 End=3"

## Speed limits tested on a sewing machine with three medium-sized stepper motors

- Maximum linear speed: 20 mm/s (visible inaccuracies at 30 mm/s)
- Maximum sewing speed: 10 stitches/s (visible inaccuracies at 15 stitches/s)

## Adjustments of the motors on the HAL application

- Motors X and Y --- MaxSpeed=15000 MaxAcc=5000 StepsPerMM=16
- Motor Z --- MaxSpeed= 3000 MaxAcc=260 StepsPerMM=400

## Simplified table for speed parameter adjustments

Pitch	Speed max	Speed limit	Stitches per sec resulting
1 mm	10 mm/s	15 mm/s	10 (15 max)
2 mm	20 mm/s	30 mm/s	10 (15 max)
>2 mm	20 mm/s	30 mm/s	less than 10



# Known issues

## The single quote does not work well

The single quote is the one used to start comments. It may happen that pressing the single quote on the keyboard does not appear. To make it appear, press the button a second time.

This defect happened only in China on some computers that had been installed in a particular way. We could not reproduce it and therefore we cannot debug.

To solve it, select the "English US" keyboard instead of the "English international" keyboard. The selection of the keyboard is in the Windows lower bar, to the right, near the clock.

## Text too large on the control buttons

On some computers in China it happened that the buttons on the left of the application had too large writing. We could not reproduce this defect.

## Variables incorrectly referred to as duplicated and other errors

Sometimes some variables turn red and when you move the cursor over them you can read that they are duplicated or you get other errors. Normally using the UP - DOWN arrows, or press the "Select next error" button, the errors disappear.

In case of persistent and incomprehensible errors it is recommended to press the right mouse button on the LOAD button. In this way the program is reloaded and all the error tables are emptied and rebuilt. Sometimes with this method some errors fix themselves automatically.

## Other known defects

Currently (Version 7.8 of 1/December/2024), we know of no other defects. If you find one, please email us at: [engineering@theremino.com](mailto:engineering@theremino.com)