Easy Touch Scripting Manual

Version 2013.2.0
August 27, 2013

CIRRIS

An employee owned company

Easy Touch Scripting Manual

1991 W. Parkway Blvd.
Salt Lake City, Utah 84119-2026
U.S.A.

Table of Contents

You may download example scripts files and a PDF version of this manual from the Cirris Community Forum.

To do this:

1. In your web browser, type in the URL https://community.cirris.com.
2. Click **Downloads** on the top menu bar.
3. Click **Documentation** under Download Categories.
4. Find the downloads Search box in the upper right corner of the download page. In this box type **Easy Touch Scripting**.
5. Click on either the Easy Touch Scripting manual or examples.
6. Click **Download**.

Introduction

What is Easy Touch scripting?

You can use Easy Touch scripting on a Cirris Easy Touch tester or on a Cirris 1100 tester that is controlled by a PC running the Cirris easy-wire software. An Easy Touch script will typically not run on an independent 1100 Tester, or on a Cirris Touch1 tester. A Cirris scripting language with some different commands is used for these applications.

By using scripting you can make the tester extremely flexible and customize a test process to fit a unique task. The following list gives examples for using scripts:

- Create complex tests that are easy for line workers to use.
- Make highly customized reports or cable tags.
- Control external devices such as lamps, hold-down clamps, markers, etc.
- Test devices such as switches, gas fuses, zener diodes, bi-color diodes, etc.
- Verify color and lighting of LED's.
- Display prompt messages.
- Hipot different nets at different voltages.

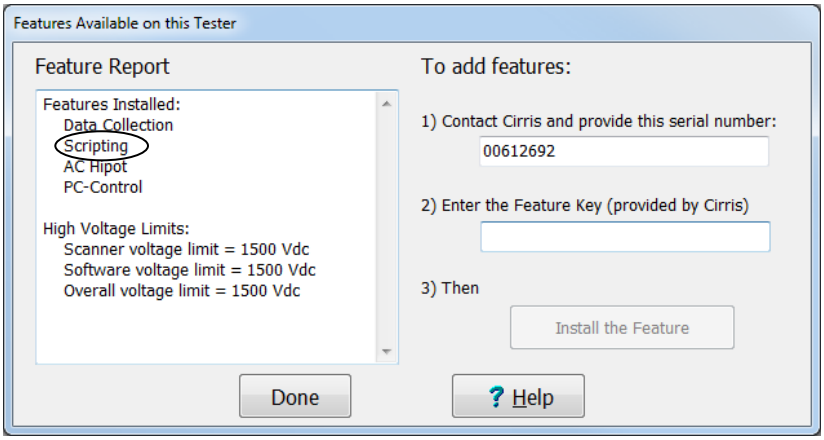
Easy Touch scripting is written in the Lua programming environment. Lua is a free programming language often used to control factory equipment. In addition to the standard Lua programming commands, Cirris has added its own programming commands to Easy Touch scripting to provide the tools needed to control the Cirris cable tester. This manual documents these tools. In addition to this Easy Touch Scripting Manual, use the Lua 5.1 Reference Manual, which documents all of the general Lua programming commands. You can find the Lua 5.1 Reference Manual at www.lua.org.

Enabling Scripting

Scripting must be enabled before you run a script file on your tester. If scripting was ordered with your tester, the scripting option should be checked on the tester's options label. You can find this label on the top front corner on the left side of the tester. If you are enabling this option in the field, use a marker to check this label when Scripting has been enabled.

You can verify or enable scripting for the Easy Touch Tester, or for the easy-wire software you are using to control your 1100 Cirris tester by doing the following:

1. From the easy-wire main menu, select **Utilities**.
2. In the System Utilities box select **Setup System Options**.
3. Under the Tester Hardware tab, select **Enable Features**.
4. If enabled, scripting should be displayed in the installed features list as shown below. If not enabled, enter the Feature Key Code received from Cirris, and press Install the Feature.



If you are using an 1100 series tester, PC Control must also be enabled in the above Enabled Features window.

Getting a Script

Writing scripts can be challenging. You may choose to have Cirris write a script for you, or modify an existing script for your application. For those who choose to write their own scripts, having an aptitude for programming and previous programming experience is highly recommended. You may download example scripts files from the Cirris community forum. See the page after the table of contents for instructions on downloading these examples.

Scripts are ASCII text files. You can write or modify a script using a text editor or a word processor running in text mode. Many programmers prefer to use a text editor that is optimized for the programming environment Notepad++ is a popular editing program that can be downloaded for free. If using Notepad++, make sure to set the language option to Lua for better programming clarity.

Whenever you are about to make major modifications to a script that has been functioning, make sure to make a copy of the script to preserve the original file. When developing a script, being able to quickly make and try script changes is critical. To this end you may find it helpful to attach a keyboard to the Easy Touch tester to edit a script. You may also store script files on a network where they can be accessed by the tester and your programming station. When using easy-wire software to control an 1100 tester, you may copy the script files to any drive and directory that is accessible to the PC running the easy-wire software.

Different kinds of scripts

There are different kinds of scripts. The kind of script you use depends on the application, and when the script executes. Each kind of script has a unique syntax and file extension requirement. The kinds of scripts are summarized below and then treated in detail in the following pages.

Test Event Scripts

Test Event Scripts execute when certain events happen during a test. A test event script does not cause a test to pass or fail. One Test Event Script can be assigned to a test program. There are two kinds of Test Event Scripts: “LUA” and “EVT.” Each of these Test Event Scripts have different syntax requirements and file extensions. EVT scripts have been developed more recently, and have the added flexibility of using parameters that can be easily modified in the tester’s user interface.

Component Scripts

You can attach one component script to test program. However, the component script file can have one or more custom components. After attaching the component script file to a test, you can add its custom to the test in manner similar to how you can add test instructions. Like test instructions, custom components must be correctly completed for a test to pass. Therefore, a component script differs from other script types in that it can cause the test to fail. Creating a custom component script can allow you to test standard components to higher level or test electrical components and conditions that differ from standard components.

Custom Report Scripts

The easy-wire software that controls the Easy Touch tester uses makes it easy to customize and print a broad range of report and label options. However, sometimes even this broad range of options may not be suitable. Custom report scripts give you full control over the content, format and presentation of the printed documentation. Note that the easy-wire reporting allows you to turn reporting on or off, or configure it differently for different test programs. On the other hand, a custom report script can run each time a test fails, each time a test passes, or after every test. If you need a custom report script for select test programs, you can use a test event script for a custom report application.

LUA Test Event Scripts

Overview

A LUA Test Event Script is executed when one or more test events occur during the test process. The easy-wire software uses a value to represent each of these events. When the event occurs, the script passes the value to the script. The test events and their values are shown in the table below.

value	LUA test event
1	The first test event occurs when the test program is loaded.
2	The second test event occurs when the low voltage test begins. The low voltage test is the first test that is performed on a cable and always occurs. If the Test Method is set to Signature Continuous Test, the low voltage test starts when the operator attaches a cable assembly. - OR - If the Test Method is set to Signature Single Test, the low voltage test starts when the operator presses Start in the Test Window.
3	The third test event occurs at the end of all tests on a cable. This includes the low voltage test, and if selected, the high voltage and intermittents test. If the Test Method is set to Signature Continuous Test, the end of all tests happens when the cable assembly is removed. - OR - If Test Method is set for Signature Single Test mode, the end of the test happens when the low voltage, and if selected, high voltage test are completed.

A LUA Test Event Script says in essence, “When a test event happens, do the following thing or things.” A LUA Test Event Script will not cause the test to pass or fail. If you want to test custom electrical characteristics of a circuit, you should use a component (CMP) script instead. Unlike CMP and EVT scripts, LUA Test Event Scripts accept no parameters. Parameters allow you store values for the script, which can then be changed in the test setup without modifying the script.

You can assign one test event script to a test program. A LUA Test Event Script must end with a .lua file extension and must include the required syntax as shown in the following section.

Required Syntax for a LUA Test Event Script

Every LUA Test Event Script must start by defining the function DoOnTestEvent. All other functions in the script are called from the DoOnTestEvent function.

```
function DoOnTestEvent (event)
        if event == eventnumber then
                .
                .
                .
         All other function calls
                .
                .
                .
       end
end
```

Note, a function definition must always end with an end statement. The easy-wire software passes the value of ***event*** to the script when the LUA test events occur. See the table of LUA test events and their values on the previous page.

Examples of a LUA Test Event Syntax

Here's a short LUA test event script that brings up a window to congratulate the operator upon loading the test program.

```
function DoOnTestEvent(iwhen)
   if iwhen == 1 then
      MessageBox("Good job loading the test program")
   end
end
```

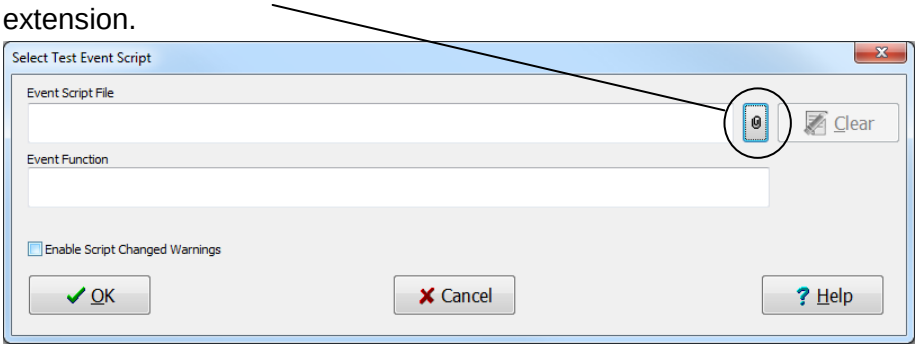
Here's a slightly longer LUA test event script that congratulates the operator at all three test events.

```
function DoOnTestEvent(iwhen)
   if iwhen == 1 then
      MessageBox("Good job loading the test program")
   end
   if iwhen == 2 then
      MessageBox("Good job starting the test")
   end
   if iwhen == 3 then
      MessageBox("Good job completing the test")
   end
end
```

Selecting the LUA Test Event Script for a Test Program

You can select one Test Event Script for a test program. Before selecting the Test Event Script, create the test program just as you would any other test program. To select a LUA Test Event Script for a test program:

1. Edit the Test Program. In the test editor press the **Set Test Defaults** tab.
2. Press **Select Test Event Script**.
3. Press the Attach button. Remember, a LUA Test Event Script must have a .lua file extension.



The script will load. Applicable error messages will be displayed.

4. **Note:** With Enable Script Changed Warnings selected in the window above, you will not be able to load the Test Program using the script if the script changes. Instead a warning window will be displayed. After you are through making frequent changes to a script, you can use this selection to help ensure the integrity of the test event script used by the test program.

5. Press **OK**.

EVT Test Event Scripts

Overview

An EVT Test Event Script can execute upon the test events shown in the table below. Each event has an associated value. The easy-wire software passes this value to the script when each of these events occurs.

Value	EVT Test Event
1	The first test event begins when the test window is initially entered. The test window is entered after the operator selects a test and presses **Test** in the main menu.
2	The second test event occurs when the low voltage test begins. The low voltage test is the first test that is performed on a cable and always occurs. If the Test Method is set to Signature Continuous Test, the low voltage test starts when the operator attaches a cable assembly. - OR - If the Test Method is set to Signature Single Test, the low voltage test starts when the operator presses Start in the Test Window.
3	The third test event occurs at the end of all tests on a cable. This includes the low voltage test, and if selected, the high voltage and intermittents test. If the Test Method is set to Signature Continuous Test, the end of all tests happens when the cable assembly is removed. - OR - If Test Method is set for Signature Single Test mode, the end of the test happens when the low voltage, and if selected, high voltage complete.
4	The forth test event occurs when the test window is open and the test is waiting to start. Note that this test event happens before each test cycle.
5	When the operator exits the Test Window to return to the main menu.

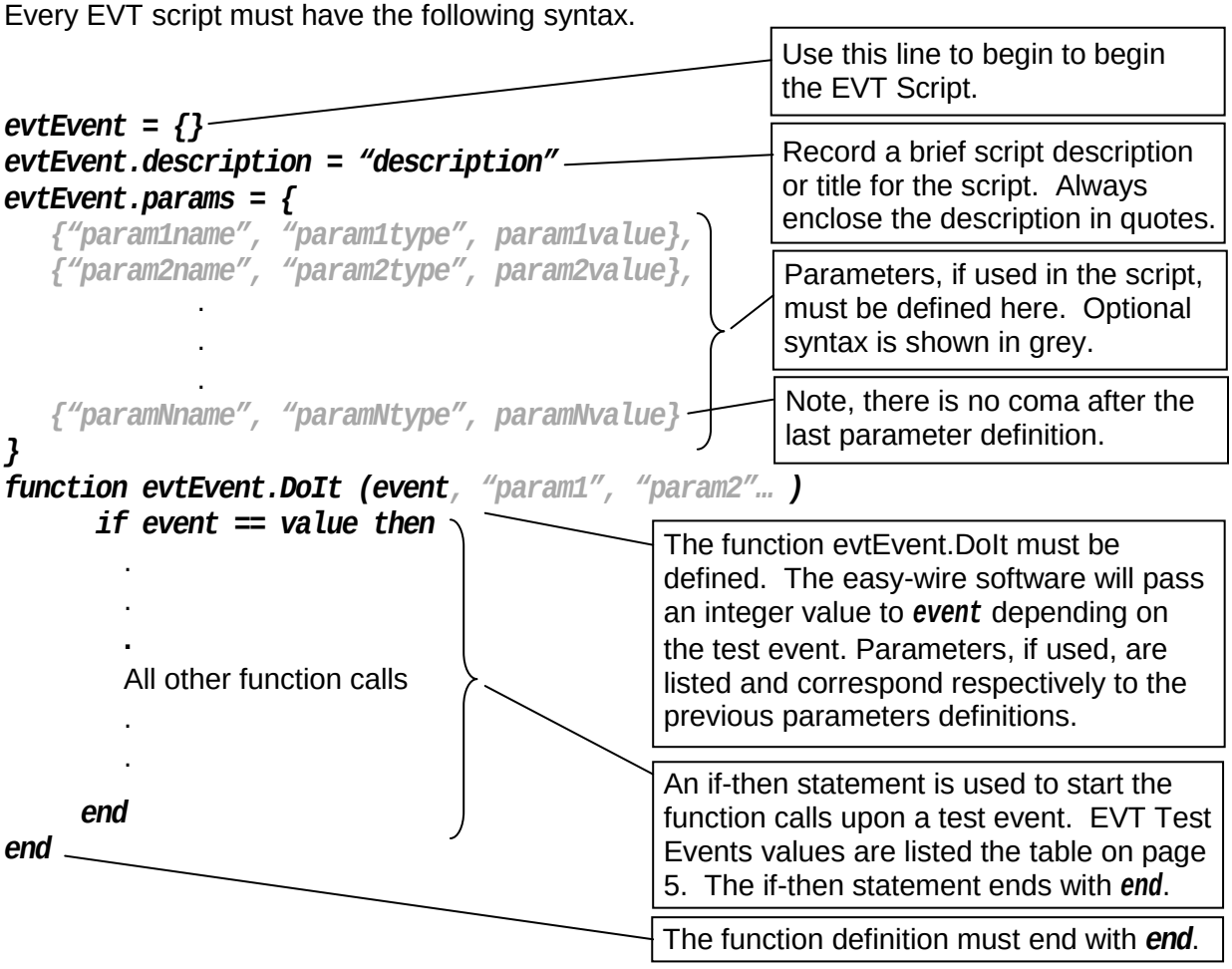
Unlike a LUA Test Event Script, an EVT Test Event Script accepts test parameters. The test parameters are values that are passed to the script, but can be easily changed in the easy-wire software's test editor. This also allows you to use the same EVT test event script in different test programs, but use different parameter values for each program as needed. For more information on parameter types see page 12.

There can be only one LUA or EVT Test Event Script used in a test program. If using parent child test programs, only the parent test program can use a test event script.

An EVT Test Event Script must end with an .evt extension and have the required syntax as described as follows.

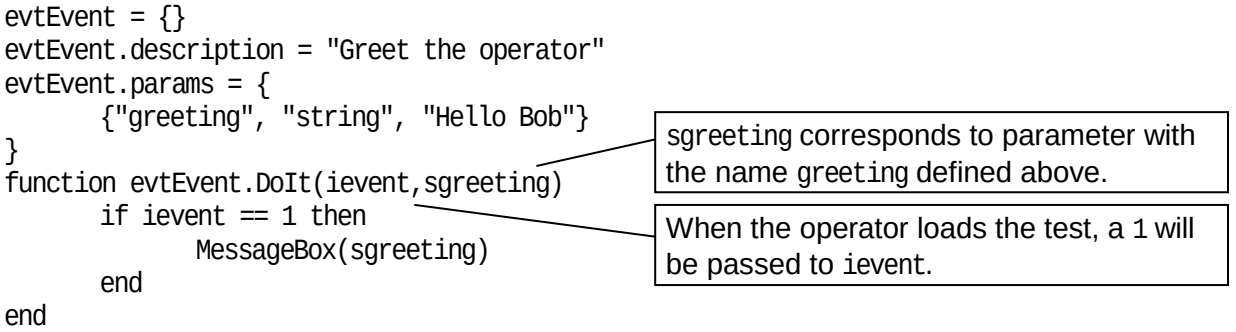
Required Syntax for an EVT Test Event Script

Every EVT script must have the following syntax.



Example of an EVT Test Event Syntax

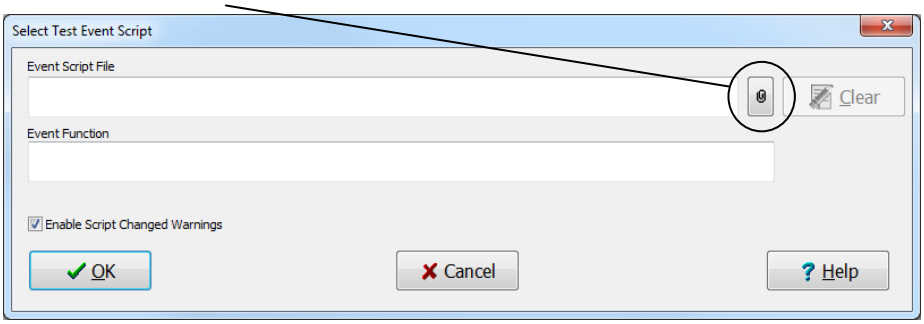
Here is a short EVT Script that would greet the operator, Bob, when he loads a test.



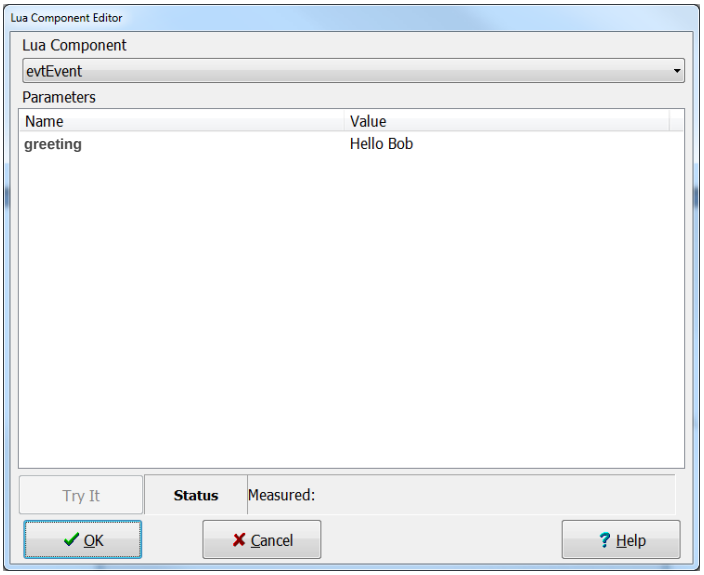
In this case the default value for the parameter `greeting` is `Hello Bob`. Maybe after several months of using this script, Jane is the new operator. The default value, `Hello Bob`, can easily be changed without modifying the script. Instead, the easy-wire Test Editor can be used to change the parameter value in the test program as shown in the next section.

Selecting the EVT Test Event Script for a Test Program

1. Edit the Test Program. In the test editor click the **Set Test Defaults** tab.
2. Press Select Test Event Script.
3. Press the Attach button. Remember, an EVT Test Event Script must have a .evt extention.

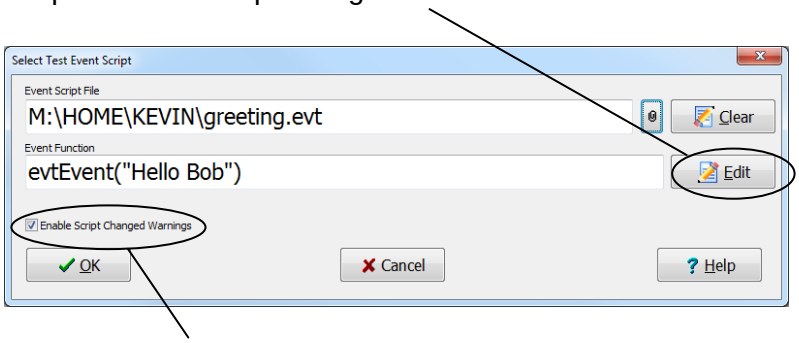


4. Select the script and press **Open**. Applicable error messages will be displayed. When the script successfully loads the parameter editor will open. If `Hello Bob` were used for the `greeting` parameter as in the preceding example, you would see the value as shown.



5. If desired, you could change “Hello Bob”. When you are satisfied with the value, press **OK**. The value will be used whenever the test is used. It is also possible for no default value to be used in the parameter definition. If no parameter is displayed, always enter a value before pressing **OK**.

At any time you can change the value of the parameter by returning to the Select Test Event Script Window and pressing **Edit**.



With **Enable Script Changed Warnings** selected, you will *not* be able to load the Test Program using the script if the script changes. Instead, a warning window will be displayed. After you are through making frequent changes to a script, make sure to check this selection to help ensure the integrity of the test event script used by the test program.

6. Click **OK** in the window above.

Parameter Types and Values

The table below shows the parameters that may be used in a EVT Test Event script.

Parameter Type	Ranges for Default Values	Description
“capacitance”	10 pF – 100 uF (10 pF increments)	Used for components that need a capacitance measurement. For capacitances outside this range, use the “number” parameter.
“current”	(Approximations) 0 = OFF, 1 = 3 uA 2 = 12 uA, 3 = 30 uA 4 = 110 uA, 5 = 376 uA 6 = 2 mA, 7 = 6 mA	Used to set the current that is applied to a test point.
“number”	Floating point: maximum four points before and after decimal	For components that need a numeric value. Also for capacitances, percentages, resistances, and voltages outside the range for these parameter types.
“percent”	1 – 99	For percentages outside this range, use the “number” parameter.
“point”	Number tag ties points together for hipot tests: < 0 means do not tie together, >= 0 means group points in this component with the same number tag together for hipot tests.	Any one test point in a test program. This point can be a fourwire point. This type can tie nets together. Points and point lists can also be tied together by assigning the same number tag.
“point list”	**< 0** = treat as a separate point (default) **≥ 0** = tie all points with this number tag together (can have multiple number tags)	Multiple test points in a wirelist that are common to one action. This type can be used to tie nets together. Points and point lists can also be tied together by assigning the same number tag.
“resistance”	.1 Ohms – 5 Megaohm	Resistance of a component. For milliOhm, four-wire, and values greater than 5Meg, use the “number” parameter.
“string”	30 characters, maximum	ASCI text variables
“textlist”		Allows the script to do different options depending on your selection. When creating a textlist parameter type, use text strings that will aid in identifying the test parameter. When the selection is made, there will be an index starting at one to identify which item in the list was selected. To access your selection, inside the test function use the first item in the input parameter list. For example: Button = {one ,two, three, four} Button[1] = the position in the array Button[2] = the text in that position
“voltage”	50 –1500, tester dependent	High voltage applied to a test point or net. For other voltages or setting external voltages, use the “number” parameter.

Component Scripts

Overview

You can add one component script to a test program. However, a component script can contain one or more “Lua Components”. You can use a Lua Component to test for a unique characteristic on the tested assembly, which characteristic could not be tested with the tester’s base capability. Just as the test instructions for wires, a Lua Component must be correctly completed for a test to pass. Therefore, a Component Script differs from other script types in that it can cause the test to fail.

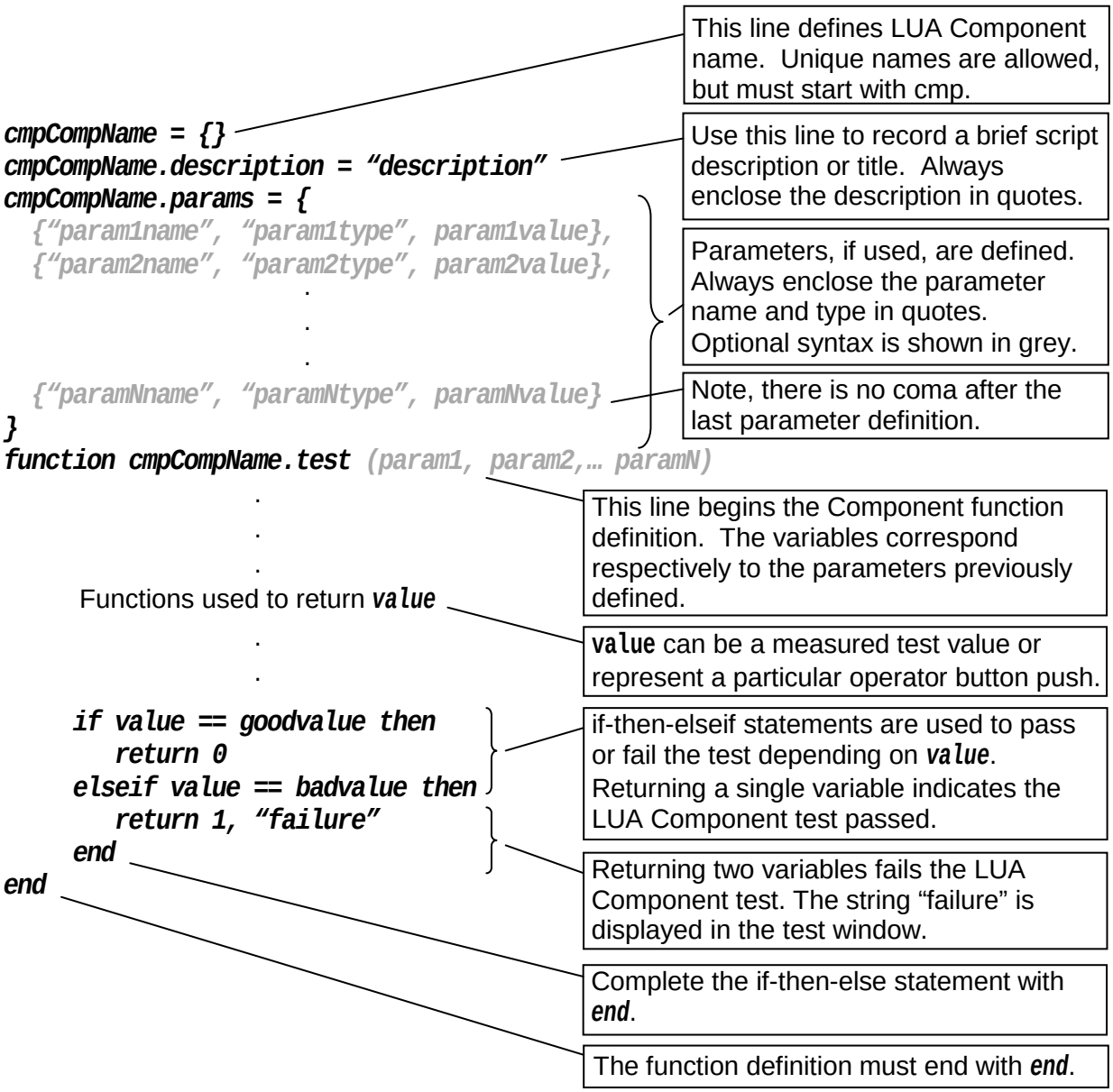
Component Scripts can use test parameters just as EVT scripts. The test parameters are values that can be used in the test, but can be easily changed in the easy-wire software’s test editor. For more information on parameter types see page 12.

One you add a Component script to a test program you one or more of the script’s Lua Components to the test program. In some instances you might even use the same Lua Component multiple times with different parameters in a test program. You add Lua Components to a test program in a similar manner to how you can add test instructions to a test program. Lua Components execute after the other test instruction have completed.

Component scripts must end with a .cmp file extension and use a defined syntax format as defined in the following section.

Component Script Syntax

Each LUA component within a Component Script file must end with a .cmp extension and have the required syntax as follows:



Parameter Types and Values

You can use the same parameter types and values in component scripts as you would in EVT Event Scripts. See page 12 for a list of these parameter types and values.

Example

The following custom component has the operator do a visual inspection. If the cable doesn't look good, the operator can reject the cable.

```
cmpInspection = {}
cmpInspection.description = "Visual Inspection"
cmpInspection.params = {
  {"Message Text", "string", "Cable look good?"},
  {"YesButton", "string", "Yes"},
  {"NoButton", "string", "No"}
}
function cmpInspection.test(sMessage, sYes, sNo)
    iResult = MessageBox(sMessage, sYes, sNo)
        if iResult == 1 then
            return 0
        else
            return 1, "Visual Inspection Failed"
        end
end
```

Name of this Lua Component is cmpInspection

Description of the LUA Component.

Parameters are defined in this section.

The parameters used in this function correspond respectively to the previous parameter definitions.

The parameters are used to present a message, and the buttons yes and no. The Message Box function returns a `1` to `iResult` if the first button is pressed; otherwise `2` will be returned.

Returning a single variable to the function will pass this inspection test.

Returning two variables fails the inspection test. The string "`Visual Inspection Failed`" display in the Test Window"

The component function is completed with an `end`.

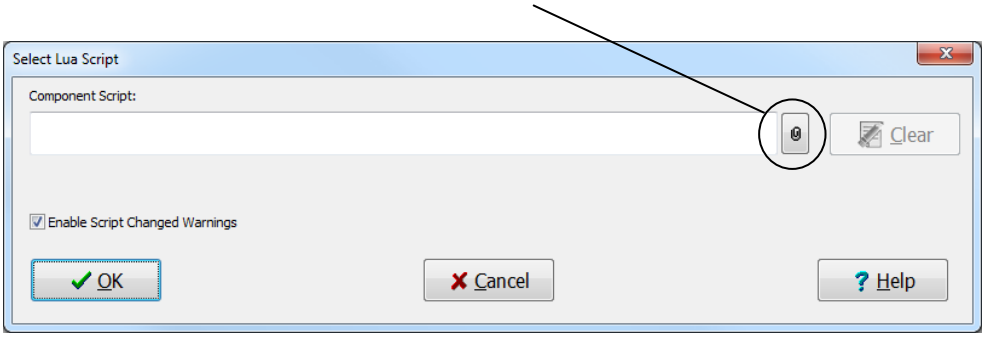
More Component Script Examples

You may download example scripts files shown in the table below from the Cirris community forum. See the page after the table of contents for instructions on downloading these examples. To transfer the script files to the Easy Touch tester you may use a USB drive, or if your Easy Touch is connected to a network, use a scripts folder on your network. Using a shared a network will allow you to easily make changes at a network station and then evaluate the results on an Easy Touch Tester that is connected to the network. Then when the script is stable, you can copy it to the Easy Touch.

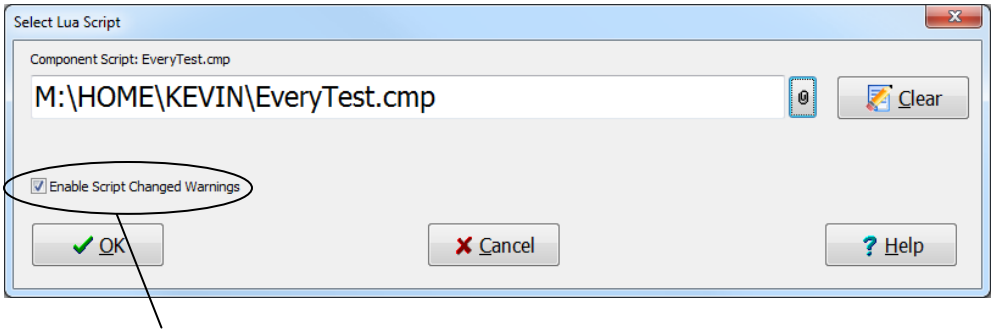
Hipot_pt_net.cmp	Applies hipot to one or more nets. The parameters of this Component allow you to select the points and hipot settings for this test.
LEDHC.cmp	Allows you to select the current for a higher current LED.
CustomLED.cmp	LED testing script

Inserting a LUA Component into a Test Program

1. In the test editor click the **Define Instructions** tab.
2. Press Select Component Script.
3. In the LUA Script window press the Attach button.

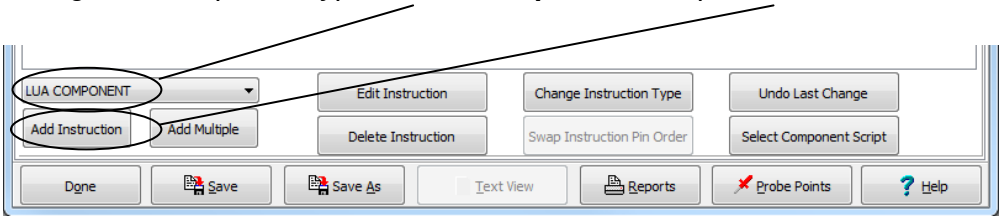


4. Navigate to the desired Component Script File. Only files with a .cmp extension will be displayed. Select the file and press **Open**. In the window below the Component Script File, EveryTest, has been selected.

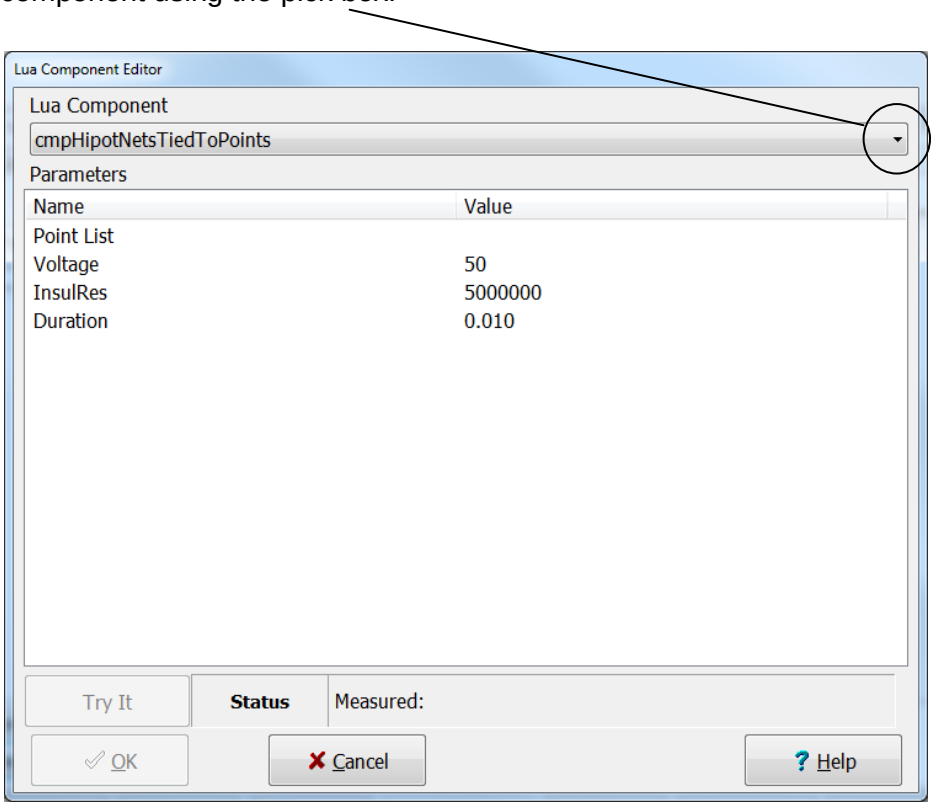


Note: With **Enable Script Changed Warnings** is selected, you will not be able to load the Test Program using the script if the script changes. Instead a warning window will be displayed. Use After you are through making frequent changes to a script, you can use this selection to help ensure the integrity of the test program using the test program.

5. Press **OK** to accept the selected Component Script file.
6. In the Test Editor select the test instruction under which you'd like to insert the LUA Component.
7. Change the component type to **Lua Component** and press **Add Instruction**.



8. If there is more than one component in the LUA script file, select the appropriate component using the pick box.



9. To change a parameter value double tap on the parameter line. A parameter value window will appear. Select an appropriate value and press **OK**.

10. When you have set all the parameter values as desired, press **OK** at the LUA Component Editor Window shown above.

    Note: The OK button will be grayed out until you have entered a parameter value for each of the parameters. In the example window above “Point List” has no default value. A value must therefore be entered before the values can be accepted.

11. Press **OK** at the Select LUA Script window to exit back to the Test Editor.

Custom Report Scripts

Overview

The easy-wire software used on the Easy Touch tester makes it easy to customize and print different standard reports and labels. However if you need to print a report or label outside the standard easy-wire software's capability, you can set up a Custom Report Script. Some custom reports may be better created using an EVT Test Event Script. We describe the advantages of the different report options below.

Standard Reports

The standard reports and labels created in the easy-wire software can be turned on, off, or configured separately *for each* test program.

Custom Report Script

A Custom Report Script will execute *for all* test programs used on the tester. A Custom Report Script can be setup to print after every good test, after every bad test, or after every tests – whether it be good or bad. Again remember, Custom report scripts are system wide applying to all test programs.

EVT Test Script

The advantage of using an EVT Test Script is that you have flexible scripting options, and you can make the custom report or label *for selected* test programs. Additionally, an EVT Test Script can allow you to print a custom report at the end of a test run.

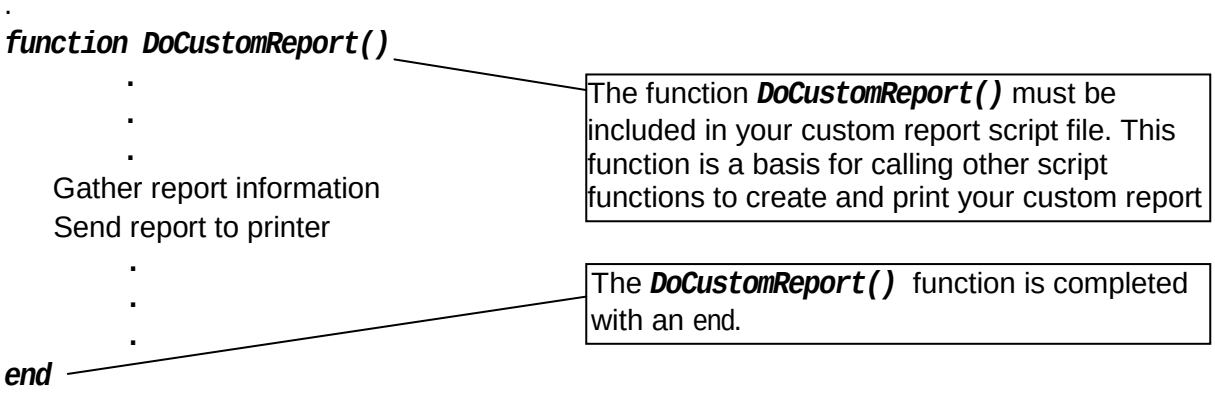
Setting up a Custom Report Script

A Custom Report Script can have one of three file names – autogood.rpt, autobad.rpt or autoall.rpt. The file name that is used determines when the Custom Report Script will run. The script autogood.rpt prints results after a good test; autobad.rpt prints results after a bad test; and autoall.rpt prints results after all tests. Again, remember Custom report scripts are system wide applying to all test programs.

To use a custom report script, the custom file - whether it be auotgood.rpt, autobad.rpt, autoall.rpt - must be in the appropriate folder location on the test system. For Vista and Windows 7, the script file must be stored the report folder on the path `C:\Users\Public\Documents\Cirris\easywire\Report` .

For older test systems using XP and 2000, the script file must be stored in the report folder on the path `C:\Documents and Settings\All Users\Documents\Cirris\easywire\Report` .

Custom Report Syntax



Custom Script Example

See the Cirris ftp site ftp://ftp.cirris.com/easytouch/scripts/ to see an example script file. The file T1Rep.rpt at this ftp site creates a Touch 1 style report for the Easy Touch tester.

Embedded Blocks

Who Should Read this Section

Intended audience

This section is intended for advanced scripting users or those with general programming experience. This section will provide details of how the embedded script blocks are implemented in order to make it easier to understand how to take full advantage of the functionality available.

Why you *might* need to use Embedded Blocks

The Easy Touch Tester has two processors involved with scripting. One processor resides on the PC motherboard; the other processor resides in the embedded system which directly controls the tester hardware. Each of these processors has their own memory and Lua environment. The PC processor is using a Lua 5.1 environment while the embedded processor is using Lua 3.2.

In normal operation, when a script is started in easy-wire software, the script begins running in the PC processor. Some of the Cirris measurement functions (such as GetResistanceMeasurement) are executed in the embedded processor so that the embedded system can directly control the tester hardware. When the PC processor script encounters such a command it transfers control to the embedded processor. The PC script then waits for the results of that function to return from the embedded processor. Each of these commands must be transmitted over the communication channel between the two processors, which is very slow compared to the execution speed of the processors.

In most scripts the time required to pass data between the PC and embedded processors is unnoticeable, but in some cases a script may run slower because of the time required for this data transmission. If such a function were executed only a few times during a script execution, the operator would not recognize a noticeable slowdown. However, if the function were called repeatedly for a hundred measurements (for the sake of getting an average, for example), the operator could notice a pause when the script executes.

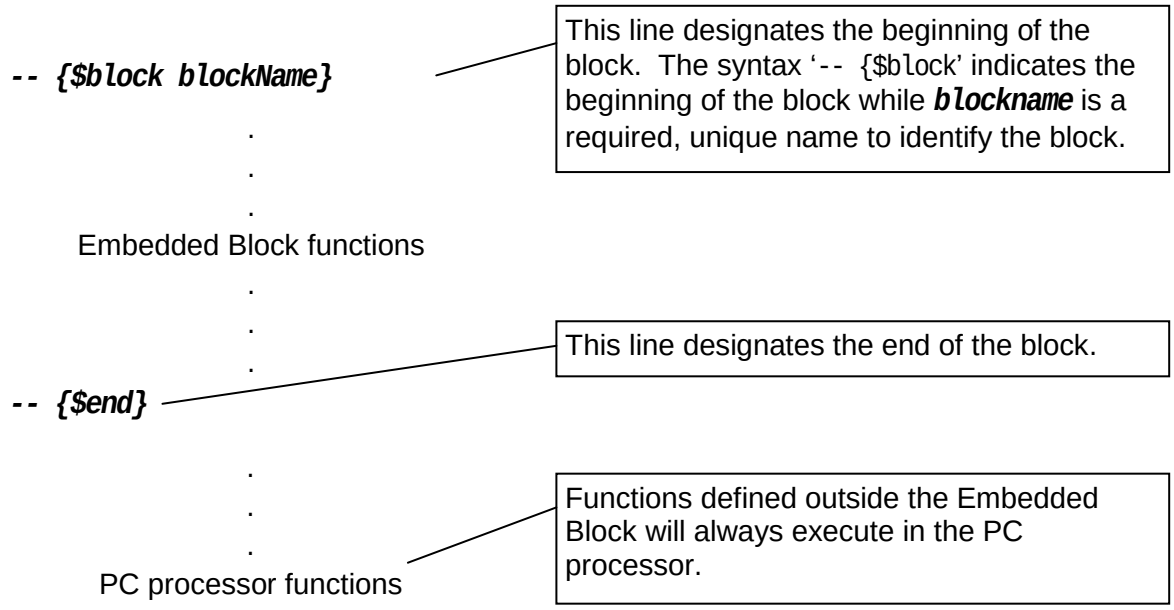
For this reason Easy Touch scripting allows the use of “embedded blocks”, in which an entire block of script commands can be designated to run exclusively together in the embedded processor. Using an Embedded Block can reduce significant data transmission and substantially increase the execution speed of the script. Thus if a speed critical section of a script is calling many Cirris measurement functions with little processing between them, using Embedded Blocks may be a good choice.

How Embedded Blocks are Implemented

Implementation

Embedded Blocks allow for user-defined blocks of code to execute in the embedded memory. This can significantly speed up script execution when many tester hardware commands are needed in tight loops. Before a script begins execution in the PC processor the script is preprocessed and everything defined in an embedded block is sent to the embedded processor's Lua environment. Thus everything within an Embedded Block declaration is defined in both environments including function definitions and variables. When an Embedded Block function is encountered in the PC script the PC processor transfers control to the embedded processor and waits for the function to return.

For example, an Embedded Block can be defined in the following manner:



This syntax is all that is required to make use of Embedded Blocks. With this example any functions defined in the Embedded Block will always execute in the embedded processor and nothing else is needed. This behavior is likely sufficient for most new scripts. However, Embedded Blocks have some additional functionality that may be useful for some new scripts and for adapting legacy scripts to make use of Embedded Blocks.

Embedded Blocks

There are times when it may be necessary, or advantageous, to execute an Embedded Block function on the PC processor. For instance, a function which does some processing of measurement results may be needed in script code that is run in the embedded processor, but may also be useful for code that is running on the PC processor. You could define a second function outside the Embedded Block with the same code, but such code is difficult to maintain. A better solution is to have the function defined in one place only and then be able to specify if the code should execute on the PC processor or the embedded processor. This behavior is accomplished through special preprocessing which takes place before the script is run in the PC processor.

When the preprocessor encounters a function defined in an Embedded Block, the function is first sent to the embedded processor in its original form. Then, in the PC script, the function is replaced with three functions which allow the function to be called in three different ways – the original function name, an embedded name, and a PC name. The embedded name causes the function to execute in the embedded processor while the PC name function will execute in the PC processor. The original function name will call either the embedded version or PC version depending on a user defined default behavior (described below).

As an example, consider the following code which represents the original script file before running the preprocessor.

```
--------------------------------------
-- {$block embBlockName }
--------------------------------------

function MyFunction(a, b)
       -- function code
end

--------------------------------------
-- {$end}
--------------------------------------
```

Required block syntax

Optional comment lines – used only to make block more visible.

After preprocessing, the script will look like essentially like the following (though some details are not shown to simplify the example):

Embedded Script

```
-------------------------------------
-- {$block embBlockName }
-------------------------------------

function MyFunction(a, b)
       -- original function code
end

-------------------------------------
-- {$end}
-------------------------------------
```

PC Script

```
-------------------------------------
-- {$block embBlockName }
-------------------------------------

function __embBlockName_MyFunction(a, b)
       -- original function code
end

function e__embBlockName_MyFunction(a, b)
       return -- call embedded version
end

function p__embBlockName_MyFunction(a, b)
       return __embBlockName_MyFunction(a,b)
end

myFunction = e__embBlockName_MyFunction


-------------------------------------
-- {$end}
-------------------------------------
```

Original function renamed. This allows for reassigning the original function name to selectively point to embedded or PC version.

Embedded name.

PC name.

This line reassigns the original function name to point to either the embedded or PC version depending on user specified default behavior. Here it is being assigned to the embedded version.

With this new version of the script the user has access to either the embedded version or the PC version of the function. Based on the example above, to call the PC version of the function the user would call

```
p__embBlockName_MyFunction(a, b)
```

and to call the embedded version the user could call either

```
e__embBlockName_MyFunction(a, b)
```

or more simply

```
myFunction(a, b)
```

since the default behavior allows for the original function name to call the embedded version. It is important to remember, however, that these new function names are only available in the new PC version of the script. Thus any function defined in an Embedded Block cannot use the alternate function names. It is also worth noting that if you use the alternative function names in your script code you will be referencing a function that does not exist until the EasyWire preprocessor processes the script. This could cause problems if you are using any external Lua syntax checking tools.

Important Notes

Because the code in an Embedded Block may execute on the embedded processor there are a few important considerations to remember when writing code for Embedded Blocks. First, because the embedded processor uses a Lua 3.2 environment, all code in an Embedded Block must be Lua 3.2 compatible. This includes ensuring that functions are defined before being called and not using newer Lua features that are not supported in 3.2 such as ‘for’ loops. A second important note is that since the code may run on the embedded processor, functions such as ‘MessageBox’, ‘DialogOpen’, and ‘PromptForUserInformation’ should not be used because the dialog boxes will be sent to the embedded processor display, which may not exist (in the case of an EasyTouch tester, for instance). These functions will cause the script to appear to be ‘hung’ because the script is waiting for user input which the user cannot give.

Global Variables

In addition to function definitions, global variables can also be defined in an Embedded Block. This could be useful for updating an existing script to run faster on an Easy Touch Tester, but is not advisable when writing a new script. When global variables are defined inside the Embedded Block both the embedded processor and the PC processor will have a unique copy of the variable, and thus the variable will no longer be truly global. If the variable is serving as a constant value that is never updated this is acceptable behavior and will allow both processors to see the constant value. However, if a global variable gets updated by one processor the change will not be represented in the other processor, which may cause unexpected behavior. Of equal importance to remember is that global variables not defined in an Embedded Block will not be defined in the embedded processor and could provide unexpected '`nil`' values if used in functions defined in the Embedded Block. The preferred way to transfer information between functions that may execute in different processors is simply to use the parameters and return values of those functions.

If you think there is a compelling reason to use global variables inside an Embedded Block you will have to provide a mechanism to ensure that the global variables in both processors remain synchronized. Some example code for such behavior is provided below.

```
--------------------------------------
-- {$block embBlockName}
--------------------------------------

myGlobal = 0

function ChangeGlobal()
	myGlobal = 100
end

function UpdateMyEmbeddedGlobal(NewValue)
	myGlobal = NewValue
end

function GetMyEmbeddedGlobalValue()
	return myGlobal
end

--------------------------------------
-- {$end}
--------------------------------------

function My_PC_Function()
	myGlobal = 5

	UpdateMyEmbeddedGlobal(myGlobal)

	ChangeGlobal()

	myGlobal = GetMyEmbeddedGlobalValue()
end
```

Synchronizing Functions – defined inside the Embedded Block to give access to embedded version of myGlobal

`myGlobal = 5` before execution
`myGlobal = 100` after execution

Changing default behavior

The default behavior for the original function name can be changed by adding a second, optional parameter to the block definition. By adding a ‘,1’ following the block name the default behavior will change such that the original function name will call the PC version of the function rather than the embedded version. The only change in the preprocessor output is on the line which reassigns the original function name as illustrated below. Note, the yellow highlighted text below is the only change from the script example on page 23.

Embedded Script

```
--------------------------------------
-- {$block embBlockName }
--------------------------------------

function MyFunction(a, b)
	local c
	c = a + b
	return c
end

--------------------------------------
-- {$end}
--------------------------------------
```

PC Script

```
--------------------------------------
-- {$block embBlockName,1 }
--------------------------------------

function __embBlockName_MyFunction(a, b)
	-- original function code
end

function e__embBlockName_MyFunction(a, b)
	return -- call embedded version
end

function p__embBlockName_MyFunction(a, b)
	return __embBlockName_MyFunction(a,b)
end

myFunction = __embBlockName_MyFunction

--------------------------------------
-- {$end}
--------------------------------------
```

With the 2nd parameter of the block definition set to ‘1’ the original function name is now set to refer to the PC version of the function.

The behavior of having the original function name refer to the embedded version of the function is the default when no parameter is provided following the block name. This behavior can also be achieved by providing a '0' as the second parameter, as illustrated below.

```
-- {$block embBlockName,0}
```

The most likely scenario for changing the default behavior of the block is when changing an existing script to work more efficiently on an Easy Touch Tester. In most scripts only a limited portion of the script will need to be put into an Embedded Block for speed improvement. However, the functions placed in the Embedded Block may call other functions that are used throughout the rest of the script which will need to run in the PC most of the time. In this case it may be beneficial to change the default behavior to have all Embedded Block functions run in the PC processor and then explicitly call the embedded versions in the few select cases where they are needed.

Calling sub-functions

In general there are two types of code in a script – code defined inside an Embedded Block, and code defined outside an Embedded Block. Functions defined in an Embedded Block may call other functions also defined in an Embedded Block. Code execution can also flow between code defined outside a block and code defined inside a block. Script execution always begins with code defined outside a block which executes on the PC processor. When execution moves from code defined outside a block to code defined inside a block we may say execution is 'entering' the block. Similarly, when execution flows from code defined inside a block to that defined outside the block we may say execution is 'exiting' the block. If more than one Embedded Block is defined in a script it is also possible to exit one block and enter directly into another block.

There are two ways in which execution can enter a block from code defined outside a block – calling the PC version of a block function or calling the embedded version of a block function. The path that is taken to enter the block determines on which processor that code will run. Once execution is started on a given processor it is desirable to continue execution on the same processor until exiting the block.

For instance, consider functions `functionA` and `functionB`, which are defined in an Embedded Block in the following example.

```
--------------------------------------
-- {$block emb}
--------------------------------------
function functionB()
	…
end

function functionA()
	…
	functionB()
end
--------------------------------------
-- {$end}
--------------------------------------
…
PC_function()
	…
	P__emb_functionA()
	…
end
…
```

In the function defined outside the block, `PC_function`, it is desirable to call `functionA` to run on the PC processor to save the overhead of transferring execution to the embedded processor. However, `functionA` calls `functionB`, which is also defined in the Embedded Block. If `functionB` executes in the embedded processor, there will be no benefit to calling `functionA` to run on the PC processor. Thus, the desired behavior in this case is to have `functionB` also execute on the PC processor since that is how `functionA` was called. However, if a different path of execution is taken in which `functionA` is running on the embedded processor, it would be equally inefficient to have `functionB` run on the PC processor. Therefore, the desired behavior is that when code execution enters a block it will continue executing on the same processor until it exits the block. This behavior is accomplished through some details of the script preprocessing which were hidden earlier. These details are explained in the following example for those who want to understand more fully, however, it is not essential to understand the details of how this accomplished and the example may be skipped. The important point to understand is that when calling code in an Embedded Block the code will execute on the same processor until it leaves that block even if other block functions are called.

Recall the original script file example used above.

```
--------------------------------------
-- {$block embBlockName }
--------------------------------------

function MyFunction(a, b)
	-- function code
end

--------------------------------------
-- {$end
--------------------------------------
```

The preprocessor adds an extra enable flag to track entering and exiting the block as illustrated in the full preprocessor output below (new details highlighted). The yellow highlighted text below shows changes from the script example on page 23.

Embedded Script

```
-------------------------------------
-- {$block embBlockName }
-------------------------------------
function MyFunction(a, b)
       -- original function code
end
-------------------------------------
-- {$end}
-------------------------------------
```

No changes are made in embedded script. Thus once execution in a block begins in the embedded processor all function calls will execute on embedded processor.

PC Script

```
-------------------------------------
-- {$block embBlockName }
-------------------------------------
__embBlockName_enable = true
function __embBlockName_MyFunction(a, b)
       -- original function code
end
function e__embBlockName_MyFunction(a, b)
       if __embBlockName_enable then
              return -- call embedded version
       else
              return __embBlockName_MyFunction(a, b)
       end
end
function p__embBlockName_MyFunction(a, b)
       __embBlockName_enable = false
       local temp = {__embBlockName_MyFunction(a,b)}
       __embBlockName_enable = true
       return unpack(temp)
end
myFunction = e__embBlockName_MyFunction
-------------------------------------
-- {$end}
-------------------------------------
```

Checks enable flag – executes on embedded processor if flag is true, otherwise calls PC version.

Enable flag is cleared to keep execution on PC processor.

Enable flag is set to allow for processing on the embedded processor in future calls. The default state for the enable flag is true.

The `temp = {…}` and `return unpack(temp)` simply ensure that all return values are returned properly.

Examining the preprocessor output reveals that the enable flag defaults to true – which enables execution to take place on the embedded processor. If execution enters a block by calling the PC version of the function this flag is cleared - which keeps execution from being transferred to the embedded processor for further Embedded Block function calls. It is also important to note that the enable flag is tied to the block name. This means that if more than one Embedded Block is defined in your code and you call functions in another block the enable flag will not affect the second block. For this reason it is best practice to keep all Embedded Block code in a single block or at least not allow interactions between blocks whenever possible. It is also worth noting that the enable flag is not intended for the user to access directly in a script.

Summary Examples

Consider the following code example for getting the average value of a measurement, which illustrates some of the principles presented in this section.

```
-------------------------------------
-- {$block emb}
-------------------------------------

function CalculateAverage(sum, samples)
	return (sum / samples)
end

function AverageResistance(myPoint1, myPoint2)
	local i = 0
	local sum = 0

	while i < 100 do
		sum = sum + GetResistanceMeasurement(myPoint1, myPoint2)
		i = i + 1
	end

	return CalculateAverage(sum, 100)
end

-------------------------------------
-- {$end}
-------------------------------------

function My_PC_Function()
	local myPoint1 = 'J1-001'
	local myPoint2 = 'J1-002'
	local fAverageValue = 0
	local fAverageTime = 0
	local i = 0

	while i < 5
		fAverageValue = AverageResistance(myPoint1, myPoint2)
		fAverageValueSum = fAverageValueSum + fAverageValue
		i = i + 1
	end



	fAverageValueSum = fAverageValueSum + p__emb_AverageResistance(myPoint1, myPoint2)

	return p__emb_CalculateAverage(fAverageValueSum / 6)
end
```

No second parameter after block name, original function name will refer to embedded version.

Since the embedded block will run in embedded processor (Lua 3.2 environment), this function must be defined before being used later in the block.

This calls the embedded version of `AverageResistence` which will call the embedded version of `CalculateAverage`.

Calls PC version of `AverageResistence` which will call the PC version of `CalculateAverage`.

The `CalculateAverage` function in the embedded block is called to run on the PC processor.

The following example executes the same as the one on the previous page, but uses the opposite default behavior for the Embedded Block. The yellow highlighted text below shows the changes from the previous script.

```
--------------------------------------
-- {$block emb,1}
--------------------------------------

function CalculateAverage(sum, samples)
	return (sum / samples)
end

function AverageResistance(myPoint1, myPoint2)
	local i = 0
	local sum = 0

	while i < 100 do
		sum = sum + GetResistanceMeasurement(myPoint1, myPoint2)
		i = i + 1
	end

	return CalculateAverage(sum, 100)
end

--------------------------------------
-- {$end}
--------------------------------------

function My_PC_Function()
	local myPoint1 = 'J1-001'
	local myPoint2 = 'J1-002'
	local fAverageValue = 0
	local fAverageTime = 0
	local i = 0

	while i < 5
		fAverageValue = e__emb_AverageResistance(myPoint1, myPoint2)
		fAverageValueSum = fAverageValueSum + fAverageValue
		i = i + 1
	end


	fAverageValueSum = fAverageValueSum + AverageResistance(myPoint1, myPoint2)

	return CalculateAverage(fAverageValueSum / 5)
end
```

1 as second parameter after block name, original function name will refer to PC version.

This calls the embedded version of AverageResistence which will call the embedded version of CalculateAverage.

Calls PC version of AverageResistence which will call the PC version of CalculateAverage.

The CalculateAverage function in the embedded block is called to run on the PC processor.

Script Errors & Debugging

Common Script Errors

"ERROR: Script file <script file> changed"

This message appears because the script was changed and the "Enable Script Changed Warnings" option was selected when the script was setup. This option used to help ensure the integrity of the test program using the test script.

To remove the message your can either:

- Change or replace the script so it is back to its original state.
- Edit each test program that uses the script. In the Test Editor either unselect the "Script Changed Warnings" option or reselect the changed script file. Before reselecting the script file, make sure to note any parameters that are used so you can re-enter them accurately after reselecting the script.

The Event or Component Script file does not run

In this scenario you have tested cables, but the script file will not run for one of two reasons:

- The Script file has not been selected in the test's setup.
- The script file contains syntax errors, which need to be corrected. To correct these errors, see *Debugging Methods* on the next page.

Custom Report Script does not print

Potential reasons include:

- The Custom Report Script is in the wrong folder.
- The Custom Report Script has the wrong file name.
- The printer is not connected to the tester.
- The wrong printer (serial or parallel) is connected to the tester.
- The printer is off-line.
- The printer is turned off.
- The printer is out of paper.

Debugging Methods

Once you have written or edited a script, you may find the syntax is not correct or it simply does not work. Here are a few techniques you can use to get your script running.

- Put the script on your network where it can be accessible to both the tester and a network workstation. That way you can evaluate your script changes quickly. Make sure “Enable Script Changed Warnings” is not selected. That way you won’t have to re-select the script after each script change. If a network is unavailable to an Easy Touch tester, make and evaluate script changes on a USB drive. Another option is to attach a keyboard to the Easy Touch tester.

- Use the Cirris Message Box function to display the value of variables at one or more points in the script’s execution.

- Revert to the simplest thing that you can get to work. Comment out code, or save a copy of the script so you can delete sections of the code. Introduce more script functionality to find script errors.

- After you’ve eliminated all the syntax errors, continue to test the script to ensure it works as intended and maintains its functionality under different scenarios.

Cirris Functions

Cirris functions organized by category

Cirris functions organized by category continued

Test Information Functions

User Interface Functions

1100 Embedded Memory Functions

Cirris Functions organized alphabetically

Cirris Functions organized alphabetically continued

Note: Some Cirris functions previously used in scripting for Cirris 1100 and Touch 1 testers are no longer supported in Easy Touch Scripting (See Unsupported Cirris Functions page 106). Some Lua 3.2 functions previously used in Cirris scripts continue to be supported (See Preserved Lua 3.2 Functions on page 105).

Date and Time Functions

Delay (delayseconds)

Sets a delay for tester functioning in seconds. Use this function when controlling relays using the low level commands and a delay is needed to let the tester catch up. This function acts like a NOOP operation. Also, see the SetDelayTimeInMilliseconds function.

For example the following lines of code set a 5 second delay

```
iDelayInSeconds = 5.0
Delay(iDelayInSeconds)
```

Another example, `Delay(.1)` delays tester functioning for 100 milliseconds.

GetDateAsText (select)

This function returns the current date information as a text string. You can select the format for the returned date information as indicated in the table below.

date information returned	*select*
function describes itself	nothing
Month only MM (no preceding 0)	***1***
Day only DD (no preceding 0)	***2***
Year only YYYY	***3***
Year only YY	***4***
Full Date MM/DD/YYYY	***5***
Full Date DD/MM/YYYY	***6***

Example

```
sCurrentDate = GetDateAsText (1)
```

If July were the current month, the function would return the string “5”.

SetDelayTimeInMilliseconds(delay)

This function sets a delay time for tester functioning in milliseconds. See also the function Delay for setting a delay time in seconds.

Examples:

```
iInputNum = 5
SetDelayTimeInMilliseconds(iInputNum)
```

The function will delay all functioning on the tester for 5 milliseconds.

```
SetDelayTimeInMilliseconds(10)
```

The function will delay all functioning on the tester for 10 milliseconds.

TimePassed (timer)

This function returns the elapsed time since a timer specified by the handle ***timer*** was opened or reset. The timer handle is integer. Time is returned as an integer value represented in milliseconds.

Example:

```
local iTimerHandle = TimerOpen(500)
SetDelayTimeInMilliseconds(300)
local iTimePassed1 = TimePassed(iTimerHandle)
if iTimePassed1 > 200 then
       MessageBox(“READY TO START”)
end
TimerClose(iTimerHandle)
```

This example will create a timer with a 500 millisecond timeout. If the time has been greater than the 200 msec, a message box will display. The timer will then be terminated.

TimerClose(timer)

This function closes the timer specified by the handle ***timer***, freeing one of the ten available timers. Timers are created using the function TimerOpen.

TimerDone(timer)

This function returns ***0*** if the timer specified by the handle ***timer*** has timed out, or the function returns ***1*** if more time is remaining. A timer is created using the function TimerOpen.

TimerReset (timer)

This function restarts the timer using the original timeout delay that was determined when the timer was created. The function uses the timer's handle ***timer***, an integer value created by the TimerOpen function.

Digital Input and Output Functions

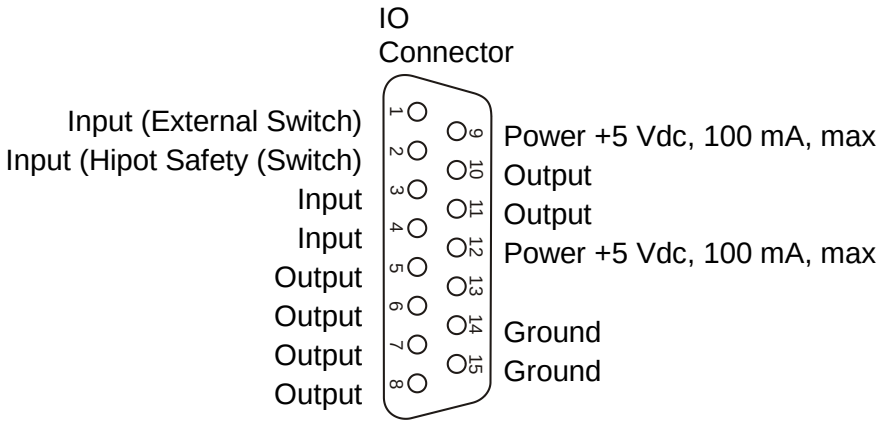
GetUserOutputStates

The full format of this function is:

output1[, output2,… output8)] = GetUserOutputStates (select1[, select2,… select8])

The function returns the state of the digital outputs on the tester's I/O port. If desired, you can specify multiple outputs and this function will return their respective states. This function will return a 0 if the output is low, or 1 if the output is high.

The state of the output port is determined by using the SetUserOutputStates function or by using the setup for Digital Outputs on the tester.



select **(integer)**	**IO Connector Pin**
1	5
2	6
3	10
4	11
5	7
6	8
7	7
8	8

ReadUserInputStates

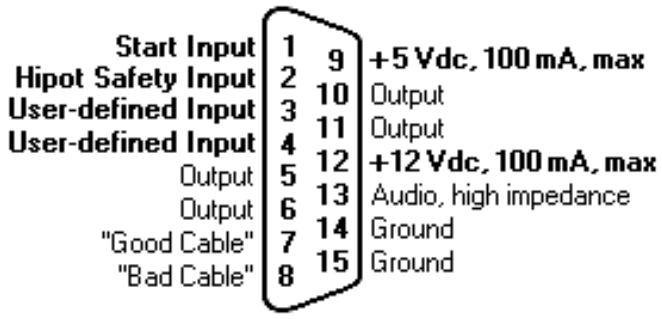
The full format of this function is:

state1, state2, state3, state4 = ReadUserInputStates(1, 2, 3, 4)

This function returns the state (on or off) of one or up to all four digital inputs on the digital I/O port. Each returned input state corresponds respectively to the specified input. A 0 is returned if the input is on (conducting). A 1 is returned if the input is off (not conducting). Each input is specified by an input select number as shown in the table below.

input select number	description
1	Input 1, External Start Switch (Pin 1)
2	Input 2, Hipot Safety Switch (Pin 2)
3	Input 3, User-Defined (Pin 3)
4	Input 4, User-Defined (Pin 4)

The figure below shows the digital IO port as viewed from the back of the tester.



The settings for External Start Switch and Hipot Safety Switch affect this function if they are turned on.

Examples:

```
externalStartSwitchState = ReadUserInputStates(1)
```

The function will get the state of the external start switch.

```
io1,io2,io3,io4 = ReadUserInputStates(1,2,3,4)
```

The function returns the values of all four inputs to io1, io2, io3, and io4.

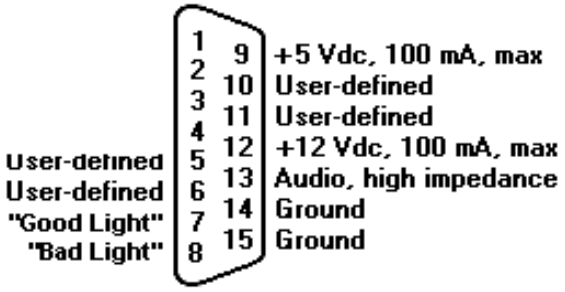
SetUserOutputStates

The full format of this function is:
SetUserOutputStates(outnum1, set1, outnum2, set2, … outnum8, set8)

This function sets the digital outputs on the user I//O port to a particular low or high logic state. This function takes up to six parameter pairs. In each pair, the first parameter ***outnum*** specifies which output to be set. The relation between output numbers and the io connector pin out on the back of the tester is shown in the table and figure below. The second parameter, ***set*** specifies whether the output is set on or off. If this parameter value is 0 the output is set on (conducting); if 1 the output is set off (not conducting). Note that the function GetUserOutputStates allows you to read the state of the outputs.

output number	description
1	Pin 5
2	Pin 6
3	Pin 10
4	Pin 11
5	Pin 7 - Good Light
6	Pin 8 - Bad Light
7	Pin 7
8	Pin 8

Output pin outs



Examples:

```
iDigitalOutput1 = 1
iOutputState1 = 0
iDigitalOutput2 = 2
iOutputState2 = 1
SetUserOutputStates (iDigitalOutput1, iOutputState1,iDigitalOutput2, iOutputState2)
```

The function above will set the pin 5 output on, and the pin 6 output off.

```
SetUserOutputStates(6,0)
```

The function will turn on the Bad Light led.

File Functions

Cirris.ChDir (x:\\directory)

This function changes the current directory to the drive, path and directory specified by the input string ***x:\\directory***. This allows the drive, path and directory to not have to be restated in subsequent file functions.

The function also returns two outputs, ***result*** and ***error*** as shown.

result, error = Cirris.ChDir (x:\\directory)

If the function is successful in making the directory, it returns ***1*** to ***result*** and ***nil*** to ***error***. If unsuccessful, the function returns a ***0*** to ***result*** and a string describing the error to ***error***.

Example:

```
iCopyGood, sCopyError = Cirris.ChDir (“c:\\junk”)
```

The function attempts to make the current directory `c:\junk` for further file functions. If successful, `1` will be returned to `iCopyGood` and `nil` to `sCopyError`.

Note: When using Easy Touch Scripting to control an 1100 tester, this command affects only the current directory of the computer controlling the tester, not the directories in the embedded memory of the tester.

Cirris.CloseDir (handle)

This function closes the current directory that is identified by a handle previously specified by the Cirris.OpenDir function. The handle input to this function is a string.

Example:

```
DirHandle=Cirris.OpenDir (“c:\\junk”)
Cirris.CloseDir (DirHandle)
```

The functions above open and close the directory c:\junk.

Cirris.CopyFile (x:\\source, x:\\destination)

This function copies a source file from the drive and path specified to the destination with the specified drive and path. Both the source and destination are specified as strings.

For example,

```
Cirris.CopyDir (“c:\\Cirris\\Data”, “c:\\Company\\Data”)
```

Cirris.CurrentDir

Returns a string for the value of the current directory. The current directory is set previously set using the Cirris.ChDir function. This function will also return ***result*** and ***error*** as shown in the full function definition below.

result, error, = Cirris.CurrentDir

If the function is successful in making the directory, it returns ***1*** to ***result*** and ***nil*** to ***error***. If unsuccessful, the function returns a ***0*** to ***result*** and a string describing the error to ***error***.

For example,

```
Cirris.ChDir ("c:\\junk")
sCurDir = Cirris.CurrentDir
```

Makes `sCurDir` equal to `c:\junk`.

Cirris.DirExists (x:\\directory)

Returns 1 if the directory on the given path exists or 0 if it doesn't.

For example, iDirThere = c:\\ProgramFiles\\Cirris\\Mydir
returns a `1` to `iDirThere` if the directory `c:\ProgramFiles\Cirris\Mydir` exists.

Cirris.MkDir (x:\\directory)

Creates a new directory as specified. If the directory given in the path does not yet exist, Cirris.MkDir attempts to create it. Cirris.MkDir returns ***1*** if it successfully creates all necessary directories, ***0*** if it could not create a needed directory.

For example, `iDirMade = Cirris.MkDir (c:\\junk)`
if successful, makes the directory `c:\junk` and returns a `1` to `iDirMade`.

Cirris.OpenDir (x:\\directory, [files, subdirectories])

Returns a handle for the specified directory. The handle can be used for the Cirris.ReadDir. The optional input ***mask*** allows you to specify a string such as `*.txt` to limit the available files of the opened directory. If optional input ***subdirectories*** is set to ***0*** subdirectories will not be available. If ***subdirectories*** is ***1*** or is unused, ***subdirectories*** are available. After opening and using a directory, use the CirrisCloseDir function to free memory resources.

For example, `iJunkHandle=Cirris.OpenDir ("c:\\junk")`
opens the directory `c:\junk` and returns the directory handle to `iJunkHandle`.

Cirris.OpenFileDialog

This function opens a dialog box so the user can browse to and select a file. To select a file the user clicks the file and clicks OK. The function returns the path and name of the selected file as a string.

Cirris.OpenFolderDialog

This function opens a dialog box so the user can browse to and select a folder. To select a folder the user clicks on a folder and clicks OK. The function returns the path and name of the selected folder as a string.

Cirris.ReadDir (handle)

This function provides information on the contents of a directory. The directory must be referenced by ***handle***. Previous to using Cirris.ReadDir, use the function Cirris.OpenDir to return ***handle***. The Cirris.ReadDir function is used in a while loop. Each time the function executes it attempts to read one entry of a directory. The function returns the name of a file or directory, and returns ***0*** for a file or ***1*** for a directory. After reading all of the entries of a directory, Cirris.ReadDir returns a nil value to the directory name.

Examples:

```
iDirHandle, sDirError = Cirris.OpenDir("C:\\Users\\Public\\Documents")
sDirEntries = ""
bDone = 0
  while bDone == 0 do
    sReadName, bIsFolder = Cirris.ReadDir(iDirHandle)
      if sReadName == nil then
            bDone = 1
      else
        sDirEntries = sDirEntries..Format("Name: %s, Folder: %d\r\n", sReadName, bIsFolder)
      end
   end
MessageBox(sDirEntries)
```

The code above opens the directory `Documents`. The `Cirris.ReadDir` function is used inside a while loop to read and identify each of the directory entries. Each entry name and its folder/file designation is added to the string `sDirEntries` using the Lua Format function. The MessageBox function is then called to display this string. Each file name will have a `0` displayed next it; each folder will have a `1`.

Cirris.RmDir (directory)

Removes the specified directory.

For example, `Cirris.RmDir ("c:\\junk")` removes the directory `c:\junk`.

Low Level Function

LowLevelCmd

The Low Level command allows you to control the bare internal workings of the tester to extend its testing methods. These low level commands are designed for use by a very experienced programmer or engineer. Read the entire section thoroughly before attempting to use these functions. After running a script that has executed any of the low level commands, you should call LowLevelCmd (9) to return the tester to a known state. This is the appropriate cleanup function.

Critical: While using these function to test complex circuitry, never connect an outside power supply over 5.5 volts to the testers test points. If you break this rule, you will void the tester's warranty and damage the tester. If you follow this rule, it is impossible for the commands listed below to damage the tester.

Low Level Function: Sink / Unsink

Sink means pull the point to ground. You can sink as many points as you like but you have to call this function for each point you sink. Once a point is sunk, it remains sunk until you clear it. The low level function format to sink or unsink a test point is:

result, error = LowLevelCmd(command, point, set)

where the inputs and outputs are defined as follows:

command	*point*	*set*	*result*	*error*
1 = Sink or Unsink Point Sink means pull the point to ground.	Text string containing the point as a default or custom label OR Integer containing the point where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter).	**0** = Unsink **1** = Sink	**0** = error **1** = no error	**nil** = No error **1** = Bad point **5 =** Bad command #

Example:
result = LowLevelCmd(1,"J1-001", 1)
This example sinks point J1-001.

Low Level Function: Turn On Relay

This low level command allows you to turn on a relay that controls a scanner point. Why would you turn on a test point relay? The test point relays switch out the analog hardware on a scanner point and connect the point to the high voltage system. The high voltage hardware can be used for hipot testing or measuring capacitance. The current source can be routed through the high voltage system, allowing you to apply current to more than one point at a time for complex measurements.

Notes:

a. You can turn on as many relays as you want but you have to call this function for each point. The relay remains on until you turn it off.
b. You cannot directly measure the voltage on a point when the relay is turned on for that point.
c. It takes time for a relay to turn on. You can turn on a lot of relays and then call Delay (0.010) to give the relay's contact time to move before performing any measurements.

The low level function format to turn on or off a relay is:

result, error = LowLevelCmd(command, point, set)

where the inputs and outputs are defined as follows:

command	***point***	***set***	***result***	***error***
2 = Turn Relay On or Off	Text string containing the point as a default or custom label OR Integer containing the point where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter).	**0** = Off **1** = On	**0** = error **1 =** no error	**nil** = No error **1** = Bad point **2** = Bad integer **5** = Bad command #

Example:

result = LowLevelCmd(2, 3, 0)

This example turns off the relay connected to point, J1-003.

Low Level Function: Source / Clear Source Vector

This low level command is needed when you want to measure devices that are nonlinear and require a fixed current, or when you want to source one point and measure voltage on another and therefore cannot use the resistance measurement command.

Note: Only one point can be connected to the current source at a time. Whenever you pick a new point, the current is removed from any prior point. This command doesn't actually turn on the current source but simply connects it to the point you choose. It does not affect the current source just the current source vector.

The low level function format to source clear a vector is:

result, error = LowLevelCmd(command, point, set)

where the inputs and outputs are defined as follows:

command	***point***	***set***	***result***	***error***
3 = Source or Clear Point	Text string containing the point as a default or custom label OR Integer containing the point where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter).	**0** = Clear source vector. **1** = Source	**0** = error **1 =** no error	**nil** = No error **1** = Bad point **5** = Bad command #

Example:
result = LowLevelCmd(3, "J3-001", 1)
This example sources the point, J3-001.

Low Level Function: Read / Clear Read Vector

This low level command operates similar to the source function. It points the read vector (vmpoint) to the desired point but does not actually read anything.

Note: Currently, there is no reason to call this function. It is here for future use.

command	***point***	***set***	***result***	***error***
4 = Read or Clear Point	Text string containing the point as a default or custom label OR Integer containing the point where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter).	**0** = clear **1** = read	**0** = error **1 =** no error	**nil** = No error **1** = Bad point **2** = Bad integer **5** = Bad command #

Low Level Function: Set Current

This low level command simply turns on/off the current source and sets its value. It does not send the current anywhere. Use the `SetSourceVector` function to send the current through the normal analog channels or use the `RouteCurrentToRelay` and `TurnOnRelay` functions to send the current through the relays. The input currents given below are approximate. The actual current (in amperes) is returned in the result.

The low level function format to set current is:

result, error = LowLevelCmd(command, level)

where the inputs and outputs are defined as follows:

command	***level***	***result***	***error***
5 = Set Current On or Off	(approximations) **0** = current off **1** = 3 uA **2** = 12uA **3** = 30 uA **4** = 110uA **5** = 376uA **6** = 2mA **7** = 6mA	Floating point number of the actual current in amperes **0** = error	**nil** = No error **2** = Bad integer **4** = Current out of range **5** = Bad command #

Example:
fCurrent = LowLevelCmd(5,7)
This example turns on 6 mA of current.

Low Level Function: Measure Voltage

The greatest accuracy for this low level command will be obtained if at least one of the points is at two volts or less with respect to ground. This is always the case if that point is sunk. See `LowLevelCmd 1` on how to sink a point.

If you are using relays to route the current to a point, you cannot directly measure its voltage using this command. Add an external connection to tie that point to another point on the tester so you can measure its voltage there.

Note: YOU CANNOT MEASURE ANY VOLTAGE GREATER THAN 5.5 VOLTS. APPLYING EXTERNAL VOLTAGE TO THE TESTER CAN DESTROY IT.

The low level function format to measure voltage is:

result, error = LowLevelCmd(command, highpoint, lowpoint)

where the inputs and outputs are defined as follows:

command	*highpoint lowpoint*	*result*	*error*
6 = Measure voltage between two points OR one input to just measure the voltage on that point	For each of two points: Text string containing the point as a default or custom label OR Integer containing the point where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter).	Floating point number containing the measured voltage (positive or negative) in volts. **0** = error	**nil** = No error **1** = Bad point **2** = Bad integer **5** = Bad command #

Example:

```
fVoltage = LowLevelCmd(6, 3, "J1-001")
```

This example will return the measured voltage between `J1-003` and `J1-001` if using an AHED-64 adapter. The first point given is expected to be the high point similar to the "red" lead on a voltmeter.

Low Level Function: Route Current to Relay

Use this low level setting if you want to perform four-wire resistance measurements using the low current sources, or if you want to source more than one point at a time. The current does not normally route through the relays. For a normal resistance measurement, leave the current in its default state.

> Use the Delay (0.010) command after calling this command to allow time for the current source to be switched over.
> The normal resistance measurement and other functions won't work properly if you forget and leave the current routed through the relays.

The low level function format to route current to relay is:

result, error = LowLevelCmd(command, through)

where the inputs and outputs are defined as follows:

command	*through*	*result*	*error*
7 = Route Current through Source Vector or through Relays	**0** = through normal source vector (default state) **1** = through relays The tester normally vectors the current through the source vector. **5** = SCSI scripts require that we test without relays connected to the current source. 1500V hardware has a wire connecting them unless we turn on a relay on the HV card connecting the HV source to the outside.	**0** = error **1 =** no error	**nil** = No error **1** = Bad point **2** = Bad integer **5** = Bad command #

Example:

```
result = LowLevelCmd(7, 1)
```

This example will route the current through the relays.

Low Level Function: Master Clear

Use this low level setting to unsink and turn off relays for all points. It disconnects the current source from the test points. It does not affect where the current is routed (to relays).

The low level function format for Master Clear is:

result, error = LowLevelCmd(command)

where the inputs and outputs are defined as follows:

command	*result*	*error*
8 = Master Clear	**0** = error **1 =** no error	**nil** = No error **5** = Bad command #

Example:

```
result = LowLevelCmd(8)
```

This example will unsink and turn off relays for all points.

Low Level Function: Set All Default

Use this low level setting to reset the current source in addition to the Master Clear setting. Use this function to return the tester to its normal operating mode. It is a good idea to call this after finishing a script containing low level commands so the tester does not get confused with any stuff set up within the script. Also, use this command in the middle of a script to restore the hardware to a default state. This function does not affect any digital I/O or the serial port.

The low level function format to set all default is:

result, error = LowLevelCmd(command)

where the inputs and outputs are defined as follows:

command	*result*	*error*
9 = Set All Defaults	**0** = error **1 =** no error	**nil** = No error **5** = Bad command #

Example:

```
result = LowLevelCmd(9)
```

This example will reset the tester for further testing.

Low Level Function: Set High Current

Use this low level setting to set the high current source. It can be used to measure highly inductive devices using the high current source since you can put in your own stabilization time. Using the High Current Source requires that you instruct the High Voltage card to route the current onto the HV buss. You also need to engage relays to source and sink the current. This means the current must flow out of and back into the High Current Source electronics. Due to the restrictions for using this command, it should be used with Cirris Systems' engineer direction only. Contact Cirris for application details.

Note: The high current source must only be used for short time periods. Turn it off as soon as you are done measuring or it can damage the circuit on the high voltage card.

The low level function format to set all default is:

result, error = LowLevelCmd(command)

where the inputs and outputs are defined as follows:

command	***current***	***result***	***error***
12 = Set High Current	**0** = Turn current source off or Floating point number containing the current in amperes. Current must be > .009 & < 1.1 amps	**-1** = Current source off or Floating point number containing the measured current in amps or **0** = error	**nil** = No error **1** = Bad point **2** = Bad integer **5** = Bad command # **6** = Bad float

Example:

```
local fCurWanted = 1.0      – default current to 1 amp
if aCur[1] = = 2 then
       fCurWanted = 0.25    – want current at ¼ amp
end
local fCur = LowLevelCmd(12, fCurWanted) – turn current on
local fVolts, err = LowLevelCmd(6, wSenseHi, wSenseLo)
if (err ~= nil) and (err ~= 0) then
       error("Invalid test point(s)")
end
LowLevelCmd(12, 0)          – turn current off
LowLevelCmd(9)              – reset tester to default state
```

This function sets the high current source to ¼ amp. It measures the voltage and then turns the high current source off. It then resets everything back to a default state.

Measurement and Test Functions

GetCapMeasurement (point1, point2)

The full format of this function is:
capacitance, error = GetCapMeasurement (point1, point2)

This function returns the measured capacitance between two points. The points can be integer values representing system points, or strings representing point labels. Depending on whether the points are valid, the function returns ***capacitance*** and ***error*** as shown in the table below.

	capacitance	***error***
If the input points are valid	Floating point number representing the capacitance measurement	***Nil***
If the input points are not valid	***0***	***-99***

It is good practice to check that error ***value*** is ***nil*** before using it in further calculations.

Example:

```
sPoint1 = "J1-001"
sPoint2 = "J2-005"
(fCapacitance, iError = GetCapMeasurement(sPoint1, sPoint2)
if(iError == nil) then
   MessageBox("Capacitance between %s and %s is %.2f nF", sPoint1, sPoint2,   fCapacitance/1e-9)
end
```

In the example, `GetCapMeasurement` returns the measured capacitance value between `"J1-001"` and `"J2-005"` to `fCapacitance`. It checks that the input points are valid, and then displays a message box with the measured capacitance value.

GetRelCapMeasurement (shield, reference)

Use this function to identify that a shield is in a given position. Cable shields are in close proximity to other *many* nets in a cable; hence shields have higher capacitance relative to the other wires in the cable.

This function measures the capacitance between a test point, which you believe is connected to the shield, to all other cable nets. This function also measures the capacitance of another test point, you believe can be used as a reference of a wire in the cable. Finally, this function divides the shield measurement by the reference measurement to return a floating point number representing the ratio. For example, if a shield were five times more capacitive than the reference point, the function will output a floating point number of about 5. The function also returns value that verifies the selected shield and reference points are valid.

The full function definition is as follows:

ratio, valid, = GetRelCapMeasurement (shield, reference)

The value of ratio can be between a floating point number between 0.01 and 100. If one of the points is invalid, the ratio will be zero and ***valid*** will return ***-99***. If both points are valid, ***valid*** will return ***nil***. The inputs ***shield*** and ***reference*** are string inputs representing any system points (such as ***35*** and ***87***), default labels (such as ***J2-003*** and ***J3-22***), or custom point labels (such as ***Shield*** and ***Brown wire***).

Example

```
sShieldPoint = "J1-026"
sReferencePoint = "J1-005"
fRatio, myResult = GetRelCapMeasurement(sShieldPoint, sReferencePoint)
```

The function above will find the ratio of capacitance of a shield with respect to the reference point, J1-005. The shield point is a point tied to the shield. The reference point is used as a point within the shield.

GetResistanceMeasurement (point1, point2)

This function returns the measured resistance value between two points. The returned resistance measurement is returned as a floating point number. The input ***point1*** and ***point2*** must be strings representing any system point (such as ***"35"*** or ***"87"***), a default label (such as ***"J2-003"***, or ***"J3-22"***), or a custom point label (such as ***"Brown wire"*** or ***"Shield"***).

The full function definition is:

resistance, valid = GetResistanceMeasurement (point1, point2)

If one or both of the input points is invalid, the function returns 0 for the resistance measurement and returns a -99 for ***valid***. If both points are valid, the function returns ***nil*** for ***valid***. It is good practice to check the ***valid*** is ***nil*** before using the resistance measurement in further calculations.

Example1

```
myPoint1 = "J1-001"
myPoint2 = "J2-005"
fMeasuredValue, myResult = GetResistanceMeasurement(myPoint1, myPoint2)
```

The function will return the measured resistance value between "J1-001" and "J2-005" in `fMeasuredValue`. The result, `myResult`, will be `nil` since there is no error.

Example2

```
local fStrapRes = GetResistanceMeasurement(VPlusStrap, ConnectingStrap)
if fStrapRes > 1.0 then
     print("AD5P-68A Adapter Not Found")
end
```

The function gets the measured resistance value to check if the adapter strapping is correct.

See the function GetResistanceMeasurement4W to get measured resistance values between two four-wire points.

GetResistanceMeasurement4W (point1, point2)

This function returns the measured resistance value between two 4-wire test points. Note that each of the 4-wire points should be designated as a 4-wire point in the wire list, and be physically connected to a 4-wire force or 4-wire sense mate. Note also, the tester also has requirements as to which test points can be used as 4-wire force and 4-wire sense points.

The returned resistance measurement is returned as a floating point number. The input points `point1` and `point2` may be either a force or sense points, but each must have its four wire mate. The input points must be strings representing any system point (such as "35" or "87"), a default label (such as "J2-003", or "J3-22"), or a custom point label (such as "Force1" or "Force3").

The full function definition is:

resistance, valid = GetResistanceMeasurement4W (point1, point2)

If one of the input points is not a valid 4-wire point, the function returns 0 to ***resistance*** and returns a ***-99*** for ***valid***. If both points are valid, the function returns ***nil*** for ***valid***. It is good practice to check that ***valid*** is ***nil*** before using the resistance measurement in further calculations.

Example

```
my4WPt1 = "J1-003"
my4WPt2 = "J1-005"
fMeasuredValue, myResult = GetResistanceMeasurement4W(my4WPt1, my4WPt2)
if myResult == nil then
     MessageBox("4W Resistance Value is:"..fMeasuredValue)
end
```

The function will return the measured four-wire resistance value between "J1-003" and "J1-005" in `fMeasuredValue`. `myResult` will be nil since there is no error.

Frequently you may use the GetPtType and Get4WPair functions in conjunction with this function. See the function `GetResistanceMeasurement` to do a standard resistance measurement between two test points.

GetTotalCapMeasurement (pointlist)

The full format of this function is:
capacitance, error = GetTotalCapMeasurement (pointlist)

This function returns the total capacitance between one or more points specified in ***pointlist*** and all other points of the device under test. The point list is a string with multiple points separated by a space. Points can be integer values representing system points, or strings representing point labels. Depending on whether the points are valid, the function returns ***capacitance*** and ***error*** as shown in the table below.

	total capacitance	***error***
If the input points are valid	Floating point number representing the capacitance measurement	***Nil***
If the input points are not valid	***0***	***-99***

It is good practice to check that value of ***error*** is ***nil*** before using it in further calculations.

Example:

```
sPoints = "J1-003 J1-005"
fCap, iError = GetTotalCapMeasurement(sPoints)
if iError == nil then
  MessageBox(Format("Point: %s, Cap: %0.1f pF", sPoints, fCap / 1e-12))
else
  MessageBox(Format("Point/s: %s is/are not valid", sPoints))
end
```

If the specified points `J1-003` and `J1-005` are valid, a message box will display the capacitance between these points and all other points in the device under test. If the specified points are invalid, a message box will display the invalid points.

HipotNetTiedToPoint

This function hipots one of the points of a net and returns the leakage resistance value and the test results of the hipotted net. All points of the DUT outside the net are held at ground while the net is hipotted.

The function ***HipotNetTiedToPoints*** can perform the same task as this function, or apply hipot to multiple points one at a time. Inputs and outputs and outputs are identical to both functions, with the exception that function ***HipotNetTiedToPoint*** only allows you to input a single point. Since ***HipotNetTiedToPoints i***s more versatile, we suggest using it rather than this function. See function ***HipotNetTiedToPoints*** for more information***.***

HipotNetTiedToPoints

This function hipots one point or multiple points. Each point may be an isolated point such as an unused connector contact, or one point of a larger net on the tested device. If the point is part of a net, all points in the net will be hipotted. This function can be used to hipot multiple nets simultaneously. While a hipot is applied to the point or points, all other DUT points are held at ground. This ensures the hipotted point or points are sufficiently isolated from other nets. The function returns the leakage resistance and the test results of each hipotted point.

If you want to use this function to apply hipot to only one or to certain nets in a tested device, turn off hipot in the test program. This function will apply hipot even with high voltage testing turned off in the test program. If the optional high voltage parameters for this function are not specified, the function will use the hipot settings used in the test program.

You may choose one of three different formats depending on the high voltage testing complexity and whether the tester has AC voltage capability. The advanced format allows to control all the high voltage settings you can control in the easy-wire software. When using the advanced settings, note that the DW Voltage must be equal to or higher than the IR voltage. Additionally, the IR voltage must be set sufficiently high for the tester to measure some IR resistances. Using the test window to try advanced settings will ensure the settings are valid. The three different formats are as follows:

Simplified Format

```
result, measured = HipotNetTiedToPoint(point [,voltage, insulres, dwell, maxsoak]
```

Advanced DC Format

```
result, measured = HipotNetTiedToPoint(point, [voltagetype, DWVvoltage, DWVcurrent,
                   dwell,IRvoltage, IRinsulres, timegoodfor, soaktime, soakuntilgood])
```

Advanced AC Format

```
result, measured = HipotNetTiedToPoint(point, [voltagetype, frequency, DWVvoltage,
         DWVcurrent, dwell,IRvoltage, IRinsulres, timegoodfor, soaktime, soakuntilgood])
```

Inputs and outputs to these functions are defined in the following tables.

inputs	input description
point	This input is a string representing the test point to be hipotted. The test point can be specified as an adapter test point as in “J1-0017”, or as the test point label such as “RED WIRE”.
voltage	This input is a floating point number representing the DC high voltage applied during both the Dielectric Withstand (DW) test and Insulation Resistances (IR) test.

Inputs continued

inputs	input description
voltagetype	This input is a string representing the voltage type, either AC or DC, applied during the DWV test. The string value is either “AC” or “DC”.
frequency	If AC is specified for voltagetype, use ***frequency*** to specify the AC frequency in terms of hertz. This input can be specified as 25, 30, 50 or 60. If DC is specified for voltage type, no ***frequency*** input is used.
DWvoltage	This input is a floating point number representing voltage applied during the Dielectric Withstand (DW) Test. ***DWvoltage*** can be specified as any within the high voltage range of the tester. The tester will only allow a DW voltage that is equal to or higher than the IR voltage.
DWcurrent	This input is a floating point number that represents the maximum current in terms of mA that can flow into the hipotted net during the Dielectric Withstand (DW) test.
dwell	If used with a DC Dielectric Withstand (DW) Voltage, this input is a floating point number (0.01 to 120) that represents the time in seconds that high voltage is applied to each net during the DWV test. If used with a AC Dielectric Withstand (DW) Voltage, this input is a integer representing the number of AC cycles that are applied to each net during the DWV test. The range is 1 to 7200 with a frequency of 60 Hertz is used. However, the range varies depending on the selected frequency.
IRvoltage	This input is a floating point number that represents the voltage that is applied during the Insulation Resistance (IR) Test. The range for this number is determined by the high voltage capability of the tester.
IRinsulRes	This input is a floating point number that represents the minimum resistance in terms of ohms allowed between unintended connections. The range for this value is 5,000,000 to 1,000,000,000.
timegoodfor	This input is a floating point number that represents the duration of the Insulation Resistance (IR) Test in terms of seconds. The range for this input is .002 to 120.
soaktime	This input Is a floating point number that represents the time in terms of seconds high voltage is applied to stabilize the measurement before doing the Insulation Resistance (IR) Test. The range for this input is .002 to 120. If soak until good is turned on, the actual soak time may be less.
soakuntilgood	This input is set to either 0 or 1. If set to 0, the Soak Until Good setting is off meaning the IR Voltage will be applied for the entire soak time. If set to 1, Soak Until Good is on, meaning once the cable has reached a passing insulation resistance (IR) test threshold, the soak will terminate and the IR test time will start.

Output	Output Description
result	**0** = Passed Test **1** = Has leakage **2** = Overcurrent **3** = Dielectric Failure **99** = Other Failure **100** = Invalid Test Point **110** = * Invalid Frequency **111** = * Invalid DWV Voltage **112** = * Invalid DWV Max. Current **113** = * Invalid DWV AC Duration **115** = * Invalid IR Voltage **116** = * Invalid IR Insulation Resistance **117** = * Invalid IR Good For **118** = * Invalid Soak **119** = * Invalid Soak Until Good * = If these errors occur, the hipot test will not be performed.
measured	Floating point number representing the measured leakage resistance OR 0 for some RESULT1 errors (2, 3, 99)

Examples:

```
iResult = HipotNetTiedToPoints("J1-001")
```

This example hipots the net tied to the point, J1-001, using the high voltage parameters in the test program. The variable `iResult` will contain the test result.

```
iResult, fMeasResis = HipotNetTiedToPoints("Blue", 50, 5000000,30)
```

This example uses the simplified format to hipot the net tied to the custom test point label, Blue. It uses the inputs which are independent of the settings in the test program. High voltage Is set 50 volts, insulation resistance to 5M ohms, and duration to 30 seconds. No soak time was given so it will be set to zero. The variable `iResult` will contain the test result and `fMeasResis` will contain the measured leakage resistance.

```
sPoints = "J1-001 J1-002 J1-003"
fVoltage = 50.0
fHipotInsRes = 10000000
fDuration = 0.100
fMaxsoak = 0.010

local iDNum = DialogOpen("Warning: High Voltage")
local iErr, fRes =
HipotNetsTiedToPoints(sPoints, fVoltage,
fHipotInsRes, fDuration, fMaxsoak);
DialogClose(iDNum)

local sErr
if iErr == 1 then
       sErr = format("Has leakage (%i M ohms)", fRes/1000000)
elseif iErr == 2 then
       sErr = "Overcurrent"
elseif iErr == 3 then
       sErr = "Dielectric Failure"
elseif iErr ~= 0 then
       sErr = format("Unexpected error #%i", iErr)
end

if iErr ~= 0 then
       MessageBox(sErr)
end
```

This example hipots the points using optional parameters, which are independent of high voltage test settings in the test program. A high voltage warning message will display during the hipot. If an error occurs, a message box will display the error.

HipotPointMask

When high voltage testing is turned on, high voltage is applied to each of the nets defined in the test program. This function allows you to mask a point that would normally be high voltage tested in the test program. If you use this function to mask one point of a net, no other point of the net will have high voltage applied to it during the high voltage test.

While you can use this function to subtract points from the tester's list of points that will be high voltage tested, you can also use this function to add points back to this list. Note however, that un-checking the setting **Hipot All Adapter Pins (Not just Connection)**, means that unconnected test points in the DUT will not be in the tester's list of points to be tested. You cannot use this function to mask these points on.

The full format for this function is:

hipotpoints, invalidpoints = HipotPointMask (select [,testpoints])

The optional input ***testpoints*** is a string of the point or points that can be added to hipot point list when the input ***select*** is ***1***. On the other hand, ***testpoints*** will be subtracted from the hipot list when ***select*** is ***0.*** The input ***select*** is an integer value that directs the function as follows:

If *select* equals,	**the function will:**
0	Subtract point or points from the hipot list
1	Add point or points to hipot list
2	Subtract all points from hipot list
3	Add all points to hipot list

Note the function can return ***hipotpoints,*** a string value of the points to be high voltage tested. Additionally the function can return ***invalidpoints,*** a string of the points from the hipot list that are invalid because they are not defined for the test. Test points can be specified by their connector points, or by their test point label.

Examples:

```
sNewHipotPoint = HipotPointMask (0, J1-013)
```

Point J1-13 is removed from the tester's existing list of points to be high voltage tested. The points that will be tested are returned to the string variable `sNewHipotPoint`.

LearnCable

Use this function to learn a cable that is connected to the tester. The cable is learned as a child wirelist and saved to the tester's embedded memory.

The full format for this function is:

```
result, error = LearnCable([1, filename,][2, description,][3, testparameters,]
                           [4, learncomponents,][8, noconnectlearn])
```

The function can return two outputs. The output ***result,*** is a text string containing messages. The output ***error*** is a text string containing errors.

To control learn settings, the following optional inputs may be used to the function:

1, filename	String ***filename*** specifies a test wire list file for the learned cable. This name cannot be the same as the wirelist currently in memory.
2, description	String ***description*** specifies a description for the cable.
3, testparameters	String ***testparameters*** specifies the low and high voltage parameters used for the test.
4, learncomponents	Integer ***learncomponents*** specifies the components types learned in the test. 1 = resistor, 2 = capacitor, 4 = diode, 8 = twisted pair, 11 = all but diodes, 15 = all
8, noconnectlearn	If ***noconnectionlearn*** is the integer 0, the learn will not occur if no connections are found. If greater than 0, the learn will happen even if no connections are learned.

If no optional inputs are given, the function will learn a cable with the following test settings:

filename = untitled.wir
cable description = last learned
connection resistance = 10 Ω
lv insulation resistance = 100k Ω
hipot = off
components to learn = no
Learn with event or custom component scripts = no
learn if no connections = yes

The functions CopyPcFileToEmbedded and CopyEmbeddedFileToPc are often used in conjunction with the LearnCable function.

Examples:

```
sResultMessage, sErrorText = LearnCable (1,"batch1.wire", 4, 1)
```

This example will learn the cable and save the wirelist as batch1.wir. The cable will be

learned with the default low voltage hand high voltage parameters. It will also learn resistors.

```
sResultMessage, sErrorText = LearnCable(3,"connection resis 5
ohm\nlv insulation resis 10k ohm\nhipot voltage 1000V\ninsulation
resis 500 M ohm\nhipot duration 0.100 sec\napply hipot to all
adapter pins")
```

In this example, the cable will be learned with the supplied low and high voltage settings. The default wire list file name and cable description will be used.

MicroLan

The full format for this function is:

result = MicroLan(input, [param])

Use the MicroLan function to talk to Dallas Memory tokens using the MicroLan protocol. See EEPROM to talk to an I^2C like a 24LC00. **Note:** 1100 testers do not support this function.

param	**param description**	*result*
Nothing	Function description	
1, DataPt, GroundPt	Setup	
2	Cleanup – Disconnects from device.	
3	Read a byte from the memory token using MicroLan protocol.	Returns byte read
4, Data	Write a byte to the memory token.	
5	iPresent.	1(yes) or 0(no)
6, NumChars	DS1993 specific command: Read a text string from memory. Skips over ROM bytes.	
7, Data, NumChars	DS1993 specific command: Write a text string to memory.	
8, SpecPin= 1(J1),2(J3),etc	Prepare to communicate to special pin.	Only used internally by Cirris.

Examples:

The following example sets up the Dallas DS1993 (or DS1994) chip. It then checks to see if it is present. If it is present, it writes the text "Hello There Fred" and reads it back out.

```
MicroLan(1, dataPt, gndPt)  -- setup
isPresent = MicroLan(5)     -- is present
if isPresent == 1 then
outtextxy(2,20,"Is attached  ")
MicroLan(7, "Hello There Fred", 17)      -- write text
local theText = MicroLan(6,80)           -- read text
outtextxy(2,21,theText)
else
outtextxy(2,20,"Not attached         ")
end
MicroLan(2)                 -- cleanup
```

The following example verifies the chip is attached to the tester, reads the product code identifying the part type and the first byte of the product serial number. See the Dallas Semiconductor product documentation for the bytes to send to your components to get the data you want out.

```
MicroLan(1, dataPt, gndPt)  -- setup
isPresent = MicroLan(5)     -- is present
if isPresent == 1 then
outtextxy(2,18,"Is attached  ")

MicroLan(4, 51)  -- write data byte and read ROM command(33h)
theProductCode = MicroLan(3)       -- read data byte.
theFirstSerNumByte = MicroLan(3)   -- read data byte.
else
outtextxy(1,18," Not attached    ")
end
MicroLan(2)                         -- cleanup
```

TestWirelist

The full format for this function is:

result = TestWirelist (testnum)

This function performs one of the three tests on the wirelist of the loaded test program. The input, ***testnum,*** is an integer number that identifies which of the three tests to perform as shown in the table below.

testnum	test
3	LV test only
4	HV test only
255	LV, Components, & HV tests

The function returns ***result***, an integer that identifies the results of the test as shown in the table below.

result	test
0	Test Passed
3	LV tested failed (includes components test)
4	HV test failed

If the test fails, you can also use the function TWLGetErrorText to return the error text that relates to the value of ***result***.

Use the function UseChildWirelist to load the child wire list before calling TestWirelist to test. When TestWirelist is used to perform a high voltage test, the high voltage test is performed on all the nets. If you want to hipot one or selected nets, use the functions HipotNetTiedToPoint or HipotNetTiedToPoints. The TestWirelist function does not affect SPC Data Collection or the cable test counters.

Examples:

```
local iTestVal = TestWirelist(3)
```

The function will perform the low voltage and components test on the current wirelist.

```
local iTestVal = TestWirelist(255)
```

The function will perform all tests (low voltage, components, high voltage) on the current child wirelist.

See also the example in the function, UseChildWirelist.

UseChildWirelist

result = UseChildWirelist(wirelist)

This function has two formats. In the format shown above the UseChildWirelist function loads a wire list from memory or disk. The function TestWirelist can then be called to test the wire list. The input ***wirelist*** is a string that defines an optional path and the wire list name. The function returns 0 to ***result*** to indicate the wire list loaded successfully, or a nonzero integer if the wire list failed to load.

UseChildWirelist(stop)

In this format, the function will stop using the wire list, and depending on the integer value of stop, either leave the wirelist in memory or free the wire list from memory as shown below.

stop	Description
0	Stop using the current child wire list but leave it in memory.
1	Stop using the current child wire list and free the memory.

This function allows you to load multiple child wire lists into memory where they can be quickly recalled and used at one at a time. After finishing using all the wire wirelists, remove all of the wire lists from memory by calling this function again.

Example 1

```
iCouldntLoad = UseChildWirelist(sChildWLFilename)
```

This example will load the named wire list contained in the string, `myChildWLFilename` and make it the current wire list running on the tester.

Example 2

```
TestArray = {"test1.wir", "test2.wir", "test3.wir" }
iNumTests = 3
iTesting = 1
while iTesting == 1 do
iButtonNum = MessageBox("Test all", "Yes", "Done")
if iButtonNum == 1 then
iTestIndex = 0
while iTestIndex < iNumTests do
MessageBox(format("Ready to test # %i", iTestIndex+1))
-- display the test number so user can get ready
iLoadFailed = UseChildWirelist(TestArray[iTestIndex])
-- make it current, loading wirelist from disk or memory as needed
if iLoadFailed ~= 0 then
MessageBox("Couldn't load wirelist")
else
iTestResult = TestWirelist(255) -- perform all tests
if iTestResult ~= 0 then
sErrors = TWLGetErrorText(iTestResult)
MessageBox(sErrors) -- Show them what failed
end
UseChildWirelist(0)--Leave in memory, to get quickly next time
end
end
else
iTesting = 0 -- Done, will cause tester to exit loop
end
end
       iTestIndex = 0
       while iTestIndex < iNumTests do
UseChildWirelist(TestArray[iTestIndex]) -- make it current
UseChildWirelist(1) -- free it from memory
end
```

This example will loop through the child wirelists: `test1.wir, test2.wir` and `test3.wir.` Test1.wir will be loaded and if no errors occur during loading, testing will be done `using test1.wir.` After testing, the child wirelist will be left in memory for quick access in the future. The script will continue for each of the child wirelists. At the end of loading and testing all the child wirelists, each child wirelist will be loaded and freed from memory.

Printer Functions

Notes on Printing

When formatting strings to print, the Cirris print functions in this section allow for either the c-like or decimal escape codes such as those in the following table:

c-like code	decimal code	description
\n	***\10***	new line
\f	***\12***	form feed

Refer to the 5.1 Lua Manual for more information on formatting strings.

Printer Status Codes

Some Cirris print functions return an integer to reflect the printer status. The meaning of these integers is shown table below.

status	description
0	ready
1	printer not selected
2	timeout
3	I/O error
4	out of paper
5	printer busy

Cirris.EndPrintJob()

Ends the print job. Used after Cirris.StartPrintJob and Cirris.Print.

Cirris.GetPrinterNamesByIndex(number)

This function is useful to find the exact name of a local Windows printer or a network printer with its server path. The exact printer name is required for the Cirris.SetPrinter function, which is used to direct printing to a printer other than the default printer.

The Cirris.GetPrinterNamesByIndex function returns the name of a printer as a string. The input ***number*** is an integer corresponding to one of the printers accessible to the test system. The function Cirris.GetNumPrinters can be used to get the number of printers accessible to the test system. For example, if Cirris.GetNumPrinters returned 3, you could input a 1, 2, or 3 into Cirris.GetPrinterNamesByIndex to see the exact names of each of the three printers.

Example:

```
iNumberOfPrinters = Cirris.GetNumPrinters()
printer_names = ""
for i=0, iNumberOfPrinters-1 do
     printer_names = printer_names..i..". "..Cirris.GetPrinterNameByIndex(i).."\n"
end
MessageBox("~Cirris.Print~".."Available Printers:\n"..printer_names.."\n")
```

The example above uses the function GetNumPrinters to find the number of printers, and then uses the function GetPrinterNamesByIndex to make a string of printers that are then displayed using the message box function.

Cirris.GetNumPrinters()

Returns the number of Windows printers that are available to the tester. The number of printers is returned as an integer, which can then be used in conjunction with the Get PrinterNameByIndex to return the printer names.

Cirris.NewPage [(immediate)]

Starts a new print page. Subsequent data to print will be on the new page. If the optional input ***immediate*** is ***1***, the function immediately causes the printer to output the current page. Any data previously sent to printer will print on this page. If ***immediate*** is not used or is ***0***, the printer waits until Cirris.EndPrintJob is called to print the page.

Cirris.Print (string)

Outputs a string to the print spool. The function Cirris.EndPrintJob will cause the string to be printed. The string can include the ascii text to be printed, and escape codes such as \n or \f to be applied to the string.

The full function definition is shown below. Note that three outputs can be returned.

number, status, error = Cirris.Print (string)

The output ***number*** identifies the number of characters sent to the print spool; ***status*** is and integer that identifies the printer state (see Printer Status Codes on page 74); ***error*** is a string that describes a print error.

Cirris.SetPrinter (name)

Selects the printer that will be used to print the subsequent print job. The input ***name*** is a string that must exactly match a local Windows printer name or network printer name with its server path. Use the function Cirris.GetPrinterNamesByIndex to get the exact system names of printers.

Note the function can return two outputs as shown below.

status, error = Cirris.SetPrinter (name)

The output ***status*** is an integer that identifies the printer state (see Printer Status Codes on page 74); ***error*** is a string that describes a print error.

Cirris.StartPrintJob()

Tells the print driver to start spooling the print job. This function will direct printing to the Windows default printer, or to the printer selected by the function Cirris.SetPrinter. The Cirris.StartPrintJob function is always followed by Cirris.Print and Cirris.EndPrintJob functions.

The function has no inputs, but can return two outputs as shown below.

status, error = Cirris.StartPrintJob()

The output *status* is an integer that identifies the printer state (see Printer Status Codes on page 74); *error* is a string that describes a print error.

SendTextToParallelPrinter(string [,return])

This function has been preserved so that scripts previously written for the Touch 1 and 1100 testers can be used with Easy Touch Scripting. When run in the Touch 1 and 1100 testers, the function would send a string to a parallel printer. However now when used with Easy Touch scripting, this function sends the string to the Windows default printer's print spool, not the parallel port on the tester. The string can include the ascii text to be printed, and escape codes such as \n or \f to be applied to the string.

There are a few notable differences when using this command with a modern printer as opposed to a parallel printer. When sending a string to a parallel printer, the printer prints immediately. On a modern printer no text is printed until a form feed (\f) is received. Also, parallel printers accepted the escape code \r (return to beginning of line) to overprint text at the beginning of a line. However, modern printers do not support this feature and will replace, not overprint, existing text when a \r is used. Additionally some parallel printers do not return to beginning of a line when a new line code (\n) is received. Windows printers return to the beginning of a line when a new line code is received.

This function allows an input ***return***. If ***return*** is non zero, a carriage return (***\r***) is always added with a new line code (***\n***). This setting was used to ensure a parallel printer would return to the beginning of a line when it received a new line code (***\n***).

The function can return two outputs as shown:

number, status = SendTextToParallelPrinter(string, return)

The output ***number***, is an integer representing the number of the characters that were printed. The function also returns an integer to ***status***. This integer identifies the printer's status as defined in printer status code table on page 74.

Examples:

```
sTextToPrint = "Text string to be printed"
iNumCharsPrinted, iStatus = SendTextToParallelPrinter(sTextToPrint)
```

The function will output "Text string to be printed" to the default printer. iNumCharsPrinted will contain the number of characters printed and iStatus will contain a number describing the current status of the printer.

```
SendTextToParallelPrinter("GOOD CABLE\f")
```

The function will cause "GOOD CABLE" to be printed on the default printer.

Tester Information Functions

Get4WPairPt (point)

This function determines if a given point is a four-wire pair point. If the point is a four-wire point, the function returns a string containing the label for the point. If not, or if the function is unsuccessful, the function returns nil. The point can be specified as an integer value representing the system point, or a string representing a point label.

Example:

```
iPinNumber = 3
testPoint = Get4WPairPt(iPinNumber)
if testPoint == nil then
    error(iPinNumber .. " is not a part of a 4W pair)
end
```

The RESULT, `testPoint`, will contain the label text if it is a fourwire pair point or nil if it is not. If it is not a four-wire pair point, a message box listing the point will be displayed.

GetHardWareVersion()

This function returns the hardware version running on the tester. There are no inputs to this function. The version is returned as a string.

GetProbedPin ( )

Use this function to get the number of the test point of the currently probed pin.

GetRawPointNum (label)

This function returns the system pin number of a custom label. System pin numbers can range be 1 and 1024 depending on the size of your test system. If the label string is invalid, the function returns nil. If no input is supplied, this function returns a description of itself.

Examples:

```
sLabelText = "GROUND"
iPinNumber = GetRawPointNum(sLabelText)
```

The function will return the pin number in `iPinNumber` for the GROUND custom label.

GetPtType (point)

This function returns a value to identify whether a given point is Stress High or Stress Low. Half of the test points on the tester can be used to supply current from the high current source in the tester. Stress High is used to supply the current and Stress Low is used to sink the current. This function is used ONLY for commands dealing with four-wire resistance measurement and the high current source.

returned point type	***point***
Integer containing one of the following codes: **1** = Stress High (can source high current) **2** = Stress Low (can sink high current)	An integer containing the pin number such as 82 OR A string containing the point label such as “J3-008”

GetTimeAsText (select)

This function returns current tester time information as a text string. The returned time information is determined by the integer input used as shown below. See also the related function GetTimeasInteger.

Returned Time Information	***select***
Current tester **hour** 0 - 23	***1***
Current tester **minutes** 0 - 59	***2***
Current **seconds** 0 - 59	***3***
Current time in the format HH:MM:SS	***4***
Number of **“clock ticks”** since the tester was turned on. See the definition of clock ticks in GetTimeAsInteger function.	***5***

GetTimeAsInteger (input)

This function returns current tester time information as an integer. The type of returned time information is determined by the input number as shown below. See also the related function GetTimeasText.

Returned Tester Time Information	***input***
Current tester **hour** 0 - 23	***1***
Current tester **minutes** 0 - 59	***2***
Current tester **seconds** 0 - 59	***3***
Number of **"clock ticks"** since the tester was turned on. There are 1000 clock ticks per second. Clock ticks begin incrementing from 0 when you turn on the tester, increment up to 2147483, the begin at -2147483 and again increment to 2147483 again and so forth.	***4***

GetSystemInfoAsText (select)

This function returns system information about the tester as a string. The system information that is returned is determined by the integer value of ***select***, as shown in the table below.

returned tester information	***select***
Tester serial number (8 characters)	***1***
Login name that was used to log into the tester	***2***
Software version of the tester	***3***
Hardware version of the tester	***4***
Number of system test points	***5***
Maximum scanner voltage of the tester	***6***
Tester type	***7***

Test Information Functions

Cirris.BadCount

Returns the total of assemblies tested bad in all test runs.

Cirris.CableID

Returns the cable ID for the cable being tested. For cable ID's to be recorded, a method for serial numbering must be selected under the Test Defaults Tab in the Test Editor.

Cirris.GetAdapters(mode, [index])

This function returns information on the adapters involved with a test. There are three modes in which this function can be called as detailed below:

Mode	Mode Description	Number of Return Values	Return Value Descriptions
0	Learn – Performs a "Learn" operation and discovers all adapters attached to the tester and then returns the number found (only used in EVT script, event 1)	1	Number of Learned adapters
1	Get Index – Returns the position, signature, and strapping of the adapter at the index specified by the second parameter to the function call (first adapter is at index 0)	3	Position, Signature, Strapping
2	Get Count – Returns the number of attached adapters as determined by the loading of the wirelist (i.e. only adapters specified in the wirelist will be considered)	1	Number of verified adapters

The Learn mode can only be called in an event script triggering upon event 1. This is because the "Learn" operation performed by GetAdapters will erase test state data and will cause the error "Not Ready to Test" when the test continues to execute after the Lua call. Calling GetAdapters during event 1 is allowed because it occurs before the test is initialized.

To get adapter information from a component script or other event number, use the Get Count mode to get the number of adapters available.

After calling GetAdapters in either Learn or Get Count mode, the function can be called

in Get Index mode with a second parameter (index) in order to get information on a specific adapter. The index provided must be in the range of 0 to (AdapterCount – 1) where AdapterCount is the value returned from GetAdapters in Learn or Get Count mode.

Example:

```
local sOutput = "Adapter Information:\n"
local iCount = GetAdapters(2)

for iIndex = 0, iCount-1 do
	local Position, Signature, Strapping = GetAdapters(1, iIndex)
	if Position == nil then
		sOutput = sOutput.. "Invalid index: "..tostring(iIndex)
	else
		sOutput = sOutput.. tostring(iIndex).."\n"
		sOutput = sOutput.. "Pos: "..Position.."\n"
		sOutput = sOutput.. "Sig: "..Signature.."\n"
		sOutput = sOutput.. "Str: "..Strapping.."\n\n"
	end
end
MessageBox(sOutput)
```

The example above could be called in a component script to print out all the information for adapters used in the test.

Cirris.GoodCount

Returns the number of cables that were tested good in the current test run.

Cirris.LotID

Returns the Lot ID for the current test run. For Lot ID to be recorded, the Enter Lot ID option must be checked under the Set Test Default**s** Tab of the Test Editor.

Cirris.RunBadCount

Returns the number of cables tested bad in the current test run. A test run begins when a test program is loaded and the first test completes, and ends when the test program is closed.

Cirris.RunGoodCount

Returns the number of cables tested good, in the current test run. A test run begins when a test program is loaded and a first test completes, and ends when the test program is closed.

Cirris.RunTotalCount

Returns the number of all cables tested, good or bad, in the current test run. A test run begins when a test program is loaded and a first test completes. A test run ends when the test program is closed.

Cirris.StationID

Returns the Station ID number. This is a unique number that that identifies the tester or test station. The Station ID number is generated when the Cirris easy-wire software is installed.

Cirris.TotalCount

Returns the total count of all cables tested in all test runs. The total count continues to accumulate for any test that completes until the total counts are cleared in the test window.

GetCableStatus ()

This function returns the good or bad status for the cable tested. The function has no inputs. If the cable status is good 0 is returned; if bad, -99 is returned.

Example

```
if(GetCableStatus() ~= 0) then
iNumBadCables++
end
```

The integer, `iNumBadCables`, will be incremented if the cable test result is bad.

GetComponentCount ()

This function returns an integer value of the number of components in the loaded Test Program. Components include custom components, resistors, diodes, capacitors and so forth. The function has no inputs.

Example:

```
iCount = GetComponentCount()
MessageBox("Number of components = "..tostring(iCount))
```

This example will display a message on the tester's screen that contains the number of components in the current wirelist.

GetErrorSignature ( )

This function returns the error signature for a bad cable as a string. If a cable is good, this function returns the cable signature as a string. There are no inputs to this function.

Example

```
if GetCableStatus() == -99 then
sErrorSig = GetErrorSignature()
MessageBox(sErrorSig)
end
```

This example will output the error signature to the screen if the cable tested was bad. For example, "8062B5-6F8NO" would be displayed on the screen.

GetComponentDetails (index, select)

This function returns test details about components in the loaded test program. The full function format is:

error, detail = GetComponentDetails (index, select)

The table below relates the function outputs and inputs.

error	***detail***	***index***	***select***
0 = No error, detail is valid. -9999 = Error, detail is *not* valid (integer)	Component Type 1 = Resistor 2 = Diode 3 = Link 4 = Capacitor 5 = Relative Capacitance 6 = Twisted Pair 7 = 4 Wire Resistor 8 = 4 Wire Wire 9 = Wire	Position of the component in the wirelist. (integer)	0 = Component Type
	Component's Pass/Fail Status 0 = Fail 1 = Pass		1 = Pass Fail Status
	From Test Point Represented as a zero based system test point. (integer)		2 = From Test Point
	To Test Point Represented as a zero based system test point. (integer)		3 = To Test Point
	Nominal or expected value applied to the component setup. (floating point)		4 = Expected Value
	Tolerance applied to the component setup. (integer)		5 = Tolerance
	Measured values from most recent test. (floating point)		6 = Measured Value 1
	Reverse voltage from diode component returned from most recent test. (floating point)		7 = Measured Value 2

GetErrorText ( )

This function returns error text on cables that have tested as bad on the tester. This error text is the same error text displayed on the error screens during cable testing. If the cable is not attached, the errors returned for this function are the same as the errors for SPC data collection. If the cable is still attached, the errors returned for this function are intermittent or low voltage errors. There are no inputs to this function. If the cable tests as good, the function returns an empty string, "".

GetNumberTested (input)

This function returns the counts of the cables tested (Total, Good, and Bad) in the current test run. These are the same counts displayed in the Test Window on the tester. Each input number will return a different count type in the table below. If the function has no input, it will return a description of itself.

returned total (integer)	***input***
total tested	***1***
tested good	***2***
tested bad	***3***

GetPinLabel (pin,[select])

This function returns a default or custom label for a raw system pin number. Pin numbers (test points) are between 1 and 128 for a base tester with no addons, between 1 and 256 for a tester with one addon, and so forth. If the function has no input, it returns an error describing the function.

returned label (string)	***pin* (integer)**	***select* (integer)**
Default Label assigned to the raw system test point. For example, “J2-024”.	A raw test point number. such as 56	Any integer greater equal to or greater than one.
Custom Label assigned to the raw system test point For example, “Red Wire”	A raw test point number. For example, 56	Not used

Example

sCustomLabelText = GetPinLabel(25)

The function will return the custom label text assigned to pin number 25 as a string in the variable `sCustomLabelText.`

iPinNumber = 5

iReturnDefaultLabel = 1

sDefaultLabel = GetPinLabel(iPinNumber, iReturnDefaultLabel)

The function will return the default label format for pin number 5 because `iReturnDefaultLabel` is set to one. It returns a string such as J1-005 in `sDefaultLabel.`

GetWirelistInfoAsText (input)

Use this function to get text information on the loaded test program. Each input number will return different information about the test program.

input (intetger)	Returned Test Program Information
1	The loaded test program
2	The current location or path of the easy-wire database test program without the filename (up to 144 characters)
3	The description of the test program. (Up to 30 characters
4	The cable signature
5	The cable signature (complex cables only)
6	The insulation test parameter signature (complex cables only)
7	The adapter position (Jx), a space, and then the adapter signature (six characters)
8	The adapter position and signature followed by a ***** separator followed by the adapter description (up to 30 characters)
10	The Low Voltage Settings with each parameter on a separate line
11	The High Voltage Settings with each parameter on a separate line
12	Each net with its connections on a separate line. These connections will not be labeled even if custom labels are used in the wirelist.
13	Each component on a separate line
14	Each custom or default test point label on a separate line
15	Each fourwire pair on a separate line
16	The CRC Signature
17	The name & location of the Custom Component Script
18	The name & location of the Test Event Script
19	SPC Data Collection Settings

Examples:

```
sWirelistText = GetWirelistInfoAsText(1)
```

The function returns the current wirelist filename as a text string in, `sWirelistText`. For example, `sWirelistText = "TESTFILE.WIR"`.

`sCableDescription = GetWirelistInfoAsText(3)`
The function returns the cable description of the current wirelist as a text string in `sCableDescription`. For example, `sCableDescription = "Cable for batch 10"`.

`sCableSignature = GetWirelistInfoAsText(5)`
The function returns the cable signature of the current wirelist as a text string in `sCableSignature`. For example, `sCableSignature = "5760A5-6F0Z2"`.

`sLowVoltSettings = GetWirelistInfoAsText(10)`
The function returns the low voltage settings as text with each parameter on a separate line in `sLowVoltSettings`. For example, `sLowVoltSettings =`

```
"CONNECTION RESIS 2.00 K ohm
 LV INSULATION RESIS 6.00 K ohm"
```

`sWirelistLabels = GetWirelistInfoAsText(14)`
The function returns each label as a text string on a separate line in `sWirelistLabels`. For example, `sWirelistLabels =`

```
"J1-001 = BLUE
 J1-002 = RED
 J5-006 = GREEN"
```

`functionDescription = GetWirelistInfoAsText()`
The function returns the function description as a text string in `functionDescription`.

IsSPCDataCollectionOn()

This function returns a 1 if data collection is on in the test program, or a 0 if data collection is off. There is no input for this function. Note that that data collection is a purchased option on the Easy Touch and the 1100 testers. Additionally, in the tester's software interface you must select **Stored Measured Test Values** for a test program to use this feature.

Example:

```
iDataCollectionOn = IsSPCDataCollectionOn()
```

`iDataCollectionOn` will contain a zero if SPC data collection is OFF in the current wirelist or a one if SPC data collection is ON in the loaded test program.

TWLGetErrorText

error = TWLGetErrorText(result)

This function returns a text string, ***error***, that describes an error from the last test after the function TestWirelist was called. The function TestWirelist function returns ***result*** which is used as an input to TWLGetErrorText. Using TWLGetErrorText is the only reliable way to get error text when testing wirelists using TestWirelist.

Example:

```
iTestVal = TestWirelist(255) -- 255 selects LV, component and HV test
if iTestVal > 0 then -- error are greater than 0
      sErrText = TWLGetErrorText(iTestVal)
end
```

This example will get the error text for the failed wirelist test.

WiresAttached()

The WiresAttached function determines whether a cable is present. It checks only the adapters listed in the currently loaded wirelist. This function has no inputs. The function returns ***nil*** if nothing is attached, or a ***1*** one or more connection is sensed.

Example:

```
iAttached = WiresAttached ()
```

`iAttached` will contain `nil` if no wires are attached, or `1` if wires are attached.

User Interface Functions

Cirris.GetWrappedText(text [, length, center])

This function applies the control codes to a text string so the text will be wrapped at a defined maximum line length. Codes may also be applied to center the text. See the table below.

input	description
text	A string containing the text which will be wrapped.
length	Integer indicating the maximum number of characters per line for the wrapped text. If unspecified, the default is 20 characters.
center	0 = Text is not centered 1 = Center text If unspecified, the default is 0.

If the input text contains a line feed code (LF) by itself, this function replaces it with a carriage return and a linefeed code (CRLF).

In Easy Touch Scripting the legacy Cirris function GetWrappedText behaves identically to the Cirris.GetWrappedText function. Note that previously a third input allowed the legacy GetWrappedText function to control whether linefeed codes would be replaced with carriage return line feeds. If used this input now ignored.

Example:

sWrappedText = GetWrappedText(sTextToWrap, 80)

The function returns the text wrapped at 80 characters per line as a string in `sWrappedText`. The text will be left justified.

Cirris.HideBackgroundImage

This function closes the background image file previously displayed using Cirris.ShowBackgroundImage function. The Cirris Test Window and other open application windows will again be visible on the test monitor.

Cirris.PressDoneButton

This function effectively presses the Done button in the test window while in a script. This command allows event script that could be in a mode to run repeatedly to exit.

Cirris.ShowBackgroundImage (image [,s] [,p])

This function displays a background image file to hide the easy-wire test window and other open application windows. The image file can serve as a background in an event script so that operator focus can be on subsequent script messages. This function is most typically used in an event script that takes control of the test. The string ***image*** specifies the image file name and path. The setting ***s*** can be set to ***1*** to stretch the image to the nearest horizontal or vertical monitor restraint. If ***p*** is additionally set to ***0***, the image can be stretched non proportionally to entirely fill the monitor. Image files may include png, jpeg, bmp, and gif file types.

button = DialogCheckBtn (handle)

The DialogCheckBtn function is used in a while loop to identify a dialog box button push. The function returns an integer to ***button*** corresponding to the button that was pressed in an open dialog box. While no button is pushed, this function returns a ***0***. The dialog box is referenced by its handle that was created using the DialogOpen function. The script is still running when a dialog box is displayed on the screen unlike a message box. Because operations are still running, any tester pop-ups will display on top of a dialog box. You can check digital I/O or perform test functions while waiting for a dialog box button to be pressed.

See the example dialog box for the DialogOpen function on the next page. The code below would identify which button is pushed for the example dialog box.

```
iDia=DialogOpen ("~Wire Color~".."What color is the wire?", "Red", "White",
"Green")
iButton = 0
    while iButton == 0 do
    iButton = DialogCheckBtn(iDia)
        if iButton == 1 then
           MessageBox("You pressed Red")
        end
        if iButton == 2 then
        MessageBox("You pressed White")
        end
        if iButton == 3 then
        MessageBox("You pressed Green")
        end
     end
     DialogClose(iDia)
```

DialogClose (handle)

Closes a dialog box identified by the input number. This function is used along with the functions DialogOpen and DialogCheckBtn. The script is still running when a dialog box is displayed on the screen unlike a message box. Because operations are still running, any tester pop-ups will display on top of a dialog box. This function is used to clean up memory. For every DialogOpen function there should be a DialogClose function.

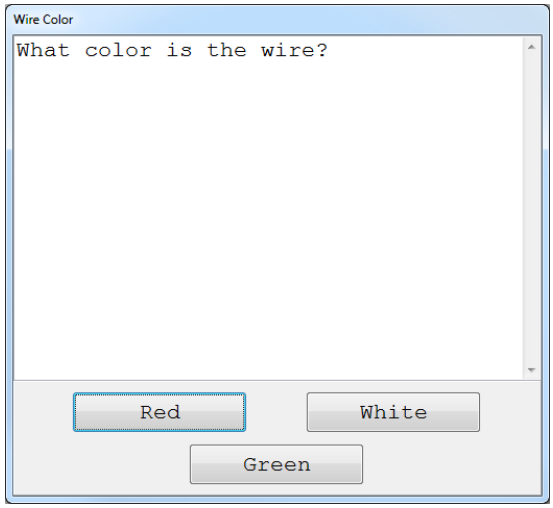
DialogOpen

handle = DialogOpen ([~title~..]message [, button1] [, button2] … [,sbutton6])

The DialogOpen function displays a dialog box containing custom text and up to six dialog buttons. This function also returns a handle to identify the dialog box. This handle is used with the DialogCheckBtn and DialogClose functions. The dialog box will not have a default CANCEL button if no other custom buttons are created. The script is still running when a dialog box is displayed on the screen unlike a message box. Because operations are still running, any pop-ups from the tester will display on top of these dialog boxes.

For example, the following statement creates a dialog box like the one below

```
iDia = DialogOpen ("~Wire Color~.."What color is the wire?", "Red", "White", "Green")
```



MessageBox

This function displays a message box on the tester's display to display a message to the operator. Up to six buttons can be displayed on the message box to allow a selection. Unlike dialog boxes, the tester and script do not continue to run when a message box is displayed. The full format for this function is as follows:

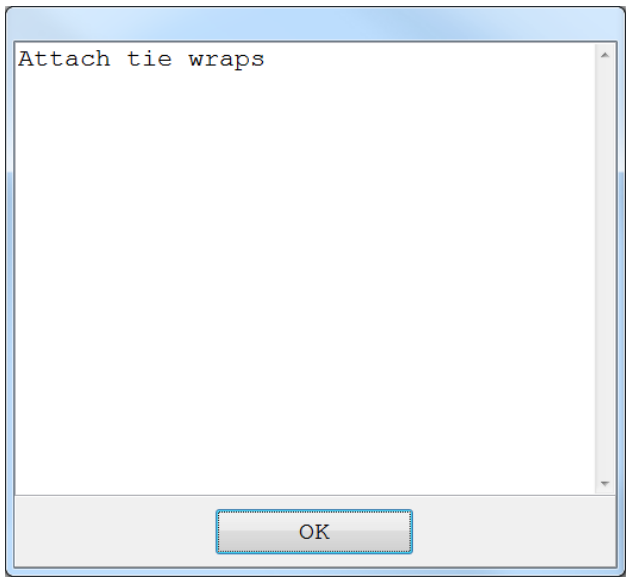
button = MessageBox ([~title~]..message [, button1] [, button2] … [,sbutton6])

The function returns an integer (1 to 6) to ***button*** to identify a pressed button. Strings are also used to define the message box message as well as the optional title and button names. If no buttons are specified, the message box will contain an OK button that the operator will have to select to continue.

Examples:

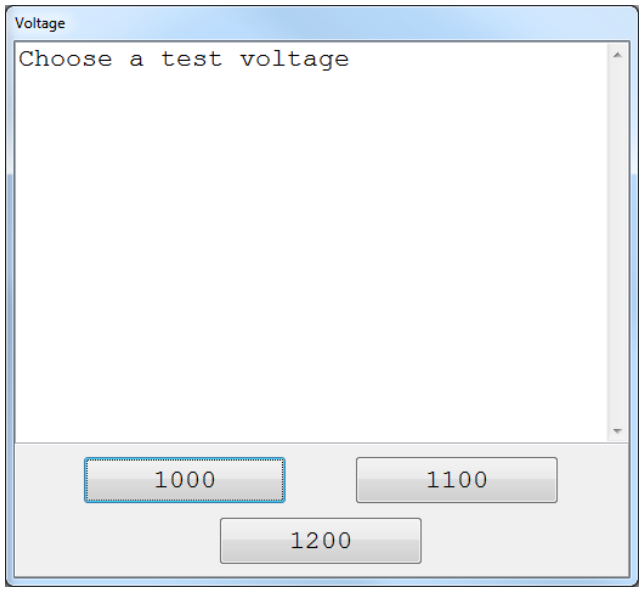
```
MessageBox ("Attach tie wraps")
```

Results in the message box below.




```
button=MessageBox ("~Voltage~.."Choose a test voltage", "1000", "1100", "1200")
```

Results in the message box below:



PlaySound (pitch, duration, volume)

This function plays a sound at a specified pitch, duration and volume. The volume is dependent upon the tester's volume setting. For a maximum sound, set the tester's volume at its maximum setting.

input	description
pitch	An integer indicating the pitch of the sound in Hertz.
duration	An integer indicating the duration of the sound in seconds.
volume	An integer from 1 to 100 indicating the volume of the sound where 100 is the loudest.

Example:

```
iPitch = 300
iDuration = .5
iVolume = 100
PlaySound(iPitch, iDuration, iVolume)
```

This example will play the sound for 500 milliseconds at 300 Hz at full volume.

PromptForUserInformation

This function displays a user prompt window on the tester and returns the valid user entry. The prompt window can receive user entry that is alphanumeric, numeric, or "password". When the password entry is used, alphanumeric entry is accepted, but asterisks hide the operator's entry characters. Numeric entry accepts only integers. At the bottom of every prompt window an OK and CANCEL button. The full format for the function is as follows:

entry = PromptForUserInformation(type, firstline, secondline, maxchar[, initial])

Inputs to this function are described in the table below.

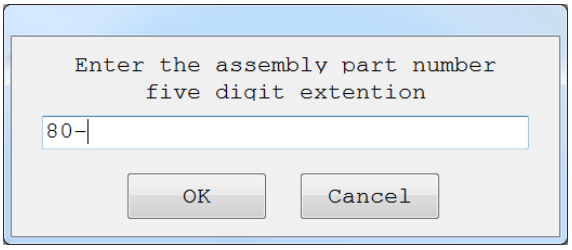
Input	Description
type	An integer value that defines the type of user input. **1** = alphanumeric **2** = numeric (integers) **5** = password
firstline	String of text - up to 30 characters - for the first prompt line.
secondline	String of text - up to 30 characters - for the second prompt line.
maxchar	Integer of the maximum number of characters the user can enter. The range for this value is 1 to 30.
initial	Contains an initial value displayed in the prompt entry box. An alphanumeric string or an integer depending the value of ***type***

The function will return the text the user enters to ***entry***, or if the user presses Cancel, the function returns ***nil*** to ***entry***. Always test for the value of ***nil*** and replace it with a valid string so subsequent routines will not break when they expect a string

Example:

```
sInput1 = PromptForUserInformation(1, "Enter the assembly part number", "five digit extention", 10,
"80-");
```

Results in the message box:



1100 Embedded File Functions

The following scripting functions allow *Easy Touch Scripting*, run on a PC, to control files stored in the embedded memory of the Cirris 1100R and 1100H testers.

Many scripts were written with *Cirris Scripting* functions designed to run internally to Cirris 1100R and 1100H testers. Many of the functions below are the similar equivalent to the Cirris Scripting functions designed for controlling files internally to the tester. To differentiate many of these functions from the previous scripting functions an underscore "_" has been added at the beginning of the function's file name. For example the first function described below, _appendto has the previous similar function, appendto.

Therefore, if you want to use a script written with *Cirris Scripting* functions for internal file control, you will have to do one of two things. You can change the functions used in the script to the new files names, or you may make an argument statement at the beginning of the script file equating the old function file name with the new. For example, `appendto=_appendto` will make the new internal function _appendto take the place of the previous appendto function.

_appendto (x:\\file)

This function opens a file from the 1100 internal memory and returns a handle for the file. If no file is opened, nil is returned to the handle. If the function is unsuccessful, a separate error message is also returned. Use this function when you want to add something to the file. This function will not erase the contents of a file. This function does not close the current output file so when done use the _closefile or _writeto functions to close the file.

Example:

```
serialNumFile, sErrMsg = appendto("c:\\cableser.dat")
write("Serial Number: 12345678")
writeto()  -- This closes the file
```

Returns the file handle in `serialNumFile` if the file "cableser.dat" opens successfully. If it fails opening the file, it will return a nil in `serialNumFile` and an error string in `sErrMsg`. The `write` function will write to the file and the `writeto` function will close the file.

CopyEmbeddedFileToPc (source, x:\\destination)

This function copies a file from internal memory of an 1100 tester to PC controlling the tester. The input ***source*** is the filename on the 1100; ***destination***, the path and filename on the PC. If the function is successful, it returns nil. If unsuccessful, it returns an error string describing the error. See also the related function CopyPcFileToEmbedded.

Example:

```
sCopyErrror = CopyEmbeddedFileToPc (“lastlrnd.sys”, “c:\\myfolder\\last.wir”)
```

In this example the last learned file from an 1100 tester, `lastlrnd.sys` is copied to a new file name `last.wir` in the folder `my folder` on the PC.

_copyfile (source, destination)

This function copies a source file to a destination file within the 1100 internal memory. If there is a copy error, its description is returned.

Example:

```
_copyfile (“test1.wir”, “test1.bak”)
```

CopyPcFileToEmbedded (x:\\source, destination)

This function copies a file from the PC running the easy wire software to the embedded memory of the 1100 tester. The input ***source*** is the path and filename on the PC; ***destination***, the filename in the 1100 memory. If this function is successful, it returns nil. If unsuccessful, it returns an error string describing the error. See also the related function CopyEmbeddedFileToPc.

DirUtils (input1, [input2])

This function can perform directory services for the 1100 embedded memory. These services include changing the current directory, displaying the current directories contents, and getting the current directory's path. The table below shows the relationship between the inputs of this function and returned directory information.

returned information	***input 1***	***input2***
New current directory's path is returned as a string, or nil if unsuccessful.	***1***	String containing the directory path that will be the current directory.
Contents of the current directory of the specified file type returned as a string, or nil if unsuccessful.	***2***	String containing a linefeed delimited list of file types to display. Examples: `"*.wir"` or `"*.*"` or `"*.wir\n*.lua\n*.evt"` or `"*.rpt\n*.cmp"` default: `"*.*"`
Current directory's path returned as a string, or nil	***3***	Not used

Example:

```
sCurDir = DirUtils(3)
sResult = DirUtils(1, sNewDir)
sDirContents = DirUtils(2,"*.wir")
sResult = DirUtils(1, sCurDir)
```

This example stores the original directory path in `sCurDir`. The current directory then becomes the directory contained in the string `sNewDir`. All the wirelist files in the directory and their available descriptions are then retrieved into `sDirContents` before changing back to the original directory.

_openfile (x:\\filename, mode)

This function opens a file in the mode specified in the string ***mode***. It returns a new file handle, or, in case of errors, nil plus a string describing the error. Potential values of ***mode*** are defined as follows:

mode	**description**
"r"	read mode
"w"	write mode
"a"	append mode
"r+";	update mode, all previous data is preserved
“w+”	update mode, all previous data is erased;
"a+"	append update mode, previous data is preserved, writing is only allowed at the end of file

_read ([readpattern])

This function reads a file according to a read pattern. If you call the function without ***readpattern***, it defaults to reading the next line or the first line for the first read. The function returns a string containing the characters read, or a nil if it fails to read anything. Previous to using the read function, the file must be opened using the _readfrom function. Use the function, readfrom, to close the file when finished reading. Potential read patterns are described below.

readptattern	**description**
“*.*”	Returns the next character or nil on end of file.
“.*”	Reads the entire file.
".*"	Returns the next line without the linefeed or nil on end of file. This is the first line when reading a file for the first time. You do not have to type in this default pattern, just call the function without using any input.
"[^\n]*{\n}" (default pattern)	Returns the next word skipping spaces if necessary or nil on end of file.
"{%s*}%S%S*"	Returns the next integer or nil if the next characters are not integers.
"{%s*}[+-]?%d%d"	Append update mode. Previous data is preserved. Writing is only allowed at the end of file.

_readfrom (x:\\filename)

Use this function to open or close a file from 1100 embedded memory. The path should be included in the filename. When used to open a file, the function returns a file handle. If the function fails to open a file, it returns a nil, and a string describing the error. When this function is called without a filename, it closes the file.

Example:

```
sFilename = "c:\\cable550.wir"
canReadFile, sErrMsg = readfrom(sFilename)
if (canReadFile ~= nil) and (sErrMsg == nil) then
   local sFirstLine = read
   local sSecondLine= read
   readfrom ()
end
```

This example combines the read function with the `readfrom` function. It will return the file handle in `canReadFile` if it successfully opens the `cable550.wir` file. If the function fails, it will return a nil in `canReadFile` and an error string in `sErrMsg`. If the file opens, the first line will be read into the string `sFirstLine` using the read function. The second line will be read into the string, `sSecondLine` using the read function. The file is then closed calling the function `readfrom` again.

_remove (x:\\filename)

This function deletes the file with the given name in 1100 embedded memory. If this function fails, it returns nil, plus a string describing the error.

Example

```
sFilename = "temp.wir"
error, sErrMsg = remove(sFilename)
```

The function removes the file "temp.wir". If the function should fail, it will return a nil to error and an error string such as "No such file or directory" in sErrMsg.

_rename (x:\\filename)

This function to renames a file in 1100 embedded memory. If the function fails to rename the file, it returns a nil, plus a string describing the error.

Example:

```
sOldFilename = "c:/temp.fil"
sNewFilename = "c:/values.fil"
error, sErrMsg = rename(sOldFilename, sNewFilename)
```

The function will rename the file, `temp.fil to values.fil`. If the function fails, it will return a nil in `error` and an error string in `sErrMsg`.

_seek (handle [,base] [, offset]) _seek

This function sets and gets the file position for 1100 an embedded memory file. The file position is measured in bytes by ***offset*** from ***base***. The value of ***base*** is a string with one of the following values.

base	description
set	base is position 0 (beginning of the file)
cur	base is current position
end	base is the end of the file

In case of success, the _seek function returns the file position, measured in bytes from the beginning of the file. If the call fails, it returns nil, plus a string describing the error.

The default value for ***base*** is ***cur***, and for ***offset*** is ***0***. Therefore, the call `_seek(file)` returns the current file position, without changing it; the call `seek(file, "set")` sets the position to the beginning of the file (and returns 0); and the call `seek(file, "end")` sets the position to the end of the file, and returns its size.

_write (x:\\filehandle, value1, value2, …)

Use this function to write to a file. Note, before using this function the file is opened or created in embedded memory using the `_writeto` function. After you are finished writing use `_writeto` to close the file. The `_write` function returns ***status*** and ***error*** values as shown in the full format statement below.

status, error = write(filehandle, value1, value2, ...)

If the function fails to write the file, it will return `nil` to ***error*** and a string describing the error to ***error***. If the function successfully writes a file, it will return a value other than `nil` to ***status*** and `nil` to ***error***.

_writeto ([x:\\fileame])

The full format of this function is:

filehandle, error = writeto ([x:\\filename])

This function can be used in one of two ways. One use is to open a new file in 1100 embedded memory so you can write something to it. Include the path and the extension for the filename. This command works with files with the following file extensions: .txt, .wir, .lua, .evt, .rpt, and .cmp. If successful, this function will return a handle for the opened file. If the function fails, `nil` is returned to ***handle*** and an error description to ***error***. Take note, if this function is run on an existing file, the contents of the existing file will be *completely erased*.

The other use of this function is to close the current output file. When a file is opened it becomes the current output file until it is closed or another file is opened. To close the current output file use this function without a filename.

Example

```
sFile = “c:\\storage.txt"
iCanWriteFile, errMsg = _writeto(sFile)
if iCanWriteFile then
     _write("Company XYZ")
     _writeto()      -- Close the file
end
```

In this example the function `writeto` will return the file handle in `iCanWriteFile` if it successfully opens the storage.dat file. If it fails, it will return a nil in `iCanWritefile` and an error string in `sErrMsg`. The function `_write`, writes the string, `Company XYZ`, to the file and `_writeto` then closes the file.

Preserved Lua 3.2 Functions

Easy Touch scripting uses Lua version 5.1. However scripts that executed within the Touch1 and 1100 testers were written in Lua version 3.2. Some Lua 3.2 functions have become obsolete in Lua 5.1. To allow you to more easily run the older 3.2 Lua scripts, the functionality of these obsolete functions is preserved in Cirris Easy Touch Scripting.

Preserved Lua 3.2 functions	New Lua 5.1 functions
abs	math.abs.
acos*	math.acos
ascii	string.byte.
asin*	math.asin
atan*	math.atan
atan2*	math.atan2
ceil	math.ceil
cos†	math.cos
cosh†	cosh.cos
date	os.date
deg	math.deg
execute	os.execute
exp	math.exp.
getn	table.getn
ldexp	math.ldexp
log	math.log.
floor	math.floor
foreach	table.foreach
foreachi	table.foreachi
format	string.format
frexp	math.frexp
gsub	string.gsub
Idexp	math.idexp
log	math.log

Preserved Lua 3.2 functions	New Lua 5.1 functions
log10	math.log10
max	math.max
min	math.min
mod	math.fmod
pi	math.pi
pow	math.pow
rad	math.rad
random	math.random
randomseed	math.randomseed
sin†	math.sin
sinh†	math.sinh
sort	sable.sort
sqrt	math.sqrt
strbyte	string.byte
strchar	string.char
strfind	string.find
strlen	string.len
strlower	string.lower
strrep	string.rep
strsub	string.sub
strupper	string.upper
tanh†	math.tanh
tinsert	table.insert
tremove	table.remove

* This function assumes an angle is specified in terms of degrees, whereas its replacement assumes an angle in terms of radians.

† This function returns an angle in terms of degrees, whereas its replacement Lua 5.1 function returns an angle in terms of radians.

Unsupported Cirris Functions

The following scripting functions may have been used in scripts executed within the Cirris Touch1 and 1100 testers, but are no longer supported in Easy Touch Scripting.

Unsupported Function	Notes
appendto	see io.output. You can use _appendto for 1100 embedded memory files.
remove	You can use "_remove" for 1100 embedded memory files.
rename	You can use "_rename" for 1100 embedded memory files.
Seek	Use "_seek" for embedded files.
remove	
rename	
call	See pcall and xpcall.
cleardirectory	See os.remove.
copyfile	See the Lua 5.1 functions io.input, io.output, io.open, and io.write. You can use _copyfile for 1100 embedded memory files.
Dofile	
GetButtonPress	
LoadWirelist	
LuaError	
openfile	You can use _openfile for 1100 embedded memory files.
outtextxy	
read	You can use _read for 1100 embedded memory files.
ReadBlockFromSerial	
readfrom	You can use _readfrom for 1100 embedded memory files.
ReadFromSerial	
remove	
rename	
SaveSPCData	
Seek	
SetCableSerialNumber	
SetSerialParams	
TestPreference	Preferences can now be saved on each test.
tmpname	
write	You can use _write for 1100 embedded memory files.
WriteBlockToSerial	
writeto	You can use _writeto for 1100 embedded memory files.
WriteToSerial	

Index

U

W