LUA Scripting Manual

Version 2010 1.0
August 19, 2010

CIRRIS
An ISO 9001 Certified Company

LUA Scripting Manual

Table of Contents

Chapter 1: Introduction

What is scripting?

Scripting is an alternate way of controlling the Touch1, 1100H+, or 1100R+ testers. It is a tool used to control the tester in applications when the factory default processes are not suitable to the task. Scripting makes the tester extremely flexible. It is used when the functionality of the tester must custom fit a unique task. Scripting uses the LUA programming language to create text files.

The LUA scripting language is Copyright (1994-1998 TeCGraf, PUC-Rio.)
Written by Waldemar Celes, Roberto Ierusalimschy and Luiz Henrique de Figueiredo.
All rights reserved.

Implementing scripting

Scripts are ASCII text files. You can write your own scripts or use the Cirris supplied default scripts as is or with little modification. If you modify existing scripts or create your own, rename the script and use either a text editor or a word processor running in text mode to edit the script. After editing, copy the script file(s) to the tester and attach the script to a wirelist. For the Touch1, you can copy the file to the hard drive or a network directory the Touch1 is attached to and can read. For the 1100 testers, you can copy script files using the supplied PC program: 1100 File Explorer.

Three kinds of scripts

The three kinds of scripts are: test event scripts, custom component scripts, and custom report scripts. The type of script should be chosen depending on the task.

Test Event Scripts (.evt & .lua)

Test Event Scripts are specific to each wirelist and need to be attached to each wirelist before they will run. There are two kinds of Test Event Scripts: "LUA" and "EVT." They differ by file extension name and by how they are configured. LUA scripts run only as written. EVT scripts run both by how they are written and by how any event parameters are set up in the wirelist. This allows you to use the same script with different parameters on several wirelists giving you more flexibility than a LUA script.

Note: EVT Scripts are supported by Touch1 software version 3.24 or higher. They are supported by 1100 software version 4.0 or higher.

Custom Component Scripts (.cmp)

Custom Component Scripts behave like "template" files for the Custom Components added to a wirelist. They supply the parameter format and test commands to the Components Test as each custom component is run. Creating custom components lets you either improve standard component specifications or create new components not covered in the tester's standard product.

Custom Report Scripts (.rpt)

Custom Report Scripts are custom replacements for the five standard printing tasks in the Touch1. They are system wide and do not need to be attached to a wirelist. If you need a report specific to a wirelist, use an EVT Test Event Script. You can replace any of the Touch1's following five standard print tasks with a custom report script:

- Test Summary
- Touch 1 Cable Documentation (wirelist report)
- Error Report
- Test Status - Good Only (auto-print)
- Test Status - Errors Only (auto-print)

Custom Report Scripts are currently unavailable on the 1100 testers.

When scripts run

Scripts run either when attached to a specific wirelist (Test Event Scripts and Custom Component Scripts) or when enabled at the system level (Custom Report Scripts).

Test Event Scripts (.evt & .lua)

Test event scripts must be attached to a specific wirelist. They run when the:

- Wirelist is loaded at tester power up or when manually retrieved.
  - and/or -
- The tester calls either the `DoOnTestEvent` function for LUA scripts or the `DoIt` function for EVT scripts at one of the following times:
  - 1 = Start of the test run before testing any cables
  - 2 = Start of the low-voltage test
  - 3 = End of all tests or when the cable is removed
  - 4 = Waiting to start test **(EVT scripts only)**
  - 5 = Main Screen **(EVT scripts only)**
  - 6 = After each net is hipotted **(EVT scripts only)**

Custom Components Script (.cmp)

Custom component scripts run during the Components Test in the order of the custom components listed in the wirelist. A custom component script must be attached to each wirelist before the components they contain are available. Also, the custom components must be added to the wirelist in a manner similar to adding standard components.

Custom Report Scripts (.rpt)

Custom report scripts run on the Touch1 when the standard print task they replace runs. The 1100 testers support three custom report scripts for automatic printing: autoall.rpt, autogood.rpt, and autobad.rpt.

- Test Summary Report—from the Print button in the Test Summary window
- Wirelist Report—from the Print button in the View/Change Wirelist window
- Error Report—from the Print buttons in various Error windows during testing
- Test Status - Good Only Report—automatic print on a good cable after removing the cable, pressing the Cancel, Home, or New Test buttons
- Test Status – Errors Only Report - automatic print on a bad cable after removing the cable, pressing the Cancel, Home, or New Test buttons

Uses for scripting

The following list gives some different examples for using scripts.

- Create complex tests that are easy for line workers to use
- Customize standard reports or create custom reports, such as cable tags
- Create and print custom labels
- Print using parallel or serial printers simultaneously
- Add data fields to SPC Data Collection, such as company-specific information and serial numbers (requires SPC Link, sold separately)
- Interface to external hardware, such as a bar code reader to record serial numbers
- Control external devices such as lamps, hold-down clamps, markers, etc
- Control the tester from up to four external inputs
- Create custom components to 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
- Get resistance measurements including four-wire and capacitance
- On the Touch1, test SCSI terminators (requires purchasing the KSTT-68 SCSI Terminator Adapter Kit and SCSI Terminator Script Disk)

Chapter 2: Setting up Scripting

Required Items

- Floppy disk: *Default Scripts Disk*
- This manual: *LUA Scripting Manual*
- Computer for editing scripts
- Feature Access Code if scripting is not already installed
- Touch1 System Requirements:
  - LUA Test Event Scripts: Software Version 2.4 or higher
  - EVT Test Event Scripts: Software Version 3.24 or higher
  - CMP Component Scripts: Software Version 3.17 or higher and at least 8MB RAM
  - RPT Report Scripts: Software Version 3.10 or higher
- 1100 System Requirements:
  - Software Version 4.0 or higher
  - 1100 File Explorer PC program for copying script files to the tester

Optional Items

- Serial and/or parallel printer
- Digital I/O
- Barcode scanner (uses keyboard port)
- Network card
- Monitor (Touch1 only)
- Keyboard

Scripting Set Up

Step 1:	Enable the Scripting Feature. (one time only)
Step 2:	Define the problem. Create a script to fix the problem by creating a new one or editing an existing one.
Step 3:	Copy the script file(s) to the Touch 1.
Step 4:	Attach the script file(s) to a specific wirelist or attach globally.
	Add any parameters (EVT scripts only) to the wirelist.
	Add any components (CMP scripts only) to the wirelist.
Step 5:	Turn off Touch 1 settings that conflict with the script(s).
Step 6:	Install auxiliary hardware. (optional)

Step 1: Enable Scripting Feature (one time only)

If scripting was ordered with your tester, the feature is already enabled so proceed to Step 2. If scripting was purchased separately, it will have to be enabled with a Feature Access Code you received at the time of purchase. This code is unique to each tester.

To find out if the Scripting Feature is enabled

1. Power up the Touch 1 and as soon as the **Initial Self Test** window opens, press **Pause**.
2. Under **Enabled Options**, the word *Scripting* displays if scripting is enabled.
3. Press **Resume** to continue.

To enable Scripting by entering a Feature Access Code

1. In the **Main Menu**, press **System Setup**.
2. In **System Setup**, press **Software Update**.
3. In **Software Update**, press **Enable Optional Feature(s)**.
4. In **Enter Feature Access Code**, enter the six-character code supplied with the scripting package (upper case only) and then press **OK**.
5. In **Feature Installation**, verify “Scripting” is listed and then press **OK**. If not, you may have the wrong access code and will need to start over.
6. Press **OK** or **Cancel** until you return to the **Main Menu**.

Step 2: Create and Edit Script File(s)

Create or edit script files using a text editor on a computer other than the Touch 1. If editing a script, make a copy of the original script on the PC’s hard drive. To edit the file, use an editor that does not word wrap. For example, Notepad has a Word Wrap selection. Also, do not save the script as a Word or a WordPerfect file. Use “Save As” with “Text Only” as the output format and do not use *.txt* the regular extension. Use the extension to indicate the script type: *.evt* or *.lua* for Test Event Scripts, *.cmp* for Custom Component Scripts, and *.rpt* for Custom Report Scripts.

When creating a script from scratch, make sure the required function for the script type is included. For details on the required functions, see the following pages:

LUA Test Event Scripts: see page 14
EVT Test Event Scripts: see page 17
Custom Component Scripts: see page 25
Custom Report Scripts: see page 35

Notes:

- Since you can only attach one Test Event and one Custom Component Script to a wirelist at a time, make sure all needed functions are included in these scripts.
- Because you configure the parameters at the Touch 1 for each wirelist, you can use the EVT and CMP Scripts as is if the script contains all the necessary functions. This allows you to use the same script with different parameters on several wirelists giving you more flexibility.
- For the Default Custom Report Scripts, replace the text “Company Name Goes Here” with your company name.

Step 3: Copy Script File(s)

Script files must exist on the Touch 1's hard drive or a network drive connected to the Touch 1 so they can be accessed.

Copying script files from a floppy disk to the hard drive

1. Insert the floppy disk into the Touch1's floppy disk drive.
2. In the **Main Menu**, press **System Setup**.
3. In **System Setup**, press **Disk Utilities**.
4. In **Disk Utilities**, press **Copy Files**.
5. In **Copy File(s)**, press **From**.
6. In **Select 'From' Loc & Files**, do the following:
   a. Highlight **<A:>** by pressing the up/down arrows and then pressing **Open Loc**.
   b. Mark script file(s) to copy by doing one of the following:
      Highlight each script by pressing the up/down arrows or the scroll bar between the arrows. Then press **Mark**. Repeat.
      – Or –
      Highlight a script file, and press **Mark All**.
      Note: Marked files display with a right angle (>) bracket.
   c. Press **OK**.
7. In **Copy Files**, press **To**.
8. In **Select 'To' Location**, do the following:
   a. If the correct directory is not already listed in the **Loc:** box, highlight it by pressing the up/down arrows and then press **Open Loc**.
   b. Press **OK**.
9. In **Copy File(s)**, verify the correct files are selected and then press **Copy.**

Step 4: Attach Script File(s)

To run scripts, you either attach them to each wirelist or attach them at the system level depending on the script type. If a different script is already attached, press **Clear** and then attach the new script.

Note: You do not have to reattach the script each time you make changes to it. If a message box persists, telling you have changed it, see page 48.

If you need a script that does not need to be attached to a wirelist, create a ~main.lua script file. If c:\~main.lua exists, this script will run every time the user returns to the main menu. The execution of this script can be canceled by pressing the cancel button during the initial load of the last learned wirelist into memory.

Test Event Scripts (.lua & .evt) are local to a specific wirelist and must be attached to each wirelist running the script. To turn on a Test Event Script, attach the script to an existing wirelist or to a wirelist that will be learned. With EVT scripts, press OK at the parameter screen whether or not the script contains parameters.

Custom Component Scripts (.cmp) are local to a specific wirelist and must be attached to each wirelist running the script. To turn on a Custom Component Script, attach the script to an existing wirelist or to a wirelist that will be learned. After attaching the script, add any custom components. In some cases, it is desirable to delete the standard components if the component script contains components that are replacements for the standard ones.

Custom Report Scripts (.rpt) are global to the system. They run the same for all wirelists. You turn on Custom Report Scripts by selecting them at the system level.

Attaching script files to a wirelist

A Test Event or Component Script is not available to run until it is "attached" to each wirelist. There are two ways to attach scripts to a wirelist:

Option 1: Attach a script before learning

When attaching a script before learning, the script becomes part of the "Last Modified Settings". All subsequent learns will create wirelists with the same script attached until another script is selected or scripting is turned off for learning.

1. In the **Main Menu**, press **Test Setup**.
2. In **Test Setup**, press **Learn Sample**.
3. In **Learn Sample**, press **Change**.
4. In **View/Change Learn Settings**, press **More**, **Script**, and **Change Script**.
5. In **Change Script**, press **Test Event Script** or **Custom Component Script**.
6. In **Select Script Type *.lua; *.evt**, or ***.cmp** select a script using the up/down arrows or scroll bar between the arrows and then press **Select.** For EVT scripts only, add any parameters.
7. To accept, continue to press **OK** until you get to the Learn Setup window, and then press **LEARN**.

Option 2: Attach a script by editing a wirelist

You can add or change scripts any time after the wirelist is created.

1. In the **Main Menu**, press **Test Setup**.
2. In **Test Setup**, press **View & Change Wirelist**.
3. In **View/Change Wirelist**, press **More**, **Script**, and then **Change Script**.
4. In **Change Script**, press **Test Event Script** or **Custom Component Script**.
5. In **Select Script Type *.lua; *.evt** or ***.cmp**, select a script using the up/down arrows or scroll bar between the arrows and then press **Select**. For EVT scripts only, add any parameters.
7. To accept, continue to press **OK** until you get to the Test Setup window.

Adding Custom Components (CMP scripts only) After Script is Attached to an Existing Wirelist

Custom Components are tested in the same order as they are entered. Only the custom components in the attached component script will be available for adding.

1. In **View/Change Wirelist**, press **More**, **Comp**, and then **Change Comp**.
2. In **Change Components**, press **Add**. If you want to change the testing order, highlight a component in the list just above where you want to add the new component then press **Add**.
3. In **Add Components**, press **Custom**.
4. For each custom component, do the following:
   - In **Custom Component**, highlight the desired component and press **Select**.
   - In **Add/Change: cmp [name]**, highlight a parameter and then press **Change** to edit the parameter.
5. After adding all the custom components, save the changes by pressing **OK** until you return to the **View/Change Wirelist** window.
6. Verify all components in the wirelist. Make sure custom components from the script will not conflict with components already in the wirelist. For example, if you leave in diode components with the same test points as the cmpLED component, you will get both diode error and cmpLED error reporting.

Adding Event Parameters (EVT scripts only) After Script Is Attached to a Wirelist

Parameters let you customize an EVT Test Event Script for individual wirelists without having to edit the script in a text editor for each wirelist. The event parameters available to the user are a function of how the script is written. Initially, you enter parameters when attaching a script to a wirelist. You can also change them later as needed.

Note: When entering parameters before learning, script parameter entries become part of the "Last Modified Settings". Consequently, all subsequent learns will use the same parameters until they are changed or scripting is turned off for learning.

1. In the **View/Change Wirelist** screen or the **View/Change Learn Settings** screen, press **More**, **Script**, **Change Script** and then press **Params**.
2. In **Change Event Parameters**, select a parameter using the up/down arrows or scroll bar between the arrows, and then press **Change**.
3. Enter each parameter setting.

   Note: Scripts can be set up so you do not need to enter every parameter. For example, the second address line of a report might not be used.
4. In **Change Event Parameters**, press **OK** to save the parameters.

Attaching Report Script Files at the System Level

1. In the Main Menu, press System Setup.
2. In System Setup, press Reports.
3. In **Standard/Custom Reports**, select a report type. To enable the report types for automatic printing, press **Auto On.**
4. In the specific **Report** type window, do one of the following:
   - Press **Custom Report** (check box), and then press the **.RPT** button. In **Select Custom Report**, highlight the intended report script and then press **Select**.

   – or –
   - Press **Standard Report** (check box) to revert to standard reports that do not use custom report scripts.

Step 5: Turn Off Conflicting Features

Because you can use scripting to enhance or replace standard features, it may be necessary to turn off the standard features so their operation will not conflict.

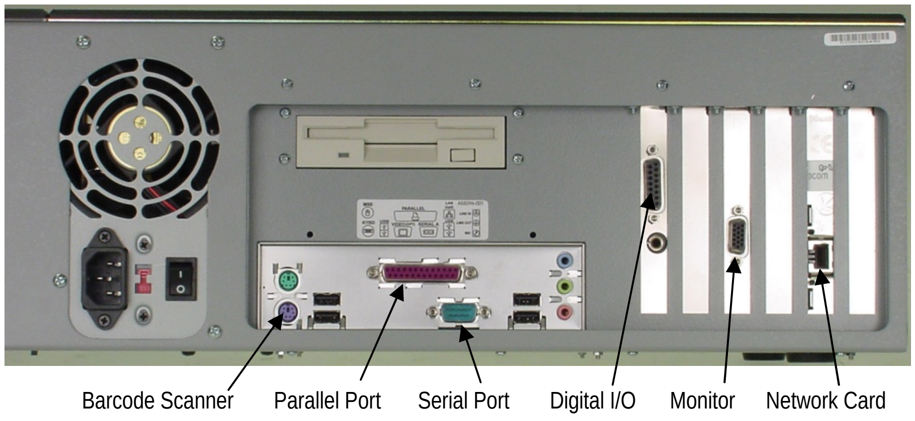
Possible conflicts:

- System versions of automatic printing can conflict with printing done by an EVT or a LUA script. System printing always executes first before any Test Event Script printing.
- The settings for the External Switch and the Hipot Safety Switch on a Touch1 tester can conflict with digital inputs controlled by scripts using the same input ports.
- The test mode setting for Single Test on a Touch1 tester can conflict with scripts which need to run in Continuous Test mode.
- Digital outputs set on the Touch1 for the I/O port can conflict with output signals set by scripts.
  - Standard reports printing to the same printer as custom reports in scripts can print on the same page. To reassign the output for the script reports to a serial printer, see the functions `SetSerialParams` and `WriteToSerial`.
- A standard component in a wirelist can conflict with a custom component script testing the same component. In some cases, it can be beneficial to use both standard and custom components to test the same component.

Step 6: Install Auxiliary Hardware

Touch 1 Tester

Note: Auxiliary hardware placement will vary with different tester serial numbers.



Parallel Printer

To install a parallel printer, use a parallel printer cable connected to the parallel port. The connector is a 25-pin female D-sub.

Note: For an example using a parallel printer, see the `goodrpt.lua` script.

Serial Printer

To install a serial printer, use a standard 9 pin serial printer cable connected to the serial port.

Note: For an example using a serial printer, see the `sernum.lua` script.

Barcode Scanner

The barcode scanner uses the keyboard port. To use both the barcode scanner and a keyboard, plug a wedge into the keyboard port. Then plug the scanner and keyboard into the wedge.

Note: For an example using a barcode scanner, see the `barcode.lua` script.

Digital Inputs and Outputs

The digital I/O port is a standard 15-pin D-sub and is on the same expansion slot card as the probe.

Note: See page 139 for a pinout of the digital inputs and outputs.

Delete Script File(s) and Security with Scripting

Deleting script file(s)

1. In the **Main Menu**, press **System Setup**.
2. In **System Setup**, press **Disk Utilities**.
3. In **Disk Utilities**, press **Delete File(s) & Location(s)**.
4. In **Delete File(s) & Location(s)**, highlight the script you want deleted and then press **Delete**.

Note: Use the **Open Loc** button to select a drive and/or location where the script resides.

Scripting and Security

Scripting can be used with security as long as the user's Security Record includes the following levels:

1. Edit Wirelist (EW) Security Level: controls turning scripts on or off.
2. Set Up Reports (SR) Security Level: controls copying script files.

1100 Scripting Setup

Step 1:	Check the Scripting Feature is enabled. This feature is a factory installation.
Step 2:	Define the problem. Create a script to fix the problem by creating a new one or editing an existing one.
Step 3:	Attach the script file(s) to a specific wirelist by editing the wirelist.
Step 4:	Copy the wirelist containing the script and the script(s) to the 1100 using the 1100 File Explorer program.
Step 5:	Turn off 1100 tester settings that conflict with the script(s).
Step 6:	Install auxiliary hardware. (optional)

Step 1: Check for Scripting Feature

If scripting was ordered with your tester, the feature is already enabled so proceed to Step 2. If scripting is purchased separately, it has to be installed at Cirris.

To find out if the Scripting Feature is enabled

1. Power up the 1100 tester and as soon as the **PASSED SELF TEST** window opens and press **PAUSE**.
2. Under **Options: Script** will be displayed if the scripting feature is enabled.
3. Press **CONTINUE** to proceed to the **Main Menu**.

Step 2: Create and Edit Script File(s)

Create or edit script files using a text editor on a computer. If editing a script, make a copy of the original script on the PC's hard drive. To edit the file, use an editor that does not word wrap. For example, Notepad has a Word Wrap selection. Also, do not save the script as a Word or a WordPerfect file. Use "Save As" with "Text Only" as the output format and do not use *.txt* the regular extension. Use the extension to indicate the script type: *.evt* or *.lua* for Test Event Scripts and *.cmp* for Custom Component Scripts. When creating a script from scratch, make sure the required function for the script type is included. For details on the required functions, see the following pages:

LUA Test Event Scripts: see page 14
EVT Test Event Scripts: see page 17
Custom Component Scripts: see page 25

Note: Since you can only attach one Test Event and one Custom Component Script to a wirelist at a time, make sure all needed functions are included in these scripts.

Step 3: Attach Script File(s) to a Wirelist

To run a script for a given wirelist, the wirelist must contain a script section. Both the wirelist containing the script section and the script must then be copied to the 1100 tester. To add the script section, the wirelist must be edited as follows. The heading for the script section is SCRIPT and it is followed by line(s) containing the script type. Both .evt and .cmp scripts contain parameters which must be included in the wirelist. For .cmp scripts, the component parameters are added under the component section, CHECK COMPONENTS, in the wirelist. For .evt scripts, the parameters are part of the test event script line. The crc and time will be updated when the wirelist is loaded into the tester's memory.

Example Syntax for a Lua Test Event Script:

```
SCRIPT
TestEvents("C:\TEST.LUA", "crc:9WDUB7", "time:1078853798")
```

Example Syntax for an Evt Test Event Script:

```
SCRIPT
TestEvents("C:\TEST.EVT", "crc:2T153M", "time:1082480186", evtEvent(123))
```

Example Syntax for a Component Script:

```
CHECK COMPONENTS
1 cmpTest(19,5)
SCRIPT
  Components("C:\TESTCMP.CMP", "crc:32F010", "time:1079608222")
```

Note: A wirelist can only have one Test Event Script (either a .lua or an .evt) and/or one Component Script as the following example shows.

```
SCRIPT
  TestEvents("C:\TEST.LUA", "crc:9WDUB7", "time:1078853798")
  Components("C:\TESTCMP.CMP", "crc:32F010", "time:1079608222")
```

Note: You do not have to reattach the script each time you make changes to it. If a message box persists, telling you have changed the script, see page 48.

If you need a script that does not need to be attached to a wirelist, create a ~main.lua script file. If c:\~main.lua exists, this script will run every time the user returns to the main menu. The execution of this script can be canceled by pressing the cancel button during the initial load of the last learned wirelist into memory.

Step 4: Copy Script File(s)

Script files and their corresponding wirelist must exist on the tester so they can be accessed. Use the supplied PC program: 1100 File Explorer to copy both scripts and wirelists containing scripts to the tester. To do the file transfer, follow these steps:

- Select the directory on the PC where the wirelist(s) and/or script(s) reside.
- Highlight the files you want to copy to the tester.
- Press the "Copy to Tester" button to begin the transfer.

Step 5: Turn Off Conflicting Features

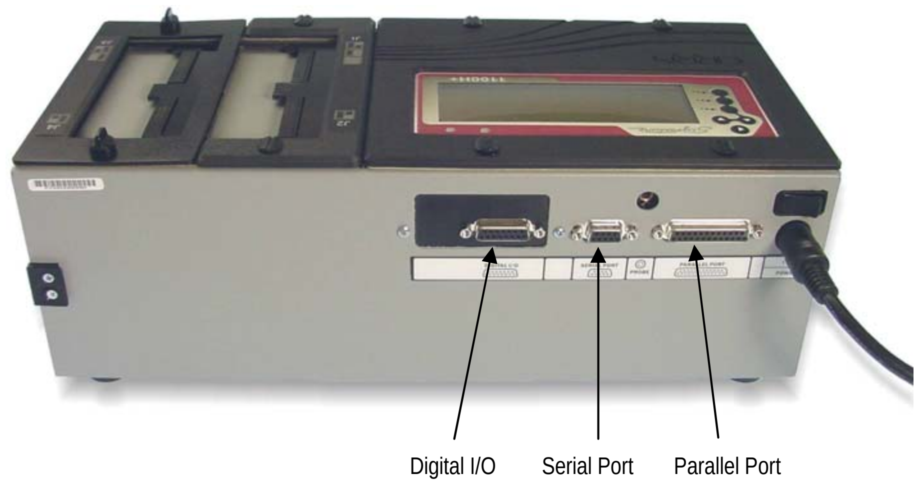
Because you can use scripting to enhance or replace standard features, it may be necessary to turn off the standard features so their operation will not conflict.

Possible conflicts:

- System versions of automatic printing can conflict with printing done by an EVT or a LUA script. System printing always executes first before any Test Event Script printing.
- The settings for the External Switch and the Hipot Safety Switch on the tester can conflict with digital inputs controlled by scripts using the same input ports.
- The test mode setting for Single Test on the tester can conflict with scripts which need to run in Continuous Test mode.
    - Digital outputs set on the tester for the I/O port can conflict with output signals set by scripts.
    - A standard component in a wirelist can conflict with a custom component script testing the same component. In some cases, it can be beneficial to use both standard and custom components to test the same component.

Step 6: Install Auxiliary Hardware

1100H+ Tester



Parallel Printer

To install a parallel printer, use a parallel printer cable connected to the parallel port. The connector is a 25-pin female D-sub.

Serial Printer

To install a serial printer, use a standard 9 pin serial printer cable connected to the serial port.

Digital Inputs and Outputs

The digital I/O port is a standard 15-pin D-sub and is on the same expansion slot card as the probe.

Note: See page 139 for a pinout of the digital inputs and outputs.

Chapter 3: LUA Test Event Scripts

Overview

LUA Test Event Scripts are the simplest kind of script. Unlike CMP and EVT scripts, LUA scripts have no parameters.

Scripting ships with four sample LUA Test Event Scripts on the *Default Scripts Disk*:

- Barcode. lua
- Sernum. lua
- Goodrpt.lua
- Badrpt. lua

Because you can only attach one Test Event Script file at a time to a wirelist, all the functions you want active in one test session must be in one script. You may need to combine functions from several scripts into one script.

If you need a script that does not need to be attached to a wirelist, create a ~main.lua script file. If c:\~main.lua exists, this script will run every time the user returns to the main menu. The execution of this script can be canceled by pressing the cancel button during the initial load of the last learned wirelist into memory.

LUA Test Event Syntax

DoOnTestEvent

Explanation:

Each different script type has required elements. For LUA Test Event Scripts, the DoOnTestEvent function is a required element. **The function DoOnTestEvent ( ) must be included in your LUA test event script file**. Use this function to call other script functions depending on the tester’s current test event.

Format:

```
DoOnTestEvent (iEventType)
    ↑               ↑
 FUNCTION         INPUT
```

INPUT `iEventType` (Integer containing the test event)	DESCRIPTION
1	Start of test run: • Press Test Cable (Touch1 or 1100: Main Screen) or TEST (Touch1: Test Setup Screen or 1100: Test Retrieved)
2	LV Test Started: • Press START TEST (Touch1 or 1100: Test Screen) • Cable is attached
3	End of all tests: • For continuous test mode: cable is removed • For single test mode: cable is removed or end of required tests

Examples:

-
  ```
  function DoOnTestEvent (iEventType)
      if iEventType = = 1 then
              do other functions here
      end
      elseif iEventType = = 3 then
              do other functions here
      end
   end
  ```

-
  ```
  function DoOnTestEvent(iEventType)
      if iEventType = = 1 then
        MessageBox("TEST RUN IS STARTING")
      elseif iEventType = = 2 then
        startNum = PromptForUserInformation(1, "ENTER ", "START  NUMBER", 30, "")
      elseif iEventType = = 3 then
        OutputReportToSerialPrinter(theReport)
      end
  end
  ```

This example will display the message "TEST RUN IS STARTING" when `iEventType` is 1 (start of the test run). It will then prompt the user for a start number when `iEventType` is 2 (low voltage test begins). Finally, when `iEventType` is 3 (end of the test) it will output a report to the serial printer. The report definition and the setup for the serial printer are done outside of the function and this example.

Sample LUA Test Event Scripts

Goodrpt.lua (Touch1 only)

Goodrpt.lua replaces the system Test Status – Good Only report with a report you can customize for each wirelist (system reports are the same for all wirelists). The report contains the following user-defined fields: *Company Name*, *Address*, *Shift*, and *Cable Serial Number*. At the start of each test run, the operator is prompted for the work shift code, operator name (if security is turned on), and the serial number from the last test (which the operator can change). After each good cable test, the report automatically prints on a parallel printer.

Note: To avoid duplication, turn off standard automatic printing (see page 9).

```
                    TEST STATUS – GOOD ONLY

Your Company Name
Your Address
                                              Date:  8/10/2001
                                              Operator:
                                              Shift:

                         GOOD CABLE
Filename: Wirelist.wir
CRC Signature: F3B493-6F350
Cable Description:  My Cable
Cable Serial Number:  23432BA
```

Badrpt.lua

Badrpt.lua replaces the system Test Status - Errors Only report with a report you can customize for each wirelist. The report contains these user-defined fields: *Company Name* and *Report Title*. "Operator" is automatically entered if Security is on. The report automatically prints after a bad test to the parallel port. If you need to output to a serial port, modify the script.

Barcode.lua (Touch1 only)

Barcode.lua prompts for bar-coded cable serial numbers. For each good cable, it saves the serial number to SPC (Statistical Process Control) data collection. The serial number can be entered by hand or scanned using a barcode scanner.

Required Items

- Barcode scanner that plugs into the computer's keyboard.
- SPC Link feature (purchased separately).

Sernum.lua (Touch1 only)

Sernum.lua or Serlabel.lua prints a label to a serial printer after each good test. The label contains a "Company Name," the date, and a serial number. At the start of each test run Serlabel.lua prompts the operator to confirm the first serial number that has been stored from the last series of tests. At the end of each good cable test, the script prints the label, updates the cable serial number, and stores it to disk for the next test.

```
COMPANY NAME       12/30/1998
Cable Serial #:   93
```

Note: For the script file contents, see page 43.
To enter your own company name, see page 47.

Chapter 4: EVT Test Event Scripts

Overview

EVT Test Event Scripts run both by how they are written and by which event parameters are selected and set up for each wirelist. Event parameters are the parts of an EVT Test Event script you can configure at the tester without having to edit the script in a text editor. These parameters let you use one script for several different wirelists. Because you can only attach one Test Event Script file at a time to a wirelist, all the functions you want active in one test session must be in one script. Consequently, you can combine functions from any EVT default script into one script. Scripting ships with two sample EVT Test Event Scripts on the *Default Scripts Disk*:

- bar_file.evt
- hvplot.evt

Note:

There is only one instance of an evt event script so a parent wirelist cannot call a child wirelist that has an event script in it. The child's events will not be called because the parent's events are still in memory.

Requirements:

Touch1 software versions 3.24 and higher and 1100 software versions 4.0 and higher

EVT Test Event Syntax

Each different script type has required elements. For EVT Test Event Scripts, each event defined in an EVT Test Event Script consists of four parts. The function DoIt() is one of these required parts. The four required parts are:

- Event Name: evtEvent
- Event Description
- Event Parameter(s)
- evtEvent.DoIt Test Function

Example:

```
evtEvent = {}
evtEvent.description = “PLC EVT Test Event Example”
evtEvent.params = {
   {"Event Parameter1", "Number”, 10 }
   {"Event Parameter2", "textlist", {"ONE", "TWO"}  }

function evtEvent.DoIt(iEventType, eventParm1, eventParm2)
   local iResult, sErrorText

   if iEventType == 1 then
           do other functions here using event parameters
   end
   elseif iEventType == 5 then
           do other functions here using event parameters
   end

   if testPassed then
          return iResult
   else
          return iResult, sErrorText
end
```

Event Name:

An event name starts with a lowercase *evt* and must be set equal to curly brackets {} not parentheses (). Currently, the only event name allowed is evtEvent.

Example: evtEvent = {}

Event Description:

An event description must be inside curly brackets {} not parentheses (). This description does not display on the tester.

Example: evtEvent. Description = "Hipot Multiple Nets"

Event Parameter(s):

- The overall event parameter definition must be inside curly brackets {}, not parentheses () and each event parameter must be inside curly brackets {}.
- Each event parameter must end with a comma except the last line.
- Each parameter consists of three parts. The three parts are:
  - Parameter Description
  - Parameter Type
  - Parameter Default Value

Parameter Description:

Parameter descriptions are displayed in a list when editing parameters for the script. They must be enclosed in quotes.

Parameter Type & Default Value:

The parameter type must be enclosed in quotes. The default value for the parameters can be changed in the edit windows on the Touch1.

Parameter Examples:

- evtEvent.params = {"Hipot Duration", "number", 0.100}

  where:

  Hipot Duration = parameter description

  number = parameter type

  0.100 = parameter default value
- evtEvent.params = {"Cable Color", "textlist", {"Red", "Green", "Black"}

  where:

  Cable Color = parameter description

  textlist = parameter type

  Red, Green, Black = default values for the scroll box

PARAMETER TYPE (in quotes)	**RANGES FOR DEFAULT VALUE**	**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 a custom component can output to a test point. These current values are calibrated during power up for each tester.
"number"	Floating point: maximum four points before and after decimal	Used for components that need a numeric value. Also used 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 used to tie 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 wirelist. This point can be a fourwire point. This type can be used to 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 – 5 Meg	Resistance of a component. For milliohm, fourwire, and values greater than 5Meg, use the "number" parameter.
"string"	30 characters, maximum	Used to display text on the tester using message or dialog boxes
"textlist"	When editing on the Touch 1, the list will display in a scroll box if it is greater than one screen.	Allows the script to do different options depending on the user's 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 the user's 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 – 1000 *or* 1500 tester dependent	High voltage applied to a test point. For other voltages or setting external voltages, use the "number" parameter.

DoIt Function

Explanation:

The function evtEvent.DoIt() must be included in your EVT test event script file. Use this function to call other script functions depending on the test event.

Format:

```
evtEvent.DoIt(iEventType, eventParm1, eventParm2)
  ↑                    ↑
FUNCTION             INPUTS
```

INPUT `iEventType`	DESCRIPTION (Integer containing the test event)
1	Start of test run: • Press Test Cable (Touch1 or 1100: Main Screen) or TEST (Touch1: Test Setup Screen or 1100: Test Retrieved)
2	LV Test Started • Press START TEST (Touch1 or 1100: Test Screen) • Cable is attached
3	End of all tests: • For continuous test mode: cable is removed • For single test mode: cable is removed or end of required tests
4	Waiting to start test (can be called repeatedly): • continuous test start • single test start or summary • single test cable error
5	Enter the Main Menu Screen
6	After each net is hipotted (used for hipot graphing - see hvplot.evt)

Example:

```
evtEvent = {}
evtEvent.description = "Output report with cable serial number "
evtEvent.params = {{"sMessage1", "string", "ENTER CABLE SERIAL NUMBER"},
                   {"sMessage2", "string", " TEST RUN IS STARTING "}}
function evtEvent.DoIt(iEventType, sMessage1, sMessage2)
        if iEventType == 1 then
                MessageBox(sMessage2)
        elseif iEventType == 2 then
                cableSerNum = PromptForUserInformation(1, sMessage1, 30, "")
        elseif iEventType == 3 then
                OutputReportToSerialPrinter(theReport)
        end
end
```

This example will display the message "TEST RUN IS STARTING" when `iEventType` is 1 (start of the test run). It will then prompt the user for a start number when `iEventType` is 2 (low voltage test begins). Finally, when `iEventType` is 3 (end of the test) it will output a report to the serial printer. The report definition and the setup for the serial printer are done outside of the function and this example.

Sample EVT Test Event Scripts

bar-file.evt (Touch1 only)

This script allows the operator to use a barcode to change directories and then load wirelist files. It has two event parameters: text for a prompt box where the directory or filename is scanned with the barcode reader or typed in and the drive for the directory path. The script prompts the user to select the wirelist to load. It automatically decides if the entered information is a file or a directory. If it is a directory, the script changes to that directory. If it is an existing wirelist file in the current directory, it loads it.

Barcodes should not include the ".wir" extension for wirelist filenames. This also applies to filenames typed in using the touch screen or keyboard.

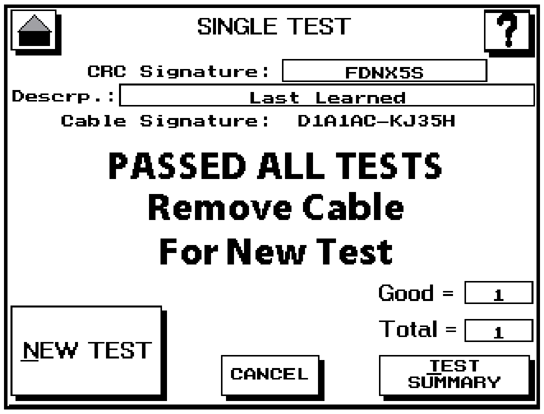
This script requires Touch1 software version 3.25 or later.

hvplot.evt (Touch1 only)

This EVT test event script displays a graph of the high voltage test. Every hipot test will be displayed and if there is a hipot failure, a pause will occur so you can view what happened. The graph is valuable for dielectric failures because it shows you where in time the failure occurred.

This script requires Touch1 software version 3.26 or later. It also requires a monitor since the hipot graph shows at the bottom of the monitor display and is not visible on the touch screen.

Good High Voltage Test Example



SINGLE TEST

CRC Signature: FDNX5S

Descrp.: Last Learned

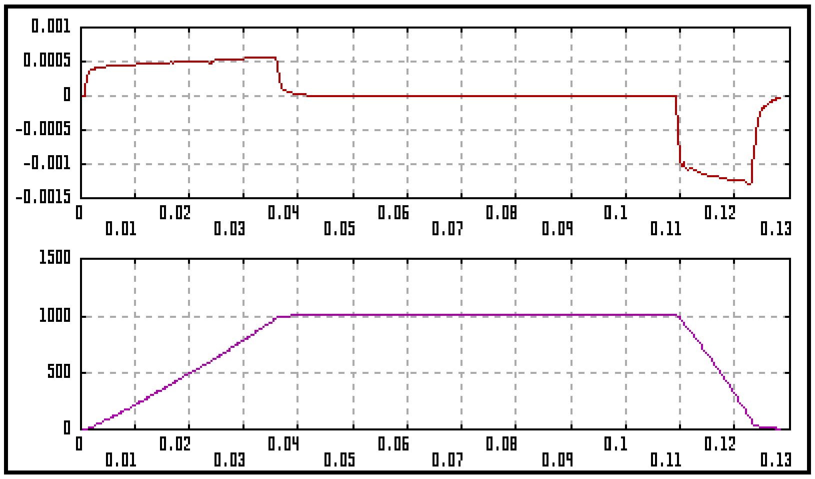
Cable Signature: D1A1AC-KJ35H

PASSED ALL TESTS
Remove Cable
For New Test

Good = 1

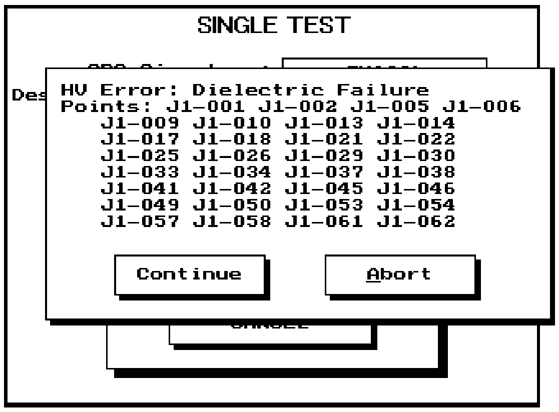
Total = 1

These pictures are the results for a good hipot test with a short dwell time. The bottom picture displays two graphs. The upper graph represents the current sensed by the Touch1's high voltage supply. The lower graph represents the voltage applied to the cable. Note the current is positive while the cable is charged up to the hipot voltage and the current is negative while the voltage is ramping down. The middle portion of the top graph shows what is wanted: a nice flat section that may even decrease slowly to near-zero current. The insulation resistance test is made at the end of the flat section while the rest of the test would be considered the ramp-up and DWV portions of the test.



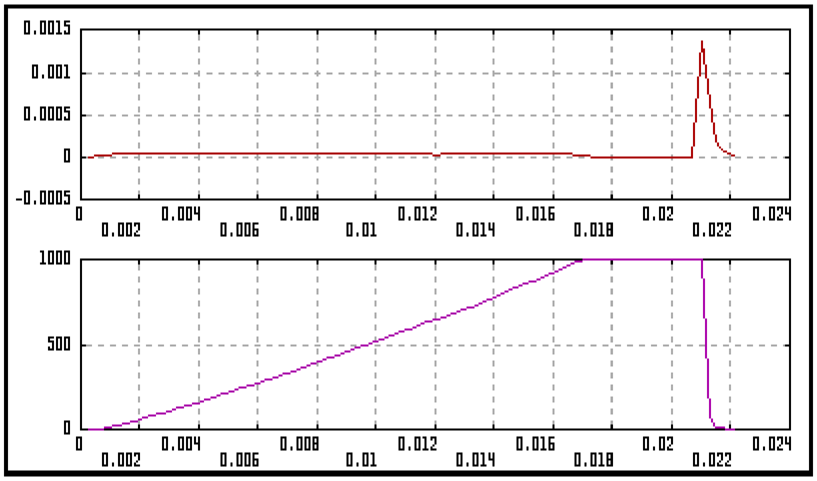
The vertical axis is current in amps for the top graph and voltage for the bottom graph.
The horizontal axis for both graphs is time in seconds.

Dielectric Failure Example



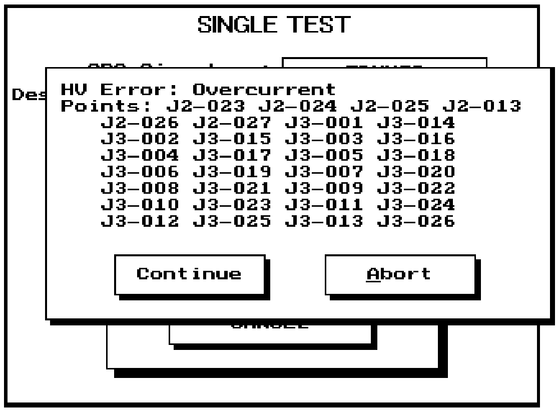
```
HV Error: Dielectric Failure
Points: J1-001 J1-002 J1-005 J1-006
   J1-009 J1-010 J1-013 J1-014
   J1-017 J1-018 J1-021 J1-022
   J1-025 J1-026 J1-029 J1-030
   J1-033 J1-034 J1-037 J1-038
   J1-041 J1-042 J1-045 J1-046
   J1-049 J1-050 J1-053 J1-054
   J1-057 J1-058 J1-061 J1-062
```

These two pictures are examples of a hipot test with a dielectric failure that occurred 0.004 seconds after it had ramped up to full voltage. The bottom picture displays two graphs. The upper graph represents the current sensed by the Touch1's high voltage supply. The lower graph is the voltage applied to the cable. While the voltage was ramping up, the current was slightly positive. When an arc occurred on the cable, the current increased dramatically. The high voltage supply sensed this and shut down.



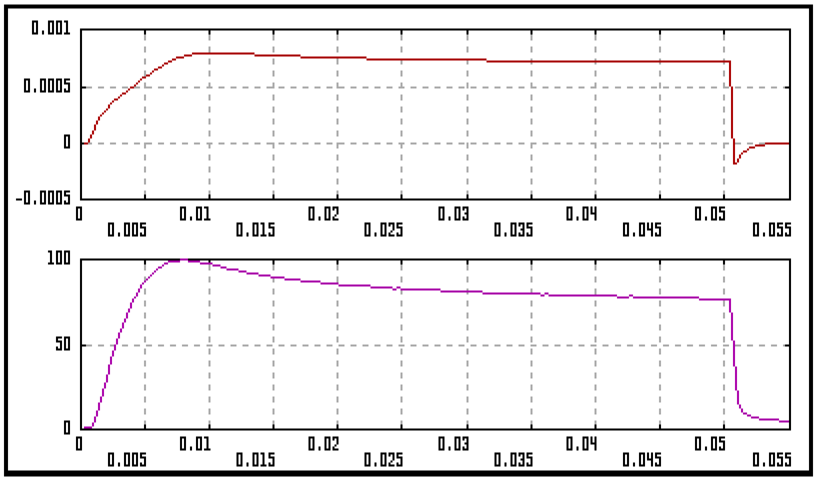
The vertical axis is current in amps for the top graph and voltage for the bottom graph. The horizontal axis for both graphs is time in seconds.

Overcurrent Example



HV Error: Overcurrent
Points: J2-023 J2-024 J2-025 J2-013
J2-026 J2-027 J3-001 J3-014
J3-002 J3-015 J3-003 J3-016
J3-004 J3-017 J3-005 J3-018
J3-006 J3-019 J3-007 J3-020
J3-008 J3-021 J3-009 J3-022
J3-010 J3-023 J3-011 J3-024
J3-012 J3-025 J3-013 J3-026

These two pictures are examples of a hipot test with an Overcurrent failure. The bottom picture displays two graphs. The upper graph represents the current sensed by the Touch1's high voltage supply. The lower graph is the voltage applied to the cable. While the voltage was ramping up, the tester sensed the current required from the high voltage supply was approaching a safety limit. The high voltage supply slowed its ramp up and even dropped the voltage to attempt to reduce the current delivered to the cable. After 0.05 seconds, the tester decided that delivering more charge to the cable could be unsafe to an operator and turned off the high voltage supply. This drained the voltage from the cable.



The vertical axis is current in amps for the top graph and voltage for the bottom graph. The horizontal axis for both graphs is time in seconds.

Chapter 5: Custom Component Scripts

Overview

Custom Component Scripts are text files that act as “templates” for the Custom Components you add to a wirelist. Before selecting which custom components to add to a wirelist, the Custom Component Script file containing the components must be attached to the wirelist. The custom components are then tested in the same order as they are added to the wirelist.

Scripting ships with three Custom Component Scripts you can use as is, modify, combine, or use as examples for scripts you write yourself. The scripts are:

- custom1.cmp
- hipot_01.cmp
- zener_01.cmp

Custom Component Syntax

Each different script type has required elements. For CMP Custom Component Scripts, each custom component defined in a Custom Component Script consists of four parts. The function .test() is one of these required parts. The four required parts are:

- Component Name
- Component Description
- Component Parameter(s)
- cmp Test Function

EXAMPLE:

```
cmpMyComponentName = {}
cmpMyComponentName.description = “Custom Component Example”
cmpMyComponentName.params = {
   {"Component Parameter1", "Number”, 10 }
   {"Component Parameter2", "textlist", {"ONE", "TWO"}  }

function cmpMyComponentName.test(compParm1, compParm2)
   local iResult, sErrorText

    do other functions here using component parameters

   if testPassed then
     return iResult
   else
     return iResult, sErrorText
    end
end
```

Component Name:

Component names must start with lowercase *cmp* and be set equal to curly brackets {}, not parentheses (). This name is displayed in the component screen where all components are listed.

Example: cmpHipotNets = {}

Component Description:

Descriptions must be inside curly brackets {}, not parentheses (). This description is displayed on the Touch 1 in the component screen when adding a custom component.

Example: cmpHipotNets.description = “Hipot Multiple Nets”

Component Parameter(s):

- The overall custom component parameter definition must be inside curly brackets {}, not parentheses () and each component parameter must be inside curly brackets {}.
- Each component parameter must end with a comma except the last line.
- Each parameter consists of three parts. The three parts are:
    - Parameter Description
    - Parameter Type
    - Parameter Default Value

Parameter Description:

Parameter descriptions are displayed in a list when editing parameters for the script. They must be enclosed in quotes.

Parameter Type & Default Value:

The parameter type must be enclosed in quotes. The default value for the parameters can be changed in the edit windows on the Touch1.

Component Parameter Example:

```
cmpLED.params = {{"anode”, "point", 1},
                 {“cathode”, “point”, 2},
                 {“Prompt Message”, “string”, “LED Is Lit?”}}
    where:
        parameter description = anode, cathode, Prompt Message
        parameter type = point, point, string
        parameter default value = 1, 2, LED Is Lit?
```

PARAMETER TYPE (in quotes)	**RANGES FOR DEFAULT VALUE**	**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 a custom component can output to a test point. These current values are calibrated during power up for each tester.
"Number"	Floating point: maximum four points before and after decimal	Used for components that need a numeric value. Also used 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"	**< 0** = treat as a separate point (default) **≥ 0** = tie all points with this number tag together (can have multiple number tags)	Any one test point in a wirelist. This point can be a Fourwire point. This type can be used to tie nets together. Points and pointlists can also be tied together by assigning the same number tag.
"Pointlist"	**< 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 pointlists can also be tied together by assigning the same number tag.
"Resistance"	.1 - 5000000	Resistance of a component. For milliohm, fourwire, and values greater than 5Meg, use the "number" parameter.
"String"	30 characters, maximum	Used to display text on the tester using message or dialog boxes
"Textlist"	When editing on the Touch 1, the list will display in a scroll box if it is greater than one screen.	Allows the script to do different options depending on the user's 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 the user's 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 – 1000 *or* 1500 tester dependent	High voltage applied to a test point. For other voltages or setting external voltages, use the "number" parameter.

.test Function

Explanation:

The function cmp<componentName>.test() must be included in your custom component script file. Use this function as a basis for calling other script functions during component testing. Each input for the function corresponds to a component parameter defined in the script file. The contents of the test function contain anything to be done when the custom component is testing. If the test function passes, a number is returned and is recorded as the measured value. If the test fails, the first part of the return is a number recorded as the measured value. The second part of the return is text that displays in the test error screens on the tester.

Format:

```
cmp<componentName>.test(definedParmX [, definedParmX])
                 ↑                    ↑
             FUNCTION             INPUT(S)
```

INPUT(S)	DESCRIPTION
`<componentName>`	Name given to the custom component.
`definedParmX`	Parameter(s) defined in the custom component parameter section.

Example:

```
•  cmpWiggle = {}
cmpWiggle.description = "Wiggle Test"
cmpWiggle.params = {{"Timeout Time", "number", 5.0},
          {"Prompt Message", "string", "Wiggle for a while..."}}

iWiggleTesting = 0 --need this to avoid infinite loops
function cmpWiggle.test(timeouttime, message)
   if iWiggleTesting == 0 then
      local iDNum = DialogOpen(message, "Continue")
      local iStartCounter = GetTimeAsInteger(4)
      local iCurrentCounter = iStartCounter
      local iBtnNum = 0
      local iTimeoutCount = timeouttime * 1000
      iWiggleTesting = 1 --need this to avoid infinite loops
      while (iCurrentCounter - iStartCounter) < iTimeoutCount do
         iFailed = TestWirelist(3)
         if iFailed == 0 then
            SetUserOutputStates(5,0,6,1)
         else
            SetUserOutputStates(5,1,6,0)
         end
         iCurrentCounter = GetTimeAsInteger(4)
         iBtnNum = DialogCheckBtn(iDNum)
         if iBtnNum > 0 then
            iCurrentCounter = 10000000    -- they hit a button
         end
      end
      DialogClose(iDNum)

      iWiggleTesting = 0 --need this to avoid infinite loops
      if iFailed == 0 then
         return 1
      else
         return 0, "Bad Wiggle"
      end
   else
      return 1
   end
end
```

Sample Custom Component Scripts

Custom Components in Custom01.cmp Script

Script File:	Component Name:	Explanation:
Custom01.cmp	cmpWiggle	Flex test to detect intermittent opens and shorts. Setup Requirements: • Use only one cmpWiggle component per wirelist • Use only Single Test Mode under Test Controls
	cmpButtonLED_NO	Tests normally open momentary switches in open & closed states. The test ends when "Abort" is pressed in the message box or when the closed switch resistance is less than the max resistance setting.
	cmpLEDBicolor	Tests both colors of a bicolor LED.
	cmpLED	Lights an LED to verify it is in the right place. This test only lights up the LED for visual testing. To do an electrical test, add in the wirelist a diode component with the same anode & cathode test points.

Component Parameter Descriptions in Custom01.cmp Script

cmpWiggle Parameters

Parameter Description	**Parameter Type**	**Description**
Timeout Time	number	The time the wiggle component takes control of testing by continuously testing the entire wirelist, including components. Recommended Range: 2 to 5 seconds **Note**: You can cancel Wiggle in the dialog box that pops up during testing
Prompt Message	string	Operator message (30 characters maximum) displayed when the wiggle component has taken over testing.

cmpLEDBicolor Parameters

Parameter Description	**Parameter Type**	**Description**
Red Anode Red Cathode	point	Test points of the bicolor LED. Most LED's distinguish the cathode by a flat part on the plastic housing.

cmpButtonLED_NO Parameters

Parameter Description	**Parameter Type**	**Description**
Switch Points 1 & 2	point	Test points connected to the switch.
Label	string	Operator message (30 characters maximum) displayed during the testing of the switch.
Max Resistance	number	Maximum allowable resistance a closed switch contact can have for a good test.
Anode Cathode	point	Connection points of the indicator LED used as an aid to the operator during the switch test. Most LED's distinguish the cathode by a flat part on the plastic housing. LED Setup: * Connect the LED directly to the tester—no series resistor required. * Place the LED close to the switch you want the operator to press.

cmpLED Parameters

Parameter Description	Parameter Type	Description
anode cathode	point	Test points of the LED. Most LED's distinguish the cathode by a flat part on the plastic housing.
Prompt Message	string	Operator message (30 characters maximum) displayed when the LED component is testing.

Component Parameters Error Text in Custom01.cmp Script

Bad cmpWiggle

Error Text:	Explanation:
Bad Wiggle	An intermittent was detected during the portion of the test controlled by the Wiggle component. Intermittents are defined by cables that test good, then bad, then good.

Bad cmpLEDBicolor

Error Text:	Explanation:
Bicolor Diode Reversed Bad or Missing Diode	Documentation errors: • Anode & cathode are swapped Assembly errors: • Reversed leads • Missing part • Defective part

Bad cmpLED

Error Text:	Explanation:
Not Lit Up	Documentation errors: • Anode & cathode are swapped. Assembly errors: • Reversed leads. • Missing part. Defective part. Operator Error: • Pressed "No" when LED was lit. **Note:** The cmpLED component only lights up the LED for visual testing. Add a Diode Component with the same anode & cathode test points to do an electrical test.

Zener_01.cmp Script

Custom Component in Zener_01.cmp Script

Script File:	Component Name:	Explanation:
Zener_01.cmp	cmpZenerDiode	Test the reverse breakdown voltage of zener diodes. **Note:** For details, see the application note that ships with the Zener Adapter Kit.

Hipot_01.cmp Script

Custom Component in Hipot_01.cmp Script

Script File:	Component Name:	Explanation:
Hipot_01.cmp	cmpHipotNets	Test selected nets at different voltages: • Test everything at a lower voltage using the normal High Voltage test then use a cmpHipoNets component for each net or group of nets tested separately at different voltages. • Test only selected nets, not the whole cable, by turning off the High Voltage Test and in place, using a cmpHipotNets component for each net or group of nets. Fast Test by grouping nets: • You can test a ribbon cable in just three tests by grouping 1st, 2nd, & 3rd points together. For example, use three cmpHipoNets by connecting: Pins 1, 5, 9, 13 etc. in the first Pins 2, 6, 10, etc. in the second Pins 3, 7, 11 etc. in the third.

Component Parameter Descriptions in Hipot_01.cmp Script

Parameter Description	Parameter Type	Description
1 Point in each Net	pointlist	A test point from each net for all nets treated as a group. **Note:** The net or group of nets (defined by one point in each net) is brought to the selected voltage then tested against everything else. Therefore, no hipot test will be done if every net is in one HipotNets component. **Tip**: To determine which test points to choose, print the wirelist from View/Change Wirelist.
Voltage	number	Voltage applied during the part of the test controlled by the cmpHipotNets component. Range: 1000V: 50 to 1000Vdc +/-5% 1500V: 50 to 1500Vdc +/-5% Recommended: one volt steps
Insulation Resistance	number	The minimum insulation resistance allowed between unintended connections. (Implies the maximum allowable current leakage of insulation.) Good Insulation has greater resistance than the Insulation Resistance setting. Range: 5 Meg to 1 Gig ohms +/-10%
Hipot duration	number	Amount of time hipot voltage is applied to each net of test points during the cmpHipotNets component test. Range: 0.01 sec. to 120 sec.
Max Soak Time	number	Maximum amount of time hipot voltage is applied before testing is done by the cmpHipotNets component. Range: 0 to 120 seconds.

```
cmpHipotNets = {}
cmpHipotNets.description = "Hipot Multiple Nets"
cmpHipotNets.params = {{"1 Point in each Net", "pointlist", 0},
                        {"Voltage", "number", 50.0},
                        {"Insulation Resistance", "number", 10000000},
                        {"Hipot duration", "number", 0.100},
                        {"Max Soak Time", "number", 0.010}}

function cmpHipotNets.test (thePointsArray, voltage, resistance, duration, max soak)
        -- This custom component hipots multiple nets as if they were one
        local fRes, iErr
        local thePoints, iIndex
        iIndex = 1
        thePoints = ""
        while thePointsArray[iIndex] ~= nil do
                -- convert the incoming array into a list of points in a text string
                thePoints = thePoints .. thePointsArray[iIndex] .. " "
                iIndex = iIndex + 1
        end

        local iDNum = DialogOpen ("Warning: High Voltage")
        iErr, fRes = HipotNetsTiedToPoints (thePoints, voltage, resistance, duration, max soak);
        DialogClose(iDNum)
        if fRes == nil then
                fRes = 0
        end

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

        if iErr == 0 then
                return iErr -- passed hipot
        else
                return iErr, sErr -- failed hipot
        end
end
```

Chapter 6: Custom Report Scripts

Overview

You can use Custom Report Scripts to add company ID and user information to print functions otherwise handled by standard reports. Custom reports print in place of standard reports to a parallel printer and before any report controlled by a Test Event Script. Scripting ships with five example custom report scripts on the *Default Scripts Disk* you can use with very little modification.

The 1100 testers support three Custom Report Scripts: autoall.rpt, autogood.rpt, autobad.rpt.

Default Report Script	**Replaces**
Autobad.rpt	Test Status - Errors Only—auto-print after bad test.
Autogood.rpt	Test Status - Good Only–-auto-print after good test
Errors.rpt	Error Report – printed from Test Error windows
Testsum.rpt	Test Summary–-printed from Test windows
Wirelst.rpt	Touch 1 Cable Documentation (wirelist)

Custom Report Syntax

Each different script type has required elements. For RPT Custom Report Scripts, there are two parts. The function DoCustomReport () is one of these required parts. The two required parts are:

- Report Definition
- DoCustomReport Function

Report Definition:

- The overall report definition must be inside curly brackets {}, not parentheses () and each individual report line section must be inside curly brackets {}.
- Each report line section must end with a comma except the last line.
- For each report line section, anything in the quotes portion will be printed. The quotes portion can contain a format string which will be evaluated. See the format function (page 75) *Chapter 10, Function Descriptions* for details.

Example:

```
myGoodCableCount = 187
{"Tested %i cables", myGoodCableCount}
```
evaluates to: `Tested 187 cables`

Report Definition Example:

```
local theReport =
   {
   { "COMPANY XYZ        %s\r", GetDateAsText(5) },
   { "Cable Serial #:  %s\r", myBarcodeNumber },
   { "\r\r\r" }
   }
```

DoCustomReport Function

Explanation:

The function DoCustomReport() must be included in your custom report script file. Use this function as a basis for calling other script functions to create and print your custom report. The KEYWORDS `function` and `end` are described in the "Function Definitions" section on page 50.

Note: To do custom reports with a custom component cmp. script or test event evt script using parameters, you must pass those parameters to the `DoCustomReport()` function. The scope for event parameters is only the `DoIt()` function and the scope for component parameters is only the `.test()` function.

Format:

```
function DoCustomReport()
                    ↑
                FUNCTION

            Gather report information
            Send report to parallel or serial printer
end
```

Example:

```
function myCustomReport ()
    local theReport =
    {
            { "                    COMPANY NAME HERE\n"},
            { "                   CUSTOM REPORT TITLE" },
            { "\n\n\n" },
            { "%37s", "Date:  " },
            { "%s\n", GetDateAsText(5) },
            { "Cable Description:  %.30s", GetWirelistInfoAsText(3) },
            { "\r" },
            { "%68s", format("Time:  %s\n", GetTimeAsText(4)) },
            { "\n\n\n" },
            { "CRC Signature:  %.12s\n", GetWirelistInfoAsText(16) },
            { "Cable Signature:  %.12s\n", GetWirelistInfoAsText(4) },
            { "\n" },
            { "%s", GetErrorText() },
            { "%c", 12 },
            { "" }          -- NOTE:  No comma needed on last text line
    }
    return theReport
end

function DoCustomReport()
    local theReport = myCustomReport()
    OutputReportToParallelPrinter(theReport)
end
```

In this example, a report is created which contains error text along with other cable information. The report is then sent out to a parallel printer.

Sample Custom Report Scripts

Autobad.rpt

Autobad.rpt prints a report that replaces the standard Test Status - Errors Only Report. You can customize the report by entering a line for your company name or whatever text you want to enter. The other information comes from the Touch 1 including an operator name if security is turned on.

```
                    [COMPANY NAME GOES HERE]
                  TEST STATUS – ERRORS ONLY

Filename:  test1.wir                            Date: 2/14/2002
Cable Description:  last learned                Time: 11:47:14
                                                Operator: Bob
CRC Signature:  M7247L
Cable Signature:  043792-6F350
Parameter Signature:

Low Voltage And High Voltage Errors:
NET 1:
   OPEN J1-005 TO J1-006
```

Autogood.rpt

Autogood.rpt prints a line for your company name or whatever text you want to enter. The other information comes from the Touch 1 including an operator name if security is turned on. The report can replace the standard *Good Only, 1 Page* auto-print report.

```
                    [COMPANY NAME GOES HERE]
                   TEST STATUS – GOOD ONLY

Filename:  untitled.wir                         Date: 2/14/2002
Cable Description:  last learned                Time: 11:53:43
                                                Operator: Joe
CRC Signature:  M7247L
Cable Signature:  043792-6F350
Parameter Signature:
Adapter Signature (s)::       Adapter Description:
       J1  D5071

Parameter Settings:
CONNECTION RESIS 5.0 ohm
LV INSULATION RESIS 100 K ohm
HIPOT VOLTAGE    OFF

Connections:
1    J1-005   J1-006

Notes:
```

Sample Custom Report Scripts *continued*

Errors.rpt

Errors.rpt prompts for a cable serial number. The other information comes from the Touch 1 including an operator name if security is turned on. The report can replace the standard error reports printed from the test windows.

```
                     [COMPANY NAME GOES HERE]
                           ERROR REPORT

Filename:  test3.wir                                Date: 2/14/2002
                                                    Time: 9:35:33
Cable Description:  my description                  Operator: Jane
Cable Serial Number:  99993333

CRC Signature:  M7247L

Cable Signature:  043792-6F350
Error Signature:  F3B493-6F350

 NET 1:
      OPEN J1-005 TO J1-006
      MISWIRE J1-027 J1-028
```

Wirelist.rpt

Wirelst.rpt prompts for a four-character work shift code. The other information comes from the Touch 1 including an operator name if security is turned on. The report can replace the standard wirelist printed from the View/Change Wirelist window.

```
                     [COMPANY NAME GOES HERE]
                  TOUCH 1 CABLE DOCUMENTATION


Filename:  test6.wir                                 Date: 2/14/2002
Cable Description:  Last Learned                     Time: 8:34:34
                                                     Operator: Bill
                                                     Shift: AM


 CRC Signature:  M7247L
 Cable Signature:  043792-6F350
 Parameter Signature:
 Adapter Signature(s):           Adapter Description:
 J1    D507F1

 Parameter Settings:
 CONNECTION RESIS 5.0 ohm
 LV INSULATION RESIS   100 K ohm
 HIPOT VOLTAGE    OFF

 Connections:
 1     J1-005    J1-006
 Notes:
```

Sample Custom Report Scripts *continued*

Testsum.rpt

Testsum.rpt prompts for a company name, a four-character work shift code, and a cable serial number. The other information comes from the Touch 1 including an operator name if security is turned on. The report can replace the standard Test Summary report printed from the Test Summary test windows.

```
                 [COMPANY NAME GOES HERE]
                   TEST SUMMARY REPORT

Filename:  test35.wir                              Date:
Cable Description:  description                    2/14/2002
Cable Serial Number:  423                          Time: 8:45:02
                                                   Operator:
                                                   Shift: SHF1



CRC Signature:  M7247L
Cable Signature:  043792-6F350
Parameter Signature:
Adapter Signature(s):           Adapter Description:
J1   D507F1

Parameter Settings:
CONNECTION RESIS 5.0 ohm
LV INSULATION RESIS  100 K ohm
HIPOT VOLTAGE    OFF

Connections:
1        J1-005    J1-006

Components:

GOOD CABLES TESTED:   5
BAD CABLES TESTED:   1
INTERMITTANT CABLES TESTED:   0
TOTAL CABLES TESTED:   6

Notes:
```

Create Labels

Before you create a label, verify that the scripting option is enabled in your 1100 or Touch1 tester (see pgs 4-5). To create and print custom labels, you will need to use a compatible label design software program that has print-to-file capability and a compatible label printer. You will also need to install the conversion utility provided by Cirris that converts the label into a script file.

The label design program that has been verified to work is the Zebra Designer software by Zebra Corp. The Zebra Designer software provides a free download and can only be used with Zebra Label printers. The TLP2844z Zebra Label printer is guaranteed to work; however other Zebra printers may also be compatible. The Zebra Designer software allows you to print custom labels with graphics and barcodes.

Download the Software

To download the Zebra Designer label design program from the Zebra website, click on the link http://www.zebra.com or type the URL in your web browser. The name of the software download is **ZebraDesigner Label Design Software**.

Design a Label

Open your label design software to design a label. Cirris recommends using the default font in the label design software to ensure usability. If you have questions, use the help system provided with the software.

To make designing labels more efficient, Cirris has made it possible for you to use "tokens" which represent run-time data you may want on a label. Cirris has provided 5 tokens for you to choose from. Each token you enter will be replaced by actual data when the label is printed. When you enter a token, it must begin and end with matching sentinels (one or more characters) for the program to recognize it. The default sentinel is the pound (#) sign. Any character or combination of characters can be used, but once you select a sentinel string for a label, you must use that same string for each token in that label. Here are some examples of what a token might look like: **#Serial#, $!Status$!, ~%*Time~%*.**

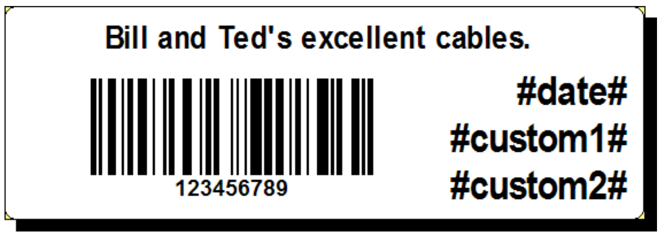
The following is a list of the available tokens and the data that will replace each token at run-time.

- The **Date** token will be replaced by the print-date of the label.
- The **Time** token will be replaced by the print-time of the label.
- The **Status** token will be replaced by either PASSED or FAILED to indicate the results of the test.
- The **Serial** token will be replaced by the serial number of the device-under-test.
- The **User** token will be replaced by name of the user performing the test.

You can also create multiple custom tokens

- The **Custom1, Custom2, Custom3,** etc. tokens will be replaced by the custom information you enter in the script file that you will generate on the following page (see an example of a Custom function on page 128).

Here is an example of what a label might look like before it's printed:

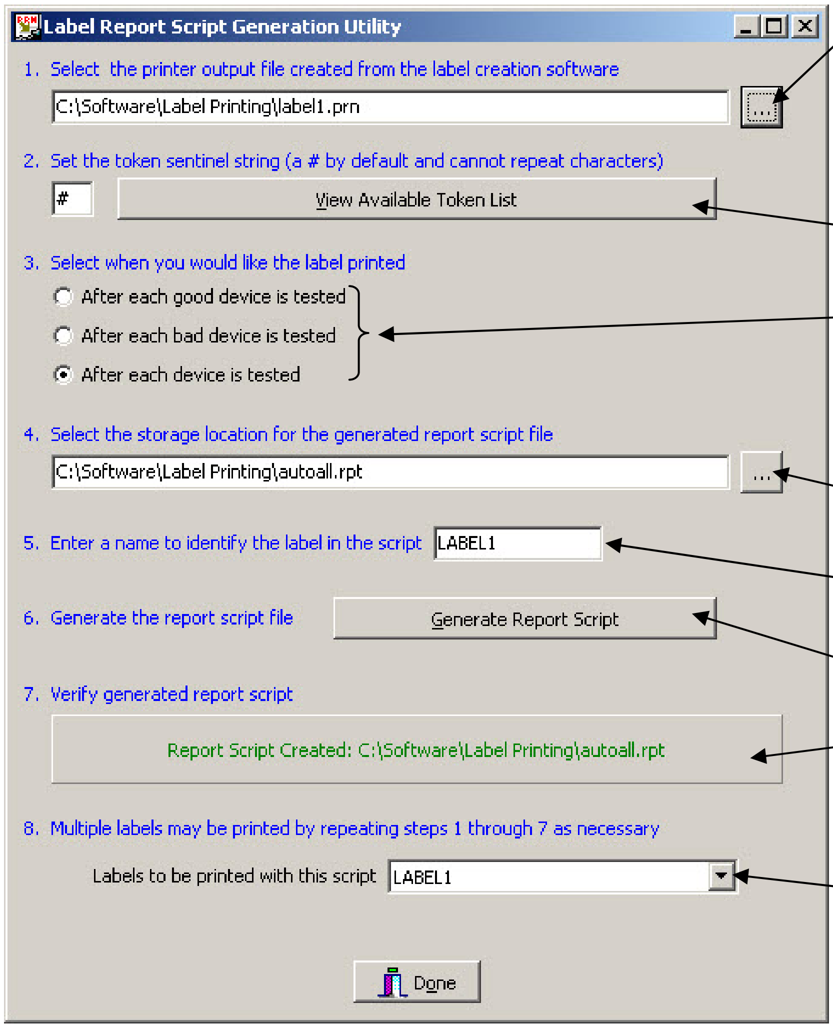


Save a Label

The label software program has the option to generate a file containing printer-specific control data (see PrintLabel on page 128). This file can be sent to the printer to create a custom label. When your label is complete, go to the File menu of the label design software you are using and click **Print.** On the next screen, check the box next to "Print to File" and click **Print.** Navigate to an accessible location, enter a filename (make sure the file extention is **.prn**), then click **Save.**

Install the Label Report Script Generation Utility

The Label Report Script Generation Utility install can be found on the CTLWIN CD that came with your tester or on the Cirris website at http://www.cirris.com/software/softdownload.html. Once installed, open the Label Report Script Generation Utility and complete the following steps.



Label Report Script Generation Utility

1. Select the printer output file created from the label creation software

C:\Software\Label Printing\label1.prn

2. Set the token sentinel string (a # by default and cannot repeat characters)

#

View Available Token List

3. Select when you would like the label printed

○ After each good device is tested

○ After each bad device is tested

[✓ After each device is tested]

4. Select the storage location for the generated report script file

C:\Software\Label Printing\autoall.rpt

5. Enter a name to identify the label in the script LABEL1

6. Generate the report script file

7. Verify generated report script

Report Script Created: C:\Software\Label Printing\autoall.rpt

8. Multiple labels may be printed by repeating steps 1 through 7 as necessary

Labels to be printed with this script LABEL1

1. Click here to navigate to the location where you previously saved your label.

2. a) Enter the sentinel(s) that you created when designing your label if you used a symbol other than pound.
 b) Click here to verify your sentinel(s) and view the token list.

3. The option you choose here will be represented in the file path as **autogood, autobad,** or **autoall** (only choose either of the first two options if you have a Touch1).

4. Click here to save the script file to an accessible location.

5. Enter a custom name or use the default name.

6. Click here to generate the script file.

7. Verify the file path. If the script file generates correctly, the text here will be green. If there is an error, the text will be red.

8. If you entered multiple labels, you can click this drop down arrow to view the labels that will print. After the label(s) has been entered, click **Done.**

Copy a Label to the Tester

Verify that the label printer is connected to the tester and that scripting is enabled.

For 1100 Testers:

1. Open the CTLWIN software that came with your tester (verify that the tester is connected to your PC).
2. At the top of the CTLWIN screen, click the drop down arrow and select **All Files (*.*)**.
3. Click **Browse** and navigate to the location of your label script file.
4. At the bottom of the CTLWIN screen, click OPEN to copy the label script file to the tester.
5. From the main menu of your 1100 tester, press **Set Preferences.**
6. From the Set Preferences menu, scroll down, make sure **Auto Print** is ON, and press **Auto.**
7. From the Select Auto Report menu, press the up or down arrow to scroll to the desired file (autogood.rpt, autobad.rpt, or autoall.rpt), and press **Accept.**

Go back to the main menu to set up a new test program or test the last learned cable.

Note: An autogood label will only print if a cable tests “good;” an autobad label will only print if a cable tests “bad;” and an autoall label will print whether a cable tests good or bad.

For Touch1 Testers:

1. From the main menu of your Touch1, select **System Setup**.
2. Toward the bottom left of the SYSTEM SETUP screen, select **Reports.**
3. At the bottom of the STANDARD/CUSTOM REPORTS screen, make sure **Auto On** is selected. Press the checkbox below the **Good Only, 1 Page** if you indicated the label to print after each good device is tested; press the **Bad Only, 1 Page** checkbox if you selected the label to print after each bad device is tested.
4. Press the **Standard Parallel Port** box next to the checkbox you selected.
5. Scroll to highlight the drive where your file is located and press **Select.**
6. Double check that the correct file is displayed and press **OK.** Press **OK** again, and then press the Home icon in the top left corner to get back to the main screen.
7. Create a test.

Print a Label

Verfiy that a compatible label printer is connected to your 1100 or Touch1 tester. After you have correctly completed the process of creating a label and have copied it to the tester, your label will automatically print when you test a cable or create a test. If your label(s) fails to print, double check that everthing is plugged in and go back through the steps; if needed call Cirris for assistance.

Chapter 7: Creating & Editing Script Files

Overview

There are three ways to create or edit a script files. They are:

- Replace default text with custom text in any of the default script files.
- Modify an existing script file by adding or deleting existing functions.
- Create a script file by using the predefined functions along with functions you create.

Rules for File Management of Script Files

- Make a copy of the script file before editing so you will have a "known good" copy for reference.
- Create or edit script files using a text editor on a computer other than the tester. The editor must not word wrap. For example, in Notepad, from the Edit menu selection, make sure there is no check by Word Wrap. If you are using a word processor, do not save the script as a Word or a WordPerfect file. Use "Save As" with "Text Only" as the output format and do not use *.txt* the regular extension. Use the extension to indicate the script type: *.evt* or *.lua* for a Test Event Script, *.cmp* for a Custom Component Script, and *.rpt* for a Custom Report Script.
- Use the standard DOS file naming convention: eight characters or less followed by the extension designated for each kind of script.
- Because you can only attach one Test Event Script and one Custom Component Script to a wirelist, enter all the functions you want active in one test session in those scripts. Custom Report Scripts are global to the system and are not attached to a specific wirelist.

Script File Contents

- Declared global variables
- Required function specific to script type:
  – EVT Test Event Script = `function DoIt` (page 20)
  – LUA Test Event Script = `function DoOnTestEvent` (page 14)
  – Custom Component Script = `function cmp<componentName>.test` (page 28)
  – Custom Report Script = `function DoCustomReport` (page 36)
- Functions that gather information, perform an operation, or output information
- Comment blocks and/or individual comments

Note: See Chapter 9 for syntax rules and Chapter 10 for a list of functions.

Test Event Script Example (*Serlabel.lua)*

This is an example of the test event script file `serlabel.lua.` Explanations are given on the different sections and how some of the functions are used together to print a label.

```
----------------------------------------------------------
-- NOTE: IN THIS FILE TWO DASHES, --, INDICATE A COMMENT
-- SO TEXT FOLLOWING THEM IS IGNORED.

-- GENERAL DESCRIPTION:
--    This file contains an example for printing a label
--    out to the serial printer.  The label includes the
--    the cable serial number which will be automatically
--    updated.
--
--    At the start of each test run:
--     Ask the operator to confirm the first serial
--     for this run.  The serial number is pulled from
--     disk where it is stored from the last series of
--     tests.
--
--    At the end of each GOOD cable tested:
--     Print the label on the serial printer.
--
--     Update the serial number and record it to disk so
--     the next day the serial numbers pick up where they
--     left off.
--
--     The label has the following format:
--          COMPANY NAME        date
--          Cable Serial #:
--          (3 linefeeds to skip to the next label)
----------------------------------------------------------
-- Global user-defined variable declarations which must be
-- set equal to something if listed here.

gLongTermCableSerialNumber = 0

----------------------------------------------------------
--    THIS NEXT SECTION IS WHERE YOU CREATE YOUR REPORTS
--    Each line must have curly brackets {} and not
--    parentheses ().
--    Each line must end with a comma.
--    Each line must have at least one section in quotes.
--
--    LabelReport() creates the label & returns the text
--    in an array that will be formatted and sent to the
--    printer.
--    LabelReport() is called by DoOnTestEvent() with
--    option = 3 at the end of each test when the
--    cable tests good.
```

Comment Block
Note: Precede all comments with a double dash,`--`

Global Variable

(Continued on next page)

```
----------------------------------------------------------
function LabelReport()
 local theReport =
 {
  {"COMPANY NAME        %s\r", GetDateAsText(5) },
  {"Cable Serial #:  %I\r", gLongTermCableSerialNumber },
  {"\r\r\r" }
 } return theReport
end



----------------------------------------------------------
--    THIS NEXT SECTION CONTAINS
--    FUNCTIONS THAT MAKE THE SCRIPT RUN
--    If you need to make changes to this section, see
--    the documentation.  You can do some serious
--    customization if you want to edit these functions.
--    We suggest you edit a COPY of the script so you can
--    have a "known good" reference later.
----------------------------------------------------------
function DoOnTestEvent(option)
 --   Called by Touch 1 with the following options:
 --   option = 1: Before testing any cables, start of run
 --   option = 2: LV Test Starts (press start button or
 --               cable attached)
 --   option = 3: End of all tests (continuous: cable
 --        removed, single: cable removed or end of
 --        required tests (LV [and hipot])

if option == 1 then

 -- Get the long term cable serial number from disk and
 -- have user confirm or update it.  Note the number on
 -- the disk is the number from the last good cable.
 -- We need to add 1 to it before asking user if it is
 -- the next cable serial number.

  GetTheLastLongTermCableSerialNumber()
  local temp = gLongTermCableSerialNumber + 1
  local tempstr = tostring(temp)

  local responseString = PromptForUserInformation(1,
    "ENTER STARTING", "CABLE SERIAL NUMBER", 30, tempstr)

  if responseString ~= nil then
    temp - tonumber (responseString)
    if tem > 0 then
      temp = temp –1
    end
    gLongTermCableSerialNumber = temp
  else
    gLongTermCableSerialNumber = temp -1
  end
```

See Chapter 8: *Function Definitions*

See the *format* function for an explanation.

You can replace this with your company name.

A **required** function in a test event script

Start of run

Prompt for the cable serial number on the touch screen

(Continued on next page)

(*serlabel.lua*, continued)

```
  elseif option == 3 then
    local iCableStatus = GetCableStatus ()
    if iCableStatus == 0 then
      IncrementAndSaveLongTermGoodCableCount ()
      local theReport = LabelReport ()
      OutputReportToSErialPrinter(theReport)
    end
  end
end

function OutputReportToSerialPrinter(theReport)
 -- This function sets up the COM port for the
 -- serial printer.  It walks through each item in
 -- theReport, and formats using the formatting
 -- instructions in the first element of each item (first
 -- part in quotes contains formatting)

  SetSerialParams("9600:8n1")

  iIndex = 1
   while theReport[iIndex] ~= nil do
    local theText = call( format, theReport[iIndex])
    WriteToSerial(theText)
    iIndex = iIndex + 1
  end
end

function IncrementAndSaveLongTermGoodCableCount()
 -- update the long-term "good cable" counter
 gLongTermCableSerialNumber = gLongTermCableSerialNumber
                                      + 1
 writeto("c:\\touch1\\storage.dat")
 local tempstr = tostring(gLongTermCableSerialNumber)
 write(tempstr)
 writeto()
end

function GetTheLastLongTermCableSerialNumber()
 -- read cable serial number from disk
 local foundFile
 local errMsg
 foundFile, errMsg = readfrom("c:\\touch1\\storage.dat")
 if foundFile ~= nil then
  local tempstr = read()
  gLongTermCableSerialNumber = tonumber(tempstr)
  else
    gLongTermCableSerialNumber = 0
    local tempstr = format("Error opening the storage.dat
                           file\n %s", errMsg)
    MessageBox(tempstr)
  end
  readfrom()
end
```

Ready the serial printer for printing

Output the label.

User-created function

Close the file

Read the cable serial number that was saved in the file.

Close the file

Changing Generic Text to Custom Text

Using your text editor, scroll or search for the text you wish to replace. Once the text is located, overwrite it with your custom text. For example, in the sample script file, *badrpt.lua*, the function `BadCableReport` contains the generic placeholder, "COMPANY NAME" you can replace with your own company name or other text.

```
function BadCableReport()
-- sets up the text for the "bad cable" report
 local theReport =
 {
   { "                                    BAD CABLE REPORT" },
   { "\r" },
   { "                                    _________________\n" },
   { "\n\n\n" },
   { "%64s", "COMPANY NAME\n" },
   { "%60s", "Date:  " },
   { "%s\n", GetDateAsText(5) },
   { "%60s", "Time:  " },
   { "%s\n", GetTimeAsText(4) },
   { "%65s", format("Operator:  %.30s\n", operator) },
   { "\n\n" },
   { "Cable Signature:  %.12s\n", cableSignature },
   { "Cable Description:  %.30s\n", cableDescription },
   { "\n\n" },
   { "%s\n" , GetErrorText() },
   { "\n\n\n\n" },
   { "" }
 }
 return theReport
end
```

Replace **COMPANY NAME** with your own company name.

Chapter 8: Debugging & Trouble-shooting

Message box "The script <scriptname> has changed" keeps displaying

This message can appear for EVT & LUA Test Event Scripts and Custom Component Scripts after making editing changes.

Touch1:

To remove the message box telling you the script has changed, press Cancel in the message box. At the Main Menu, press Test Setup. In Test Setup, press View & Change Wirelist and then press OK. If the changes to the script are correct, save the corrected script on the Touch1.

1100's:

To remove the message box telling you the script has changed, press Cancel in the message box. At the Main Menu, press the down button and then Do File Management. In Do File Management, press Save Current Test to save the wirelist.

Script file does not run

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

- Test Event or Custom Component Script file is not attached to the wirelist or Custom Report Script is not enabled at the system level.
- Script file contains coding errors.

Examples:

- Scripting is not turned enabled in the wirelist you used to test your cables. See Attach Script File(s), page 6.
- The wrong script file is selected in the wirelist used to test your cables. Change the selected script file to the correct one.
- The script file contains syntax errors, which need to be corrected. See Debugging methods, page 49.

Script Report does not print

In this scenario, you have finished testing cables, but the script report did not print.

- Scripting is not turned on at the system level for the custom report or for the wirelist used to test your cables. Custom Reports are enabled at the system level and test event script reports are individually "attached" to each wirelist.
- The wrong script file is selected in the wirelist used to test your cables. Edit the current wirelist and change the selected script file to the correct one.
- 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.

Duplicate fields in SPC data

In this scenario, you have finished writing the SPC data, but upon examining it, you find that you have multiple fields containing the same data.

- You have created a user-defined field that duplicates one of the predefined fields for SPC data. In your script file, remove the call to `SaveSPCData`() that saves the data twice.

Custom button is disabled in the Add Components screen (Touch1)

In this scenario, you want to add a scriptable custom component but the Custom button is disabled.

- The Scripting Feature has not been enabled on this tester (page 5).
- Your security level does not allow this function (page 11).
- A Custom Component Script has not been attached to the wirelist (page 6).

Debugging methods

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

How to quickly test changes to a script file on a Touch1

1. To quickly debug script files, put the script on your network or on a floppy disk. If your script is on the Touch 1's internal hard disk, you can't update it quickly.
2. Attach the script to the wirelist from the network or floppy location. After you finish, you will need to change it back to the original location.
3. Each time you want to change the script, take out the floppy disk, pop it into your computer, and edit the script.
4. Save the script to the floppy disk (as plain text) and put it back into the Touch 1.
5. On the Touch 1, go into the "View & Change Wirelist" window and press Cancel. This forces the tester to reload the wirelist and script from the floppy disk or network location. At this point, it will instantly check the script file's syntax and report any errors.
6. If the script does not have any syntax errors, you can then execute the script to check its operation.
7. If the script still has errors, repeat the steps by editing the script and saving it to the floppy.

When the script file works as intended, copy the script to its original location (either the network or the Touch 1's hard drive) and change the location when you enable the script with the wirelist.

Syntax error(s)

- To provide more information about errors when creating and compiling a script, insert the `$debug` command at the top of the script file. When an error occurs in a script compiled with this option, it displays the line where the error occurred.
- Comment out or delete sections of code until you get the script to run without errors then put the sections back in until the script no longer runs to find the error.

Break a line of your program into several lines without changing the task of the function. For instance, you may break code into small functions then have a function call those small functions. Putting too many commands into a single line and/or function can make the script difficult to debug.

Script runs but not as intended

- Use a message box to display values and types of variables.
- Use the **outtextxy()** (Touch1 only) and **print()** functions to display messages on the screen to let you know what part of your script is being executed. This is extremely helpful if your code is crashing or hanging somewhere. By using these functions in various places in your script, you can quickly see which parts run and in what order. Use the **outtextxy()** function if you do not want text to scroll off the screen and if you want information to be in a fixed place on the screen. Otherwise, use the **print()** function. A helpful technique is to put spaces after whatever you display so you don't confuse old values with new ones.

  For example on a Touch1, do the following:

```
outtextxy(9, 8,"                          ")
outtextxy(9, 9," ************************ ")
outtextxy(9,10," *                      * ")
outtextxy(9,11," *    YOUR MESSAGE      * ")
outtextxy(9,12," *                      * ")
outtextxy(9,13," ************************ ")
outtextxy(9,14,"                          ")
```

Chapter 9: LUA Syntax

Overview

This section describes the syntax for the LUA programming language. This syntax must be followed when writing scripts for the tester.

Reserved Words

LUA is a case-sensitive language. Names can be any combination of letters, digits, and underscores but they cannot begin with a digit. The following words are reserved for the LUA language and cannot be used as variable names in your script.

and	end	nil	return
do	function	not	then
else	if	or	until
elseif	local	repeat	while

Function Definitions

To call a function in LUA, you must use the reserved words: *function* and *end* along with a function name. Function names are case sensitive, cannot be numeric, cannot start with a numeric, and they must be spelled correctly. Results from the function are returned using the reserved word *return*. If the function has no return, the function returns with no results.

Examples:

-
  ```
  function DoOnTestEvent(option)
      if option = = 1 then
          MessageBox("Starting the Low Voltage Test")
      end
  end
  ```

-
  ```
  function testInputs()
      local externalStartSwitchState, HipotSafetySwitch = ReadUserInputStates(1, 2)
      return externalStartSwitchState, HipotSafetySwitch
  end
  ```

Functions

- In this manual, optional input parameters are enclosed in brackets, `[ ]`.
- Input and return variables can initially be named anything but these names then should be used throughout the function.
- All input variables must be defined before calling the function.
- The order of the inputs is important and must be followed.
- All input and output functions in LUA use two file handles, _INPUT and _OUTPUT, which are stored as global variables. Initially, _INPUT = _STDIN and _OUTPUT = _STDOUT. The 1100 testers do not support STDIN and STDOUT.
- A nil is returned on failure for all I/O functions unless otherwise stated. Only one file can be used for output and one for input at any given time.

Comments

- Denote comments by two dashes - -. All text following a comment is ignored until the end of the line. Comments can start anywhere outside of a string.

Numbers

- Numbers can be written with an optional decimal part or exponent. Use the ^ character to raise something to an exponential power.
  Examples: `1, 6.001, 5.9e-4, 0.29e5, R2 = R1e^(B/T2 – B/T1)`
- For trigonometric functions, angles are in degrees and not radians.

Strings

- Strings can contain C-like escape sequences such as `'\n', '\t', and '\r'`.
- To display a quote in a text string, use \".
- When indexing a string, the first character is at position 1, not 0, as in the C language.
- Use "**..**" to concatenate strings and/or variables together for message text.

Statements

- The LUA language allows multiple assignments with the list of variables on the left side separated by commas and the list of expressions on the right side separated by commas.
  For example: myPartNum, myRevisionNum = 34, 2
- A return statement is used to return a value or values from a function. It is the last statement in a list of statements. This avoids unreachable code.
- Local variables may be declared anywhere in a list of statements. They can be initialized when they are declared; otherwise, they are set to nil. The scope for local variables begins after they are declared. It ends with the last statement in the list. Global variables do not need to be declared. **Any variable is assumed global unless it is declared local. Global variables stay around until the tester is turned off.**
- The syntax for while, if, else, ifelse, and repeat statements have required keywords. The following examples have the keywords highlighted.

while example:

```
while theReport[iIndex] ~= nil do
    local theText = call(format, theReport[iIndex])
    SendTextToParallelPrinter(theText)
    iIndex = iIndex + 1
end
```

if else example:

```
if (iSPCDataOn == 1) and (option == 1) then
    SetCableSerialNumber(cableserialnumber)
else
    MessageBox("SPC Data Collection\nis NOT turned ON")
end
```

elseif example:

```
if option == 1 or option == 2 then
    GetTheLastCableSerialNumber()
elseif option == 3 then
    MessageBox("Test has ended!")
end
```

Statements, continued

repeat example:

```
repeat
    local theText = call(format, theReport[iIndex])
    SendTextToParallelPrinter(theText)
    iIndex = iIndex + 1
until theReport[iIndex] == nil
```

Expressions

- Examples of simple expressions are:
  Option = 1
  MyCableDescription = “Green Cable”
  iIndex = iFirstNumber
- The LUA language can perform arithmetic operations on numbers or strings that can be converted to numbers. The conversion of strings to numbers takes place automatically. The arithmetic operators are as follows:

+ (addition)	**/** (division)
- (subtraction)	**^** (exponentiation)
***** (multiplication)	**-** (negation)

- The LUA language provides a list of relational operators that return nil as false (0). The equality operator, ==, first compares the types and if they are different the result is nil. Otherwise, the values are compared. The relational operators are as follows:

< (less than)	>= (greater than or equal to)
> (greater than)	~= (not equal to)
<= (less than or equal to)	== (equal to)

Example: “10” == 10 returns false because a string is different from an integer

- The LUA language provides the three logical operators:
  and:
  This operator returns nil if its first argument is nil; otherwise, it returns its second argument.
  or:
  This operator returns its first argument if it is different from nil; otherwise, it returns its second argument.
  not:
  This operator negates the argument.
  Example: If not option == 1 then is the same as if option ~= 1 then
- The LUA language provides a string concatenation operator denoted by “..”. Numbers are automatically converted to strings.
  Examples:
  MySerialNumber = “1234” .. “5678” makes the string “12345678”.
  Homedir .. “myfile.exe” appends the file, myfile.exe, onto the path in the variable Homedir.
- Precedence from highest to lowest priority for operators is as follows:
  or and == ~ = > = < = > < .. + * (unary) not ^
- Parentheses are strongly recommended to avoid confusion when there is more than one comparison.

Example: if x > y and w < z then

In this example, the “and” of y and w is done first. The comparison between x and z is then made. Parentheses around the pairs (x > y) and (w< z) would avoid the problem.

Expressions, continued

- Expressions in lists are assigned consecutive numerical indices, starting at 1.
  Example:
  ```
  Values = {“END”, 10, “GO”}
  is equal to
  temp = {}
  temp[1] = “END”
  temp[2] = 10
  temp[3] = “GO”
  values = temp
  ```

Global Variable Space

Use global variable space to pass data back and forth between one type of script to another. For example, a component script has collected data a test event script needs. Using global variable space, the data can be changed and then passed to the other script type. The wirelist must have both scripts attached to work. There is a one megabyte limit to the amount of data that can be stored.

Scripting is split into four global environment sections: COMPONENT, EVENT, REPORT, and SYSTEM. Component scripts operate exclusively in the COMPONENT section, report scripts operate exclusively in the REPORT section, test event scripts operate exclusively in the EVENT section, and the SYSTEM sections is reserved for internal use only.

Each of the four global sections can be seen in every script. To use one of these global variables, attach your own data to the global variable data space as if it were an array name using a string. After using the data, flush the data stored in the variables by assigning them to nil. If the memory is not cleaned up, it will stay around until the tester is turned off.

For example, a component script could store data as shown below. Note all data is stored as a string. Any unused or potentially used memory should be initialized to nil.

```
components[1] = “my string for the other script”
components[2] = “1.2345”
components[3] = nil
components[4] = nil
components[5] = nil
```

Now an event script could access the component data as shown below.

```
local index = 1
while components[index] ~= nil do
  myTable[index] = components[index]
end
```

Now a copy of the component data exists in the event scripts variable, “myTable”, for processing.

Miscellaneous Prompt Box Syntax

See the functions `DialogOpen` (page 65) and `MessageBox` (page 120) for details on adding titles, adding hotkeys, adding underlining, and changing font size in prompt boxes.

Chapter 10: Script Functions

Functions Organized Alphabetically

Functions Organized by Category

Function Descriptions

abs

Explanation:

Use this function to return the absolute value of the input number.

Format:

```
iResult = abs(myInputNum)
  ↑        ↑         ↑
RESULT   FUNCTION  INPUT
```

INPUT `MyInputNum`	RESULT `iResult`
Call `abs` to find the absolute value of `myInputNum`.	Contains the absolute value for the number, `myInputNum`.

***Examples*:**

-
  ```
  myInputNum = -5
  iResult = abs(myInputNum)
  ```
  This example returns `iResult` equal to 5.

-
  ```
  iAbsValue = abs(21)
  ```
  This example returns `iAbsValue` equal to 21.

acos

Explanation:

Use this function to return the arccosine of a number. Angles are in degrees and not radians.

Format:

```
iResult = acos(myInputNum)
  ↑         ↑       ↑
RESULT   FUNCTION  INPUT
```

INPUT `MyInputNum`	RESULT `iResult`
Call `acos` to find the arccosine of `myInputNum`.	Contains the arccosine value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = .5
  iResult = acos(myInputNum)
  ```
  The function will return `iResult` equal to 67.

-
  ```
  iArcCosine = acos(.8)
  ```
  The function will return `iArcCosine` equal to 41.

appendto

Explanation:

Use this function to open an existing file so you can write something in it. It **will not** erase the contents of the file. This function **does not** close the current output file when done so use the function `writeto` to close the file.

Note: 1100 testers do not support this function. To do this on 1100 testers: write the file to a string, append to the string and write the string back out to the file.

Format:

```
myFile, sErrMsg = appendto(sFilename)
   ↑                   ↑         ↑
 RESULTS            FUNCTION   INPUT
```

INPUT `sFilename`	RESULT1 `Myfile`	RESULT2 `sErrMsg`
A string containing the name of an existing file that will be opened for writing.	Contains the filename's handle **OR** nil = failed opening the file	A string describing the error if the function fails.

Example:

-
  ```
  serialNumFile, sErrMsg = appendto("/DOS/touch1/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.

ascii

Explanation:

Use this function to get the ASCII code of a character.

Format:

```
sASCIICodeText = ascii(sText [, iIndexOfChar])
      ↑              ↑          ↑
  RESULT          FUNCTION    INPUTS
```

INPUT1
`sText`
	INPUT2
	`iIndexOfChar`
	`(optional)`
		RESULT
		`sASCIICodeText`

Character or string you want the ASCII code on.	Integer containing the index of the incoming character. If not used, the function will return the ASCII code of the first character. **Note:** When indexing a string, the first character is at position 1, not 0, as in C.	Text containing the ASCII code of the character.

Examples:

- `sASCIICodeText  = ascii("ABCDEFG", 4)`

  This example returns `the sASCIICodeText = "68"` because the ASCII code of the fourth character, 'D', is 68.

- `sASCIICodeText = ascii("BAD")`

  This example returns `sASCIICodeText = "66"` because the ASCII code of the first character, 'B' is 66.

asin

Explanation:

Use this function to return the arcsine of a number. Angles are in degrees and not radians.

Format:

```
myValue = asin(myInputNum)
   ↑         ↑       ↑
RESULT   FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `myValue`
The number you want the arcsine on.	Contains the arcsine value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = .15
  myValue = asin(myInputNum)
  ```
  The function will return `myValue` equal to 9.585.

-
  ```
  myValue = asin(.1)
  ```
  The function will return `myValue` equal to 6.377.

atan

Explanation:

Use this function to return the arctangent of a number. Angles are in degrees and not radians.

Format:

```
myValue = atan(myInputNum)
   ↑         ↑       ↑
RESULT   FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT
The number you want the arctangent on.	Contains the arctangent value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = .7
  myValue = atan(myInputNum)
  ```
  The function will return `myValue` equal to 38.880.

-
  ```
  myArcTan = atan(.4)
  ```
  The function will return `myArcTan` equal to 24.22.

atan2

Explanation:

Use this function to return the arctangent of `myY/myX`. The function uses the signs of both arguments to determine the quadrant of the return value. Angles are in degrees and not radians.

Format:

```
myValue = atan2(myY, myX)
   ↑         ↑       ↑
RESULT   FUNCTION  INPUTS
```

INPUT1 `myY`	INPUT2 `myX`	RESULT `myValue`
Numerator of the fraction `myY/myX` the arctangent will be computed on.	Denominator of the fraction `myY/myX` the arctangent will be computed on.	Contains the arctangent value of the inputs, `myY/myX`.

Example:

```
myValue = atan2(.3,7)
```

The function will return `myValue` equal to 2.454.

call

Explanation:

Use this function to call a function with a table of arguments.

Format:

```
call(myFunction, args)
  ↑               ↑
FUNCTION        INPUTS
```

INPUT	DESCRIPTION
myFunction	The name of the function that will be called.
args	Arguments of the function to be called.

Example:

-
  ```
  iIndex = 1
  while myReport[iIndex] ~= nil do
     local sFormattedText = call(format, {myReport[iIndex]})
     SendTextToParallelPrinter(sFormattedText, 1)
     iIndex = iIndex + 1
  end
  ```

  This example calls the format function to format the array of report text and returns the formatted text in `sFormattedText`. It then sends the formatted text to the printer.

ceil

Explanation:

The ceiling function returns the smallest integer not less than the input number. The resulting number will round upward. Use this function to round up real numbers to an integer value for input to other functions. See the function floor() for rounding downward.

Format:

```
iResult = ceil(myInputNum)
   ↑         ↑        ↑
 RESULT   FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `iResult`
Contains the number the `ceil` function will be computed on.	Contains the smallest integer value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = 19.6
  iResult = ceil(myInputNum)
  ```
  The function will return `iResult` equal to 20.

-
  ```
  iResult = ceil(5.8)
  ```
  The function will return `iResult` equal to 6.

cleardirectory

Note: 1100 testers do not support this function

Explanation:

Use this function to a delete a directory if it exists at the current location. If the directory does not exist, the directory will be created. To delete subdirectories or directories in different locations include the directory path.

Format:

```
cleardirectory(sDirectory)
     ↑              ↑
  FUNCTION        INPUT
```

INPUT `sDirectory`	RESULT
String containing an optional path and the directory to be deleted or created.	If the directory contained in `sDirectory` exists, it is deleted. If the directory does not exist, it is created.

Example:

-
  ```
  sTempDirectory = “TempDir”
  if iCopiedOk then
     cleardirectory(sTempDirectory)
  ```
  This example deletes the temporary directory, “TempDir”, if the variable iCopiedOk was set.

copyfile

Explanation:

Use this function to copy a file. The function will use the current directory if no path is given.

Format:

```
sErrorText = copyfile(sSourceFileName, sDestFileName,
                      [iProgressMeterId])
  ↑                 ↑                   ↑
RESULT          FUNCTION              INPUTS
```

INPUT1 `sSourceFileName`	INPUT2 `sDestFileName`	INPUT3 **(optional)** `iProgressMeterId`	RESULT `sErrorText`
String containing an optional path and the name of the file to be copied. If the path is not included, the file must be in the current directory.	String containing an optional path and the name of the destination file. If a path is not included, the file will be copied to the current directory.	Integer containing a Touch 1 screen's widget id where the progress meter will be located.	String containing one of the following: • nil = no error • Could not open: "filename" • Corrupted file: "filename" • Copy Aborted **(1100 only)**

Example:

-
  ```
  if iBackupNeeded then
     copyfile("test1.wir", "a:\test1.wir")
  ```

  In this example, the wirelist file, `test1.wir` will be copied to a floppy disk if the variable iBackupNeeded was set.

cos

Explanation:

Use this function to return the cosine of a number. Angles are in degrees and not radians.

Format:

```
myValue = cos(myInputNum)
  ↑         ↑      ↑
RESULT   FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `myValue`
Call `cos` to find the cosine of `myInputNum`.	Contains the cosine value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = 8
  myValue = cos(myInputNum)
  ```

  The function will return `myValue` equal to .992.

-
  ```
  myCosine = cos(13)
  ```

  The function will return `myCosine` equal to .974.

date

Explanation:

Use this function to get the date and time. There are several different format codes available and these codes can be combined. If no format is used for format, the function will return a generic date and time representation.

Note: See the `GetDateAsText` function for another way to get the date.

Format:

```
date([sFormatCode])
  ↑       ↑
FUNCTION  INPUT
```

INPUT `sFormatCode`	RESULT
`%a`	Abbreviated weekday name
`%A`	Full weekday name
`%b`	Abbreviated month name
`%B`	Full month name

`%c`
`OR`
`No Input`
	Generic date and time representation. This is the same format returned when no input is used.

`%d`	Day of the month as a decimal number (01-31)
`%H`	Hour (24-hour clock) as a decimal number (00-23)
`%I`	Hour (12-hour clock) as a decimal number (01-12)
`%j`	Day of the year as a decimal number (001-366)
`%m`	Month as a decimal number (01-12)
`%M`	Minute as a decimal number (00-59)
`%p`	AM or PM
`%S`	Second as a decimal number (00-59)
`%U`	Week number of the year as a decimal number (00-52) where Sunday is the first day of the week
`%w`	Weekday as a decimal number (0-6) where 0 is Sunday
`%W`	Week number of the year as a decimal number (00-52) where Monday is the first day of the week
`%x`	Generic date representation
`%X`	Generic time representation
`%y`	Year without century as a decimal number (00-99)
`%Y`	Year with century as a decimal number (four digits)
`%%`	Character, %

Examples:

- `myDate = date("%A %B %d, %Y")`

  This example returns the string "Monday, July 24, 1998" in `myDate`.

- `currentDate = date()`

  This example returns the string "Wed Jul 22 11:53:25 1998" in `currentDate`.

delay

Explanation:

Use this function to set a delay time in seconds. If the function has no input it will return a description of itself. Use this function for controlling serial printers that need a delay time or 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.

Format:

```
  Delay(fDelayInSeconds)
    ↑        ↑
FUNCTION   INPUT
```

INPUT `fDelayInSeconds`	RESULT
A float corresponding to the number of seconds of delay time.	Delay for the number of seconds equal to `iDelayInSeconds`.

Examples:

-
  ```
  iDelayInSeconds = 5.0
  Delay(iDelayInSeconds)
  ```
  The function will delay all functioning on the Touch 1 for 5 seconds.

- `Delay(.1)`

  The function will delay all functioning on the Touch 1 for 100 milliseconds.

DialogCheckBtn

Explanation:

Use this function to check if any button has been pressed in the displayed dialog box. The dialog box is created using the function `DialogOpen` and closed using the function `DialogClose.` 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 these dialog boxes. You can check digital I/O or perform test functions while waiting for the user to press and release a button.

Format:

```
iNumPressedButton = DialogCheckBtn(iDialogNumber)
      ↑                      ↑              ↑
    RESULT               FUNCTION         INPUT
```

INPUT `iDialogNumber`	RESULT `iNumPressedButton`
An integer containing the identification number for the dialog box. This number is created by the DialogOpen function.	An integer containing the number of the pressed button **OR** 0 = no button has been pressed

Example:

See the `DialogOpen` function.

DialogClose

Explanation:

Use this function to close 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 must be used to clean up memory. For every `DialogOpen` function there should be a `DialogClose` function.

Format:

```
DialogClose(iDialogNumber)
     ↑                ↑
  FUNCTION          INPUT
```

INPUT `iDialogNumber`
An integer containing the identification number of the dialog box to close. This number is created by the `DialogOpen` function.

Example:

See the `DialogOpen` function.

DialogOpen

Explanation:

Use this function to display, on the tester, a dialog box containing custom text and up to six custom buttons. For the Touch1, the dialog box will be centered on the screen with the underlying screen showing. For 1100 testers, the dialog box is the entire screen. The box is automatically sized to fit the required custom text. The buttons automatically align but do not automatically size. The dialog box WILL NOT have a default CANCEL button if no other custom buttons are created. Up to ten dialog boxes can be opened at one time. Opening more than ten dialog boxes at one time will cause a stack overflow and the tester will have to be power cycled. The stack is not reset after an overflow and or after reloading the script. 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. The function returns a number identifying the dialog box. This number is used when manually checking for a button press.

NOTES:

- Color changes to the screen will only be seen when a monitor is attached to the Touch 1.
- The Touch 1 screen is 320 x 240 pixels where a character is 8 x 6. A button cell size in pixels is 32 x 30 which allows four characters per row & five characters per column. Buttons are three button cells wide so eleven characters fit into one button cell.

DialogOpen, *continued*

Miscellaneous Prompt Box Syntax

- **String Concatenation:** Use "**..**" to concatenate strings and/or variables together for message text.
- **Titles:** 1100 dialog and message boxes can have titles. The Touch1 will ignore the title and the 1100 will display it on the first line of the dialog or message box.

The title is embedded in the message string as follows: sMsg = "~title~message"

There are several control sequences to enhance dialog and message box displays. They are as follows:

- **Hot keys:**

You can define *ShowHotKey* and *HideHotKey* to add hot keys (keyboard shortcuts) to dialog and message boxes. The codes are as follows:

Show Hot Key = <K *x*> where *x* is the letter of the hot key **or** use the code: "\26\02\01"

Hide Hot Key = <KH *x*> where *x* is the letter of the hot key **or** use the code: "\26\02\02"

Cancel Key = \27

Enter Key = \13

Example:

```
ShowHotKey = "\26\02\01"
sMessage = "Hot Key Test"
local iBtnNum = MessageBox(sMessage, "S" .. ShowHotKey "kip","<KH \27>Cancel")
```

This example displays a message box with Skip and Cancel buttons. The "S" key is a defined hot key that activates when pressed. The ESC is not a hot key so will not activate when pressed. The ".." means concatenate the text together as one string.

- **Underlining:**

You can define *StartUnderline* and *StopUnderline* to add underlining to dialog and message boxes. The codes are as follows:

Start underline = <U> **or** "\25\01\01"

Stop underline = <\\U>**or** "\25\01\02"

Example:

```
StopUnderline = "\25\01\02"
MessageBox("No Underline" ..  "<U>Underlined" ..StopUnderline ..
           "No Underline", "CANCEL")
```

This example displays a message box with a Cancel button. The text "No Underline" is not underlined followed by the text "Underlined" which is underlined and "No Underline" is not underlined. The ".." means concatenate the text together as one string.

- **Font Changes:**

You can define fonts to display text for dialog and message boxes in different touch1 fonts. The codes are as follows:

Big Font = <B>

Big Font with multiplier = <B 2> **or** <B 3> where <B 2> is twice & <B3> is triple the size of <B>

Small Font = <S>

Small Font with multiplier = <S 2> **or** <S 3> where <S 2> is twice & <S3> is triple the size of <S>

Very Big Font = "\28\02\03\01"

Example:

```
VeryBigFontCode = "\28\02\03\01"
local iBtnNum = MessageBox("Regular Font" .. VeryBigFontCode ..
                           "VeryBig Font" .. "<S>Small Font",
                           "Cancel")
```

This example displays a message box with a Cancel button. The text "Regular Font" is in the normal Touch1 big font. The text "VeryBigFont" is in very big font and then the text "Small Font" is in the Touch1's smallest font.

DialogOpen, *continued*

Format:

```
     iDialogNumber = DialogOpen([iFromColor, iToColor,] sMessageText
                                  [, sButtonText1] [, sButtonText2]
                                  [,sButtonText3], [, sButtonText4]
                                  [, sButtonText5] [,sButtonText6])
          ↑              ↑                   ↑
        RESULT        FUNCTION            INPUTS  ( [ ] = optional )
```

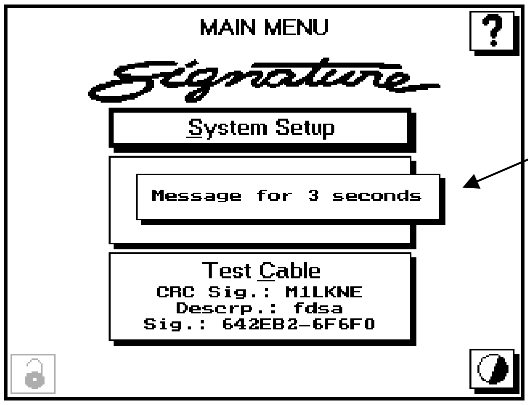
INPUTS	INPUT DESCRIPTION		RESULT `iDialogNumber`
`iFromColor,` `iToColor` **(optional pair)**	0 = BLACK 1 = BLUE 2 = GREEN 3 = CYAN 4 = RED 5 = MAGENTA 6 = BROWN 7 = LIGHTGRAY	8 = DARKGRAY 9 = LIGHTBLUE 10= LIGHTGREEN 11 = LIGHTCYAN 12 = LIGHTRED 13 = LIGHTMAGENTA 14 = YELLOW 15 = WHITE	An integer identifying the dialog box just opened. Use this number as an input for the functions `DialogCheckBtn` and `DialogClose`.
`sMessageText`	String containing text for the dialog box.		
`sButtonText1` – `sButtonText6` **(optional)**	String containing text to put in the custom buttons 1-6.		

DialogOpen, *continued*

Examples:

Note: For 1100 testers, the message box is the entire screen.

-
  ```
  local idNum = DialogOpen("Message for 3 seconds")
  Delay(3)
  DialogClose(idNum)
  ```

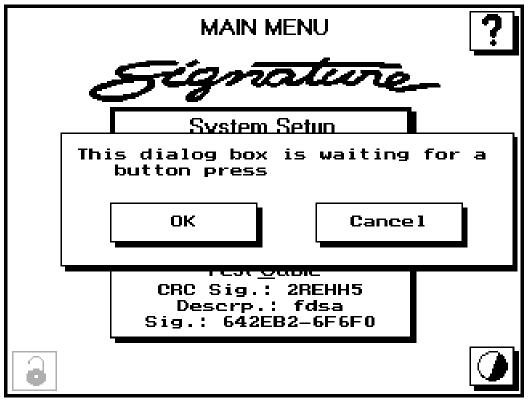


Message for 3 seconds

CRC Sig.: M1LKNE
Descrp.: fdsa
Sig.: 642EB2-6F6F0

This message will display for three seconds.

```
myColors = {{0,10},{15,14}}
local idNum = DialogOpen(myColors, "This dialog box is waiting for a button press",
"OK", "Cancel")
    local iBtnNum = 0
    while iBtnNum < 1 do
       iBtnNum = DialogCheckBtn(idNum)
    end
    DialogClose(idNum)
```



This dialog box is waiting for a button press

CRC Sig.: 2REHH5
Descrp.: fdsa
Sig.: 642EB2-6F6F0

This dialog box will remain open until the OK or Cancel button is pressed. It also changes colors on the monitor. Everything black changes to green and everything white changes to yellow.

DirUtils

Explanation:

Use this function to help with directory manipulation such as listing the current directory contents for a specified file type, getting the current director path, or changing to a specified directory. If the file type is a wirelist (.wir), the result will return the wirelist filename and its description if available.

Format:

```
sResult = DirUtils(iInputNum, sInput2)
   ↑          ↑              ↑
RESULT     FUNCTION       INPUTS
```

INPUT1 `iInputNum`	INPUT2 `sInput2`	RESULT `sResult`
`1` = change directory	String containing the path and directory the current directory will become.	nil **OR** String containing the current directory path.
`2` = get current directory contents	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: "*.*"	nil **OR** String containing the contents of the current directory of the specified file type. For wirelist (.wir) file types: The string will contain the filename and its description if available. The filename and description will be separated by a space, the pipe character: |, and a space.
`3` = get current directory path	NOT USED	nil **OR** String containing the current directory path.

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.

dofile

Explanation:

Use this function to execute statements and/or functions contained in other files. For example, the function can run a subroutine contained in an external file. It will open the file and run the code contained in it.

Format:

```
dofile(sFilename)
   ↑        ↑
FUNCTION  INPUT
```

INPUT1 `sFilename`	RESULT1 myResult	RESULT2 myValues **(optional)**
String containing the filename that contains statements and functions to run.	nil = errors occurred **OR** A value other than nil if the statements in the file ran.	If there are no errors, the function will return values from the statements that were executed.

Example:

-
  ```
  dofile("setcomm.lua")
  where: setcomm.lua =
           SetSerialParams("9600:8n1")
           Delay(0.5)
  ```

  This example will open the file, `setcomm.lua` and run the statements in that file to setup the COM port as follows: 9600 baud rate, 8 data bits, no parity, and 1 stop bits.

dostring

Explanation:

Use this function to execute statements and/or functions contained in the input string.

Format:

```
myResult [, myValues] = dostring(sString)
       ↑                    ↑           ↑
    RESULTS             FUNCTION      INPUT
```

INPUT1 sString	RESULT1 myResult	RESULT2 myValues **(optional)**
String containing statements and functions to run.	nil = errors occurred **OR** A value other than nil if the statements in the string ran.	If there are no errors, the function will return any values from the statements that were executed.

dostring, continued

Examples:

-
  ```
  local sReport = "SerialNumber = 12345678"
  myResult, SerialNumber = dostring(sReport)
  ```

  This example will execute the statements contained in the string, `myReport`. It will assign 12345678 to `SerialNumber` and return that value if there is no error. If there is an error, `myResult` will contain a nil; otherwise, it will contain a non-nil value.

-
  ```
  function iMakeTable(iRows, iColumns)
      local iRow
      local cpText = "{"
      iRow = 1
      while iRow <= iRows do
              if iRow > 1 then
                      cpText = cpText .. ",\n"
              end
              cpText = cpText .. "{"
              local iCol = 1
              while iCol <= iColumns do
                      if iCol > 1 then
                              cpText = cpText .. ", "
                      end
                      cpText = cpText .. "nil"
                      iCol = iCol + 1
              end
              cpText = cpText .. "}"
              iRow = iRow + 1
      end
      cpText = cpText .. "}"
      cpText = "return " .. cpText .. "\n "
      local aTable = dostring(cpText)
      return aTable
  end
  ```

  This example will call the function dostring to make a table iRows by iColumns.

-
  ```
  function TryDoStringList(TheList, TheMessage)
      local sCommand = " KeyPressNum = MessageBox(\""..TheMessage.."\""
      local iIndex = 1
      while TheList[iIndex] ~= nil do
              sCommand = sCommand..format(", \"%s\"",TheList[iIndex])
              iIndex = iIndex + 1
      end
      sCommand = sCommand..", \"Cancel\")"
      local myResult = dostring(sCommand.."\nreturn KeyPressNum")
  end

  TryDoStringList( {"one","two","three"}, "Some Message")
  ```

  This example creates a message box with three buttons. The id of the pressed button is then returned to the dostring function.

EEPROM

Explanation:

Use this function to talk to an I^2C (like 24LC00). See the `MicroLan` function to talk to Dallas Memory tokens using the MicroLan protocol. The valid address range for writing to and reading from is the address range of the eeprom. For example on an eeprom with 8 bytes, the valid range is 1-8. To talk to an eeprom on the scanner test points, some setup is needed. Connect a 10 K ohm resistor from VCC (pin 8) to sclk (pin 6) and another one from VCC (pin 8) to sdata (pin 5). Tie pins 1-4 (address & GND) together to establish an address for the part. Leave pin 7 floating (low) or pull to ground (pin 4).

Note: 1100 testers do not support this function

Format:

```
myResults = EEPROM(myInputs)
   ↑           ↑          ↑
```
RESULTS FUNCTION INPUTS

INPUTS `myInputs`	INPUT DESCRIPTION (event to run)	RESULTS `myResults`
1, GndPt, VCCPt, ClkPt, DataPt	Setup	nil = OK **OR** Error code
2	Clean up. It disconnects from the device.	nil = OK **OR** Error code
3, [fromAddress]	Read a byte from the EEPROM	nil = OK **and** Returns byte read **OR** Error code
4, toAddress, Data	Write a byte to the EEPROM	nil = OK **OR** Error code
5, fromAddress, MaxNumChars	Read Text	nil = OK **and** Returns text read **OR** Error code
6, toAddress, Data	Write Text	nil = OK **OR** Error code

Example:

-
  ```
  EEPROM(1, GroundPt, VCCPt, ClkPt, DataPt)
  EEPROM(6, 0, "testkd") -- write "testkd" to eeprom
  theResult = EEPROM(5, 0, 90) -- read "testkd" (90 chars max) from eeprom
  EEPROM(4, 4, ascii("e")) -- replace the "k" with a "e"
  theResult = EEPROM(5, 0, 90) -- read "tested" from eeprom
  print(theResult)
  EEPROM(4, 5, 248)   -- write a 248 to location 5
  theCharNum = EEPROM(3, 5) -- read the 248 from where "d" once was
  print(theCharNum)  -- get the non-ascii number
  EEPROM(2)
  ```

This example sets up the eeprom and then writes "testkd" to the eeprom. It reads back up to 90 characters maximum, including the "testkd". It then replaces the "k" with an "e" to form "tested" and again reads from the eeprom. It then writes a byte 248 to location 5 and then reads it out again from location 5. Finally, it does cleanup setting everything back to a default state.

error

Explanation:

Use this function to display an error message and stop running the current function. The error will be captured in a message box with the text, `Script error:` and a CANCEL button. The script file and the point where the error occurred will be displayed.

To provide more information about errors when creating and compiling a script, insert the `$debug` command at the top of the script file. When an error occurs in a script compiled with this option, it will display the line where the error occurred.

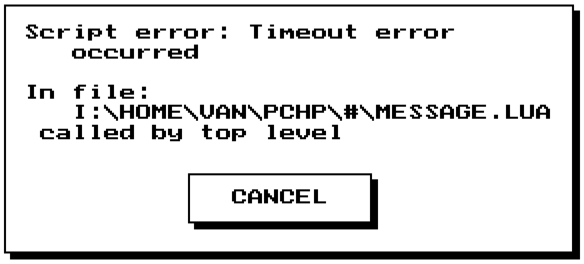
Format:

```
error(sErrorMessage)
  ↑           ↑
FUNCTION    INPUT
```

INPUT	DESCRIPTION
`sErrorMessage`	String containing the error message to display.

Examples:

- `error("Timeout error occurred")`



```
Script error: Timeout error
   occurred

In file:
   I:\HOME\VAN\PCHP\#\MESSAGE.LUA
 called by top level
```

This Touch1 example displays the message "Timeout error occurred".

- `error("Timeout error occurred")`

```
    SCRIPT ERROR
  Error: Timeout
   error occurred
File: error.lua
```

```
    SCRIPT ERROR
    called by
  DoOnTestEvent
CANCEL
```

This 1100 example displays the message "Timeout error occurred".

execute

Explanation:

This function invokes a copy of the DOS command processor and passes the command contained in the input string to it for processing. The function returns a system-dependent error code.

Note: 1100 testers do not support this function

Format:

```
execute(sCommand)
   ↑        ↑
FUNCTION  INPUT
```

INPUT `sCommand`	RESULT
A string containing the command to be executed.	Executes the command passed into the function.

Example:

-
  ```
  execute("ls >listing")
  readfrom(“listing”)
  tmpList = read(“.*”)
  readfrom()
  MessageBox(tmpList)
  ```

  This example will execute the LINUX directory command and put the results in the file called `listing`. The `listing` file is then read and the contents are stored in the variable `tmpList`. A message box then displays the contents of `tmpList` on the touch screen.

floor

Explanation:

The function returns the largest integer not greater than the input number. The resulting number will round downward. Use this function to round down real numbers to an integer value for input to other functions. See the function `ceil` for rounding upward.

Format:

```
iResult = floor(myInputNum)
   ↑        ↑        ↑
RESULT   FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `iResult`
Contains the number the `floor` function will be computed on.	Contains the largest integer value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = 7.93
  myValue = floor(myInputNum)
  ```

  The function will return `myValue` equal to 7.

-
  ```
  iResult = floor(8.51)
  ```

  The function will return `iResult` equal to 8.

format

Explanation:

Use this function to format information for output to the display or to a printer. This function is necessary to organize text to be printed using the Cirris print functions: `WriteToSerial` and `SendTextToParallelPrinter`. The input, `sFormatString,` follows the same rules as the printf() in C. The function `strcat` is also useful to combine strings together in conjunction with `format`.

Format:

```
myFormattedResult = format(sFormatString, item1, item2, item3,
                                                          ...)
       ↑                  ↑            ↑
   RESULT             FUNCTION      INPUTS
```

INPUTS	INPUT DESCRIPTIONS	RESULT `myFormattedResult`
`sFormatString`	A string containing the codes to format the input items. See *Some Format Codes* below for a table of different codes.	Result containing the input items formatted according to the `sFormatString.`
`item1, item2, item3, ...`	Items that will be formatted according to the `sFormatString`.	

Some Format Codes:

Code	**Description**
\n	A new line
\r	Back up to beginning of the current line. This can be used for underlining text outputted to a printer
%%	The "%" character
%c **or** \12	Formfeed for printing
%e	Expect an exponential number after the format code and use it to replace the %e
%f	Expect a floating point number after the format code and use it to replace the %f
%10.2f	Expect a floating point number after the format code and left justify it in a ten character area with two characters after the decimal point and use it to replace the %10.2f. For example, to get 00972.1 use %05.1f
%g	Expect a number of type double after the format code and use it to replace the %g . Trailing zeros are removed from the result and a decimal-point character only appears if it is followed by a digit
%i	Expect an integer (number) after the format code and use it to replace the %i
%6i	Expect an integer after the format code and right-justify it in a six character area, using it to replace the %6I
%s	Expect a string after the format code and use it to replace the %s
%36s	Expect a string after the format string and right-justify it in a 36 character area which will replace the %36s
%x	Expect a hexadecimal number after the format code and use it to replace the %x. Hexadecimal notation uses the digits "0" - "9" and the characters "a" -"f" or "A" -"F" for "x" or "X" conversions respectively, as the hexadecimal digits

format, *continued*

Examples:

- `myFormattedResult = format("Tested %i cables,” myGoodCableCount)`

  In this example, `"Tested %i cables"` is the `sFormatString` and `myGoodCableCount` is `item1`. The function will return `myFormattedResult = Tested 187 cables` if `myCoodCableCount` is 187.

- `myFormattedResult = format("Operator:  %30s\n", operator)`

  In this example `"Operator: %30s"` is the sF`ormatString` and `operator` is `item1`. This function will return `myFormattedResult` = Operator: BOB if `operator` is BOB.

- `myFormattedResult = format("GOOD CABLE\r__________")`

  In this example `"GOOD CABLE\r___________"` is the `sFormatString` . There are no `items`. This function will return `myFormattedResult` = GOOD CABLE if it is outputted on a printer. The code, `\r`, cannot be used for output on the screen because you cannot back up over text that has already been displayed.

- `"Voltage".."="..format("%3.2f",fVolts).."V"`

  In this example "%3.2f " is the `sFormatString` . There are no `items`. In this example, the string will be formatted as follows if fVolts = 1.2345 Voltage = 1.23 V

Get4WPairPt

Explanation:

Use this function to determine if the given point is a fourwire pair point. A string containing the label for the point is returned or it returns ‘nil’ if there is an error or the point is not a fourwire pair. The input for point is an integer number between 1 and 1024 or a string containing the point label.

Format:

```
sFourwirePairLabel = Get4WPairPt(thePoint)
         ↑                  ↑             ↑
       RESULT           FUNCTION        INPUT
```

INPUT `thePoint`	RESULT `sFourwirePairLabel`
An integer containing the pin number between 1 and 1024 **OR** A string containing the point label	nil = not a fourwire pair point **OR** A string containing the label text

Example:

-
  ```
  local iPinNumber = 3
  local 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 fourwire pair point, a message box listing the point will be displayed.

getbuttonpress

Explanation:

Use this function to get the button id number for the button pressed on the current screen. This function is useful for displaying a message if a certain button was pressed.

Format:

```
iButtonIdNumber = getbuttonpress()
         ↑                 ↑
      RESULT           FUNCTION
```

RESULT for Touch1 `iButtonIdNumber`
An integer containing the id number for the button just pressed on the screen.

RESULT for 1100 testers
Integer containing one of the following codes: **0** = None **1** = Down Button **2** = Back Button **4** = Up Button **8** = Line 2 **16** = Line 3 **32** = Line 4

Example:

```
if getbuttonpress() == iStartHipotBtn then
      MessageBox(“High Voltage Warning!“)
end
```

In this example, a message box will be displayed if the start hipot button was pressed.

GetCableStatus

Explanation:

Use this function to get a good or bad status for the cable tested. The function has no inputs.

Format:

```
iCableStatus = GetCableStatus()
      ↑                ↑
   RESULT          FUNCTION
```

RESULT `iCableStatus`
Integer containing one of the following codes: **0** = Good **-99** = Bad

Example:

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

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

GetCapMeasurement

Explanation:

Use this function to get the measured capacitance value between two points. It is good practice to check the result is nil before using it in further calculations.

Format:

```
fMeasuredValue, myResult = GetCapMeasurement(myPoint1, myPoint2)
                    ↑                       ↑                ↑
                 RESULTS                FUNCTION          INPUTS
```

INPUTS `myPoint1, myPoint2` (points for measurement)	RESULT1 `fMeasuredValue`	RESULT2 `myResult`
String containing the first and second points as a default or custom label. **OR** Integer containing the first and second points where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter)	Floating point number containing the measurement between myPoint1 & myPoint2. **OR** **0** = invalid input points	**nil** = No error **-99** = invalid point

Example:

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

  The function will return the measured capacitance value between "J1-001" and "J2-005" in `fMeasuredValue`. It checks the result is nil before displaying a message box with the measured capacitance value.

GetComponentCount

Explanation:

Use this function to count number of components in the current wirelist.

Format:

iCount = GetComponentCount()

RESULT `iCount`
Integer containing the total number of components in the component section of the current wirelist.

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.

GetComponentDetails

Explanation:

Use this function to get setup and test result details about components in the current wirelist.

Format:

iResult,iType = GetComponentDetails(iComponentIndex, iDetail)

Input iComponentIndex	**Input** iDetail	**Result** iType	**Result** iResult
Integer Position of the component in the wirelist.	0 = Component Type	Integer 1 = Resistor 2 = Diode 3 = Link 4 = Capacitor 5 = Relative Capacitance 6 = Twisted Pair 7 = 4 Wire Resistor 8 = 4 Wire Wire 9 = Wire	Integer 0 = No error, iType is valid. -9999 = Error, iType is invalid
	1 = Pass Fail Status	Integer 0 = Fail 1 = Pass	
	2 = From Test Point	Integer Raw node number. NOTE:This value is zero based.	
	3 = To Test Point	Integer Raw node number. NOTE: This value is zero based.	
	4 = Expected Value	Float Nominal or expected value applied to the component setup.	
	5 = Tolerance	Integer Tolerance applied to the component setup.	
	6 = Measured Value 1	Float Measured values from most recent test.	
	7 = Measured Value 2	Float Reverse voltage from Diode component returned from most recent test.	

Example:

```
iComponentIndex = 1
iDetail = 2
iResult,iType = GetComponentDetails(iComponentIndex, iDetail)
if iResult == 0 then
        MessageBox("From point = "..GetPinLabel(iType + 1))
end
```

This function will get the "From point" assigned to the component in position 1 of the current wirelist. The function uses the GetPinLabel function to convert the raw node information to the labeled node information then displays that in a message box on the tester's screen.

Note: Since the raw node number is zero based, a one was added to the returned value. The GetPinLabel function does not work on a zero based argument.

GetDateAsText

Explanation:

Use this function to get the current date as a text string either in individual units or all together. Each input number will return a different date selection. You can get the complete date in two different formats: MM/DD/YYYY or DD/MM/YYYY. If the function has input, it will return a description of itself.

Note: 1100 testers do not support this function

Format:

```
sTextResult = GetDateAsText(iInputNum)
     ↑                ↑          ↑
   RESULT          FUNCTION    INPUT
```

INPUT `iInputNum` (integer)	RESULT `sTextResult`
Nothing	Function description
1	String containing current **month** (no preceding 0)
2	String containing current **day** (no preceding 0)
3	String containing current **year** in format YYYY
4	String containing current **year** in format YY (no preceding 0, 2001=1)
5	String containing **current date** in format MM/DD/YYYY
6	String containing **current date** in European format DD/MM/YYYY

Examples:

- `sCurrentMonth = GetDateAsText(1)`

  The function will return the current month as a text string in `sCurrentMonth`. For example, `sCurrentMonth` = "5".

- `sCurrentYearText = GetDateAsText(3)`

  The function will return the current year as a text string in the RESULT, `sCurrentYearText`. The year will be formatted as "YYYY".
  For example, `sCurrentYearText` = "1999".

- `sCurrentDate = GetDateAsText(5)`

  The function will return the current date as a text string in the format "MM/DD/YYYY" in the RESULT, `sCurrentDate`. For example, `sCurrentDate` = "10/03/1998".

- `sCurrentDate = GetDateAsText(6)`

  The function will return the current date as a text string in the European format "DD/MM/YYYY" in the RESULT, `sCurrentDate`.
  For example, `sCurrentDate` = "14/09/1998".

GetErrorSignature

Explanation:

Use this function to get the error signature for a bad cable as a text string. If the cable tests good and this function is called, it returns the cable signature. The function has no inputs.

Note: This function returns cable signatures and not CRC signatures.

Format:

```
sSigText = GetErrorSignature()
      ↑                ↑
   RESULT          FUNCTION
```

RESULT `sSigText`
String containing the error signature if the cable tested bad. **OR** String containing the cable signature if the cable tested good.

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.

GetErrorText

Explanation:

Use this function to get 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, "".

Note: See the function `TWLGetErrorText` for use with the function `TestWirelist`.

Format:

```
sTestResult = GetErrorText()
     ↑                ↑
   RESULT         FUNCTION
```

RESULT `sTestResult`
String containing error text if the cable tested was bad **OR** An empty string if the cable tested good.

Examples:

- `sTestResult = GetErrorText()`

  If the cable tests bad, the function will return the errors as text in `sTestResult`. If the cable tests good, it will return an empty string, "", in `sTestResult`.
  For example, if the cable had an open, the error text would be formatted as follows:
  "NET 1:
  J1-001 OPEN J1-002 J1-007"

- `sCableErrorText = GetErrorText()`

  If the cable tests bad, the function will return the errors as text in `sCableErrorText`. If the cable tests good, it will return an empty string, "", in `sCableErrorText`.
  For example, if the test finished before the cable was hipotted, the error text would be as follows:
  "Cable Not Hipot Tested"

GetHardwareVersion

Explanation:

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

Format:

```
sHardwareVersion = GetHardwareVersion()
      ↑                  ↑
   RESULT            FUNCTION
```

RESULT `sHardwareVersion`
String containing the hardware version running on the tester.

Example:

- `sHardwareVer = GetHardwareVersion()`
  `MessageBox(sHardwareVer)`

  In this example, the hardware version for the tester would be displayed in a message box.

GetNumberTested

Explanation:

Use this function to get a count on the cables tested (total, number good, number bad, number intermittent). These totals are the same totals displayed in the Test Summary screen on the tester. Each input number will return a different count type. If the function has no input, it will return a description of itself.

Format:

```
iTested = GetNumberTested(iInputNumber)
   ↑            ↑              ↑
RESULT      FUNCTION         INPUT
```

INPUT `iInputNum (integer)`	RESULT `iTested`
Nothing	Function description
1 = total cables	Integer containing the total number of cables tested
2 = good cables	Integer containing the number of good cables tested
3 = bad cables	Integer containing the number of bad cables tested
4 = intermittents	Integer containing the number of intermittents

Examples:

- `iInputNum = 1`
  `iTested = GetNumberTested(iInputNum)`

  The function returns the total number of cables tested in the RESULT, `iTested`.

- `iNumGoods = GetNumberTested(2)`

  The function returns the number of good cables in the RESULT, `iNumGoods`.

- `iTotalBadTested = GetNumberTested(3)`

  The function returns the number of bad cables in the RESULT, `iTotalBadTested`.

GetPinLabel

Explanation:

Use this function to get the default or custom label for a pin number. Pin numbers (test points) are between 1 and 1024.

Format:

```
sLabelText = GetPinLabel(iPinNumber, [iReturnDefaultLabel])
   ↑                ↑                   ↑
RESULT           FUNCTION             INPUTS
```

INPUT1 `iPinNumber`	INPUT2 `iReturnDefaultLabel` **(optional)**	RESULT `sLabelText`
An integer containing a pin number (test point) between 1 and 1024.	**≥ 1** = return the label in the default Jx-xxx format	String containing the custom label or the default test point if a second parameter was used or nil if there is no label
NOT USED	NOT USED	Function description returned as an error.

Examples:

- `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 the string Jx-005 (x is the adapter number) in `sDefaultLabel.`

- `GetPinLabel()`

  The function will return an error giving the function description.

GetPtType

Explanation:

Use this function to determine 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 low-level commands (see LowLevelCommand) dealing with fourwire resistance measurement and the high current source.

Format:

```
iPointType = GetPtType(thePoint)
   ↑              ↑            ↑
 RESULT        FUNCTION      INPUT
```

INPUT `thePoint`	RESULT `iPointType`
An integer containing the pin number between 1 and 1024 **OR** A string containing the point label	Integer containing one of the following codes: **1** = Stress High (can source high current) **2** = Stress Low (can sink high current)

Example:

```
SrcPt = "j1-002"
SinkPt = "j1-005"
local tPt = Get4WPairPt(SrcPt)
if tPt = = nil  then
            error(SrcPt .. " is not part of a 4W pair.")
end
if GetPtType(SrcPt) = = 1 then    -- relay could tie this point to the HC source
      wStressHi = SrcPt
      wSenseHi = tPt
else
            wStressHi = tPt
      wSenseHi = SrcPt
end
tPt = Get4WPairPt(SinkPt)
if tPt = = nil then
            error(SinkPt .. " isn't part of a 4W pair.")
end
if GetPtType(SinkPt) = = 1 then  -- relay could tie this point to the HC source
            wStressLo = tPt
      wSenseLo = SinkPt
else
            wStressLo = SinkPt
      wSenseLo = tPt
end
MessageBox("StressHi: " .. wStressHi .. "\nSenseHi: " .. wSenseHi .. "\n\nStressLo: " .. wStressLo .. "\nSenseLo: " ..
wSenseLo)
```

In this example, the wirelist is assumed to have fourwire pairs. In fourwire pairs, one point is “hidden” in the wirelist. In this example, there are two non-hidden points that are each part of a fourwire pair. The two points are “j1-002” and “j1-005”. The two hidden points are “j1-001” and “j1-006”. When performing tests that use the high current source, turn on the relay in the fourwire pair that connects to Stress High to supply the current. On the other side of the component you are testing, turn on the relay tied to Stress Low to sink the current. Since either the displayed or the hidden point could be tied to Stress High, you can use this function to determine which points tie to which Stress bus inside the tester.

GetRawPointNum

Explanation:

Use this function to get the pin number for a custom label. Pin numbers are between 1 and 1024.

Format:

```
iPinNumber = GetRawPointNum(sLabelText)
    ↑              ↑             ↑
  RESULT        FUNCTION       INPUT
```

INPUT `sLabelText`	RESULT `iPinNumber`
Nothing	Function Description
String containing the custom label text.	Integer containing the pin number of the custom label. Pin numbers are between 1 and 1024. **OR** nil = invalid point

Examples:

- `sLabelText = “GROUND”`
  `iPinNumber = GetRawPointNum(sLabelText)`

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

- `GetRawPointNum()`

  The function will return an error giving the function description.

GetRelCapMeasurement

Explanation:

Use this function to measure the capacitance of a single net to all other points and then divide that value by the capacitance of another net to all other points. This is method for verifying a shield is in a given position. It is good practice to check the result is nil before using it in further calculations.

Format:

```
fRatio, myResult = GetRelCapMeasurement(myShieldPoint,
                                         myReferencePoint)
          ↑                 ↑               ↑
       RESULTS          FUNCTION         INPUTS
```

INPUTS `myShieldPoint,` `myReferencePoint` (points for measurement)	RESULT1 `fRatio`	RESULT2 `myResult`
String containing the shield and reference points as a default or custom label. The shield point is a point connected to a shield. The reference point should be a point inside the shield. **OR** Integer containing the shield and reference points where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter)	Float containing the ratio (0.01-100) of the measured capacitance of the shield point to all other points divided by the capacitance of the reference point to all other points. **OR** **0** = invalid input points	**nil** = No error **-99** = invalid point

Example:

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

  The function will find the relative 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

Explanation:

Use this function to get the measured resistance value between two points. It is good practice to check the result is nil before using it in further calculations. See the function GetResistanceMeasurement4W to get measured resistance values between two fourwire points.

Format:

```
fMeasuredValue, myResult = GetResistanceMeasurement(myPoint1,
                                                       myPoint2)
            ↑                          ↑                  ↑
         RESULTS                    FUNCTION            INPUTS
```

INPUTS `myPoint1, myPoint2` (points for measurement)	RESULT1 `fMeasuredValue`	RESULT2 `myResult`
String containing the first and second points as a default or custom label. **OR** Integer containing the first and second points where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter)	Floating point number containing the measurement between myPoint1 & myPoint2. **OR** **0** = invalid input points	Contains one of the following codes: **nil** = No error **-99** = invalid point

Examples:

-
  ```
  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.

-
  ```
  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.

GetResistanceMeasurement4W

Explanation:

Use this function to get the measured resistance value between two fourwire points. It is good practice to check the result is nil before using it in further calculations. See the function `GetResistanceMeasurement` to get the measured resistance value between two test points.

Format:

```
fMeasuredValue, myResult = GetResistanceMeasurement4W(my4WPoint1,
                                                      my4WPoint2)
```

↑ RESULTS ↑ FUNCTION ↑ INPUTS

INPUTS
`my4WPoint1,`
`my4WPoint2`
(points for measurement)
	RESULT1
	`fMeasuredValue`
		RESULT2
		`myResult`

String containing the first and second fourwire pair points as a default or custom label. **OR** Integer containing the first and second fourwire pair points where J1-001 = 1, J3-001 = 65 (The number increments the same as an AHED-64 adapter)	Floating point number containing the fourwire measurement between my4WPoint1 & my4WPoint2. **OR** **0** = invalid fourwire pair point inputs	Contains one of the following codes: **nil** = No error **-99** = invalid fourwire pair point

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 fourwire resistance value between "J1-003" and "J1-005" in `fMeasuredValue`. `myResult` will be nil since there is no error.

GetSystemInfoAsText

Explanation:

Use this function to get system information about the tester. Each input number will return a string containing different system information. For the operator name, if security is on, the function returns the operator name of the user who unlocked the Touch 1. If the function has no input, it will return a description of itself. If no system information is available, the return result will be nil.

Format:

```
sTextResult = GetSystemInfoAsText(iInputNum)
    ↑                   ↑                   ↑
  RESULT             FUNCTION             INPUT
```

INPUT `iInputNum` (integer)	RESULT `sTextResult` (string)
Nothing	Function description
1	String containing an eight character **tester serial number**
2	If security is ON, a string containing the **operator name** (up to 30 characters) of the person who unlocked the Touch 1
3	String containing the **software version** of the tester
4	String containing the **hardware version** of the tester
5	String containing the number of **test points** in system
6	String containing the overall scanner voltage of the tester
7	String containing the tester type: “Touch1” or “1100H+” or “1100R+” or “Touch1 LV”
	or **nil** = no system information for input

Examples:

- `sTextResult = GetSystemInfoAsText(1)`

  The function returns the tester serial number as a text string in the RESULT, `sTextResult`. For example, `sTextResult` = "12345678".

- `sOperatorName = GetSystemInfoAsText(2)`

  If security is on, the function returns the operator name of the user that unlocked the Touch 1 in the variable, `sOperatorName`. For example, `sOperatorName` = "JOHN DOE".

- `functionDescription = GetSystemInfoAsText()`

  The function returns the function description as a text string in the RESULT, `functionDescription`.

GetTimeAsInteger

Explanation:

Use this function to get the current time as an integer in individual units. Each input number will return a different time selection (hour, minutes, seconds, clock ticks). For the number of clock ticks, the function returns the number of ticks of processor time since the Touch 1 was turned on until the number hits a long integer and then it wraps around. There are 1000 ticks per second. If the function has no input, it will return a description of itself.

Note: 1100 testers do not support this function

Format:

```
iTimeFormat = GetTimeAsInteger(iInputNum)
    ↑                ↑               ↑
  RESULT          FUNCTION         INPUT
```

INPUT `iInputNum` (integer)	RESULT `iTimeFormat` (integer)
Nothing	Function description
1	Integer containing the current **hour** up to 23
2	Integer containing the current **minutes**: 0 - 59
3	Integer containing the current **seconds**: 0 - 59
4	Integer containing the number of **clock ticks** Ticks increment and decrement like this: 0 (Turn on Touch1) .. .. 2147483 -2147483 .. .. 0 .. .. 2147483 -214783 0

Examples:

- `iTimeFormat = GetTimeAsInteger(1)`

  The function returns the current hour as an integer in the RESULT, `iTimeFormat`. For example, `iTimeFormat` = `15`.

- `iCurrentSeconds = GetTimeAsInteger(3)`

  The function returns the current number of seconds as an integer in the RESULT, `iCurrentSeconds`. For example, `iCurrentSeconds` = 58.

GetTimeAsText

Explanation:

Use this function to get the current time as a text string in individual units or all together. Each input number will return a different time selection (hour, minute, second, total time, clock ticks). For the number of clock ticks, the function returns the number of ticks of processor time since the Touch 1 was turned on. There are 1000 ticks per second. If the function has no input number, it will return a description of itself.

Note: 1100 testers do not support this function

Format:

```
sTextResult = GetTimeAsText(iInputNum)
     ↑                ↑         ↑
  RESULT           FUNCTION    INPUT
```

INPUT `iInputNum` (integer)	RESULT `sTextResult` (string)
Nothing	Function description
1	String containing current **hour** up to 12
2	String containing current **minutes**: 0 - 59
3	String containing current **seconds**: 0 - 59
4	String containing current **total time** up to 23:59:59
5	String containing the number of **clock ticks**

Examples:

- `sTextResult = GetTimeAsText(1)`

  The function returns the current hour as a text string in the RESULT, `sTextResult`. For example, `sTextResult` = "11".

- `sCurrentSecondsText = GetTimeAsText(3)`

  The function returns the current number of seconds as a text string in the format "SS" in the RESULT, s`CurrentSecondsText`. For example, s`CurrentSecondsText` = "23".

- `sCurrentTime = GetTimeAsText(4)`

  The function returns the current time as a text string in the format "HH:MM:SS" in the RESULT, s`CurrentTime`. For example, s`CurrentTime` = "12:43:08".

- `sNumClockTicks = GetTimeAsText(5)`

  The function returns the number of clock ticks since the Touch 1 was turned on as a text string in the RESULT, s`NumClockTicks`. For example, s`NumClockTicks` = "4737".

- `functionDescription = GetTimeAsText()`

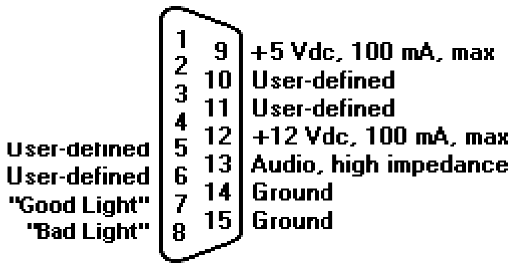
  The function returns the function description as a text string in the RESULT, `functionDescription.`

GetUserOutputStates

Explanation:

Use this function to get the state of the digital outputs on the user I/O port. You can use any combination of the six inputs and the function will return their respective states (logic low or logic high). The function reads the output port directly. The state of the output port is determined by using the `SetUserOutputStates` function or by using the setup for Digital Outputs on the tester.

Pinout:



Format:

```
iOutputState1,
iOutputState2,
iOutputState3,
iOutputState4,
iOutputState5,
iOutputState6,
iOutputState7,
iOutputState8 = GetUserOutputStates(iDigitalOutput1,
                                    iDigitalOutput2,
                                    iDigitalOutput3,
                                    iDigitalOutput4,
                                    iDigitalOutput5,
                                    iDigitalOutput6,
                                    iDigitalOutput7,
                                    iDigitalOutput8)
```

↑ RESULTS ↑ FUNCTION ↑ INPUTS

INPUTS `iDigitalOutputX` (integer)	RESULTS `iOutputStateX` (integer)
1 = Pin 5	Integer containing one of the following codes: **0** = Low (on or conducting) **1** = High (off or not conducting)
2 = Pin 6	
3 = Pin 10	
4 = Pin 11	
5 = Good Light & Pin 7 (Touch1) **5** = Good Light Only (1100)	
6 = Bad Light & Pin 8 (Touch1) **6** = Bad Light Only (1100)	
7 = Pin 7 (1100 only)	
8 = Pin 8 (1100 only)	

GetUserOutputStates, *continued*

Examples:

-
  ```
  iDigitalOutput1 = 1
  iDigitalOutput2 = 2
  iDigitalOutput3 = 3
  iDigitalOutput4 = 4

  iOutput1, iOutput2,
  iOutput3, iOutput4 = GetUserOutputStates(iDigitalOutput1,
                                          iDigitalOutput2,
                                          iDigitalOutput3,
                                          iDigitalOutput4)
  ```
  The function will get the logic states of all the outputs: Pin 5, Pin 6, Pin 10, and Pin 11.

- `iGoodLightState = GetUserOutputStates(5)`

  The function will get the state of the Goodlight pin where 0 = light on.

- `iOutputState = GetUserOutputStates(4)`

  The function will get the state of Pin 11.

GetWirelistInfoAsText

Explanation:

Use this function to get text information on the current wirelist. Each input number will return different information about the wirelist. If the function has no input number, it will return a description of itself.

Format:

```
sWirelistText = GetWirelistInfoAsText(iInputNum)
     ↑                       ↑                    ↑
   RESULT                FUNCTION               INPUT
```

INPUT `iInputNum` (integer)	RESULT `sWirelistText` (string)
Nothing	Function description
1	String containing the current **wirelist filename**
2	String containing the current **location or path** of wirelist file without the filename. (up to 144 characters)
3	String containing the **cable description** of wirelist. (Up to 30 characters)
4	String containing the **cable signature**
5	String containing the **wirelist signature** (complex cables only)
6	String containing the **insulation test parameter signature** (complex cables only)
7	String containing the **adapter position** (Jx), **a space, and then the adapter signature** (six characters)
8	String containing the **adapter position and signature** followed by a ***** separator followed by the **adapter description** (up to 30 characters)
9	NOT USED
10	String containing the **Low Voltage Settings** with each parameter on a separate line
11	String containing the **High Voltage Settings** with each parameter on a separate line
12	String containing 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	String containing each **component** on a separate line
14	String containing each **custom or default test point label** on a separate line
15	String containing each **fourwire pair** on a separate line
16	String containing the **CRC Signature**
17	String containing the name & location of the **Custom Component Script**
18	String containing 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 the RESULT, `sWirelistText`. For example, `sWirelistText` = "TESTFILE.WIR".

GetWirelistInfoAsText, *continued*

- `sCableDescription = GetWirelistInfoAsText(3)`

  The function returns the cable description of the current wirelist as a text string in the RESULT, `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 the RESULT, `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 the RESULT, `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 the RESULT, `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 the `RESULT, functionDescription.`

GetWrappedText

Explanation:

Use this function to wrap text at a specified line length. If no line length is input, the default for the Touch 1 is 35 characters per line and 20 characters per line for the 1100. Using the other optional parameters, the text can be centered and linefeeds can be replaced with carriage return linefeeds.

Format:

```
sWrappedText = GetWrappedText(sTextToWrap, [iLineLength],
                               [bCenter], [bReplaceLFWithCRLF])
      ↑                ↑                    ↑
    RESULT          FUNCTION             INPUTS  ( [ ] = optional )
```

INPUTS	INPUT DESCRIPTION	RESULT `sWrappedText`
`sTextToWrap`	A string containing the text which will be wrapped.	String containing the wrapped text at the specified line length.
`iLineLength` **(optional)**	Integer indicating the number of characters per line for the wrapped text. Defaults if no line length is given are 35 characters per line for the Touch1 and 20 characters per line for the 1100.	
`bCenter` **(optional)**	0 = Text is not centered 1 = Center text Default = 0	
`bReplaceLFWithCRLF` **(optional)**	0 = Linefeeds not altered 1 = Replace linefeeds (LF) with carriage return linefeeds (CRLF) Default = 0	

Example:

- `sWrappedText = GetWrappedText(sTextToWrap, 80, 0 ,1)`

  The function returns the text wrapped at 80 characters per line as a string in the RESULT, `sWrappedText`. The text is not centered and linefeeds (LF) were replaced with carriage return linefeeds (CRLF).

HipotNetTiedToPoint

Explanation:

Use this function to hipot a net containing the given input point. This function puts the tester in a known state by clearing all relays, sinking all points, and then turning on the relay for the input point. All unused points are grounded. If optional parameters are given, the high voltage settings in the wirelist will be overridden and the optional parameters will be used. The voltage, insulation resistance, and duration must be given if using the optional parameters. If no optional parameters are given, the high voltage settings in the loaded wirelist will be used. The function will return two results: a number indicating the status of the test and the measured leakage resistance. If the function is called without any parameters, it will return a description of itself. This function will do a hipot test whether or not high voltage testing is turned on in the loaded wirelist.

Format *(Standard Settings):*

```
iResult, fResisMeasured = HipotNetTiedToPoint(sPoint [,fVoltage,
                                             fInsulRes, fDuration
                                                  [,fMaxSoakTime]])
             ↑                    ↑                          ↑
          RESULT              FUNCTION                    INPUTS
```

INPUTS	INPUT DESCRIPTION	RESULT1 `iResult`	RESULT2 `fResisMeasured`
`sPoint`	String containing the test point on the net to be hipotted.	Integer containing one of the following codes: **0** = Passed Test **1** = Has leakage **2** = Overcurrent **3** = Dielectric Failure **99** = Other Failure, such as user-aborted hipot. **100** = Invalid Test Point **101** = * Invalid Voltage **102** = * Invalid Insulation Resistance **103** = * Invalid Hipot Duration **104** = * Invalid Soak Time * = If these errors occur, the hipot test will not be performed.	Float containing the measured leakage resistance **or** nil for some RESULT1 errors (2, 3, 99)
`fVoltage` (**optional**)	Float containing the voltage applied during the hipot test. Range: 50 – 1000 volts.		
`fInsulRes` (**optional**)	Float containing the insulation resistance. It is the minimum resistance allowed between unintended connections. Values: 5,000,000-1,000,000,000 Ω		
`FDuration (seconds)` (**optional**)	Any float containing the duration of the hipot. Values: 0.01 - 120 seconds		
`FMaxSoakTime (seconds)` (**optional**) (Set to zero if not included with the other optional parameters.)	Float containing the maximum amount of time the hipot voltage is applied to the test point after the DWV portion of the test and before the IR test. Soak time stops when the measured insulation resistance exceeds `fInsulRes` for the duration of the IR test. Values: 0(off), 0.01 - 120 seconds		

HipotNetTiedToPoint, *continued*

***Format** (Advanced DC Settings):*

```
iResult, fResisMeasured = HipotNetTiedToPoint(sPoint, “DC”,
                                fDWVVoltage, fDWVCurrent,
                                iDCDuration, fIRVoltage,
                                fIRInsulRes, fTimeGoodFor, fSoak,
                                iSoakUntilGood)
          ↑                ↑                        ↑
       RESULT           FUNCTION                 INPUTS
```

INPUTS	INPUT DESCRIPTION	RESULT1 `iResult`	RESULT2 `fResisMeasured`
`sPoint`	String containing the test point on the net to be hipotted.	**0** = Passed Test **1** = Has leakage **2** = Overcurrent **3** = Dielectric Failure **99** = Other Failure such as user aborted hipot **100** = Invalid Test Point **110** = * Invalid Frequency **111** = * Invalid DWV Voltage **112** = * Invalid DWV Max. Current **114** = * Invalid DWV DC 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.	Float containing the measured leakage resistance **or** nil for some RESULT1 errors (2, 3, 99)
`“DC”`	String indicating using advanced DC settings.		
`fDWVVoltage`	Float containing the voltage applied during the Dielectric Strength Test. Range: 50 – 1500 volts		
`fDWVCurrent`	Float containing the maximum current that can flow through insulation during hipot test. Range: .1mA – 1.5 mA		

Continued on the next page

HipotNetTiedToPoint, *continued*

`fDCDuration`	Any float containing the time in seconds the voltage is applied to each net of points during the DWV test. Range: 0.01 - 120 seconds		
`fIRVoltage`	Float containing the voltage applied during the Insulation Resistance Test. Range: 50 – 1500 V		
`fIRInsulRes`	Float containing the insulation resistance. It is the minimum resistance allowed between unintended connections. Values: 5,000,000-1,000,000,000 Ω		
`fTimeGoodFor`	Float containing the time for the duration of the Insul Res Test after the Soak Time. Range: .002 – 120 sec		
`fSoak`	Float containing the time before starting the Insul Res Test. Depends on iSoakUntilGood value. If iSoakUntilGood = 0, tester waits this time before starting test. If iSoakUntilGood = 1, tester applies HV up to this time before the IR part of the test. Range: .002 – 120 sec		
`iSoakUntilGood`	**0** (off) = Use fSoak to set a definite time period before starting the Insul Res Test. **OR** **1** (on) = Use fSoak to set the longest time to wait for acceptable insul current leakage before reporting an error.		

HipotNetTiedToPoint, *continued*

Format *(Advanced AC Settings):*

```
iResult, fResisMeasured = HipotNetTiedToPoint(sPoint, “AC”,
                              iFrequency, fDWVoltage,
                              fDWVCurrent, iACDuration,
                              fIRVoltage, fIRInsulRes,
                              fTimeGoodFor, fSoak,
                              iSoakUntilGood)
        ↑                ↑                     ↑
      RESULT          FUNCTION               INPUTS
```

INPUTS	INPUT DESCRIPTION	RESULT1 `iResult`	RESULT2 `fResisMeasured`
`sPoint`	String containing the test point on the net to be hipotted.	**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.	Float containing the measured leakage resistance **or** nil for some RESULT1 errors (2, 3, 99)
`“AC”`	String indicating using advanced AC settings.		
`iFrequency`	Integer containing the frequency used for the DWV test. Range: 25-60 Hz		
`fDWVVoltage`	Float containing the voltage applied during the DWV test. Range: 50 – 1000 volts AC		
`fDWVCurrent`	Float containing the maximum current that can flow through insulation during AC hipot test. Range: .1mA – 1.5 mA		

Continued on the next page

HipotNetTiedToPoint, *continued*

`iACDuration`	Integer containing number of cycles the voltage is applied to each net of points during the DWV test. Values: 1-7200 cycles		
`fIRVoltage`	Float containing the voltage applied during the Insulation Resistance Test. Range: 50 – 1500 V		
`fIRInsulRes`	Float containing the insulation resistance. It is the minimum resistance allowed between unintended connections. Values: 5,000,000-1,000,000,000 Ω.		
`fTimeGoodFor`	Float containing the time for the duration of the Insul Res Test after the Soak Time. Range: .002 – 120 sec		
`fSoak`	Float containing the time before starting the Insulation Resistance (IR) Test. It depends on iSoakUntilGood value. If iSoakUntilGood = off, tester waits this time before starting test. If iSoakUntilGood = on, tester up to this time before reporting an error. Range: .002 – 120 sec		
`iSoakUntilGood`	**0** (off) = Use fSoak to set a definite time before starting the IR Test. **OR** **1** (on) = Use fSoak to set the longest time to wait for acceptable insul current leakage before reporting an error.		

Examples:

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

  This example hipots the net tied to the point, J1-001, using the high voltage parameters in the loaded wirelist. The variable `iResult` will contain the test result. See the custom component script, `Hipot_01.cmp` on page 32.

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

  This example hipots the net tied to the custom test point label, Blue. It uses the following optional parameters which are independent of the wirelist parameters: high voltage = 50 volts, insulation resistance = 5M ohms, and duration = 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.

HipotNetsTiedToPoints

Explanation:

Use this function to hipot multiple nets (like connectors) containing the given input points at one time. This allows manual control over high speed hipot. It is the fastest way to hipot a cable with a known interconnect pattern. This function puts the tester in a known state by clearing all relays, sinking all points, and then turning on the relay for the input point. All unused points are grounded. If optional parameters are given, the high voltage settings in the wirelist will be overridden and the optional parameters will be used. The voltage, insulation resistance, and duration must be given if using the optional parameters. If no optional parameters are given, the high voltage settings in the loaded wirelist will be used. The function will return two results: a number indicating the status of the test and the measured leakage resistance. If the function is called without any parameters, it will return a description of itself. This function will do a hipot test whether or not high voltage testing is turned on in the loaded wirelist.

Format *(Standard Settings):*

```
iResult, fResisMeasured = HipotNetsTiedToPoints(sPoints
                                   [,fVoltage, fInsulRes, fDuration
                                   [, fMaxSoakTime]])
                   ↑                 ↑                        ↑
                RESULT           FUNCTION                   INPUTS
```

INPUTS	INPUT DESCRIPTION	RESULT1 `iResult`	RESULT2 `fResisMeasured`
`sPoints`	String containing the test points on the net to be hipotted.	Integer containing one of the following codes: **0** = Passed Test **1** = Has leakage **2** = Overcurrent **3** = Dielectric Failure **99** = Other Failure **100** = Invalid Test Point **101** = * Invalid Voltage **102** = * Invalid Insulation Resistance **103** = * Invalid Hipot Duration **104** = * Invalid Soak Time * = If these errors occur, the hipot test will not be performed.	Float containing the measured leakage resistance **or** nil for some RESULT1 errors (2, 3, 99)
`fVoltage` (**optional**)	Float containing the voltage applied during the hipot test. Range: 50 – 1000 volts.		
`fInsulRes` (**optional**)	Float containing the insulation resistance. It is the minimum resistance allowed between unintended connections. Values: 5,000,000 – 1,000,000,000 Ω		
`fDuration` (**optional**)	Float containing the duration of the hipot. Range: 0.01 – 120 seconds		
`fMaxSoakTime` (**optional**) (Set to zero if not included with the other optional parameters.)	Float containing the maximum amount of time the hipot voltage is applied to the test point after the DWV portion of the test and before the IR test. Soak time stops when the measured insulation resistance exceeds `fInsulRes` for the duration of the IR test. Values: 0(off), 0.01 - 120 seconds		

Format *(Advanced DC Settings):*

```
iResult, fResisMeasured = HipotNetsTiedToPoint(sPoints, “DC”,
                               fDWVVoltage, fDWVCurrent,
                               iDCDuration, fIRVoltage,
                               fIRInsulRes, fTimeGoodFor, fSoak,
                               iSoakUntilGood)
           ↑                ↑                       ↑
        RESULTS          FUNCTION                 INPUTS
```

INPUTS	INPUTS DESCRIPTION	RESULT1 `iResult`	RESULT2 `fResisMeasured`
`sPoints`	String containing the test points on the net to be hipotted.	**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 **114** = * Invalid DWV DC 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.	Float containing the measured leakage resistance **or** nil for some RESULT1 errors (2, 3, 99)
`“DC”`	String indicating using advanced DC settings.		
`fDWVVoltage`	Float containing the voltage applied during the Dielectric Strength Test. Range: 50 – 1500 V		
`fDWVCurrent`	Float containing the maximum current that can flow through insulation during hipot test. Range: .1mA – 1.5 mA		
`iDCDuration`	Any float containing the time in seconds to voltage is applied to each net during the DWV test. Range: 0.01 - 120 seconds		
`fIRVoltage`	Float containing the voltage applied during the Insulation Resistance Test. Range: 50 – 1500 V		
`fIRInsulRes`	Float containing the insulation resistance. It is the minimum resistance allowed between unintended connections. Range: 5,000,000 - 1,000,000,000 Ω		
`fTimeGoodFor`	Float containing the time for the duration of the Insul. Res Test after the Soak Time. Range: .002 – 120 sec.		

Continued on the next page

HipotNetsTiedToPoints, *continued*

`fSoak`	Float containing the time before starting the Insul Res Test. Depends on iSoakUntilGood value. If iSoakUntilGood = off, tester waits this time before starting test. If iSoakUntilGood = on, tester up to this time before reporting an error. Range: .002 – 120 sec		
`iSoakUntilGood`	**0** (off) = Use fSoak to set a definite time before starting the Insul Res Test. **OR** **1** (on) = Use fSoak to set the longest time to wait for acceptable insul current leakage before reporting an error.		

Format *(Advanced AC Settings)*:

```
iResult, fResisMeasured = HipotNetsTiedToPoint(sPoints, “AC”,
                              iFrequency, fDWVoltage,
                              fDWVCurrent, iACDuration,
                              fIRVoltage, fIRInsulRes,
                              fTimeGoodFor, fSoak,
                              iSoakUntilGood)
          ↑                   ↑                     ↑
        RESULT             FUNCTION               INPUTS
```

INPUTS	INPUT DESCRIPTION	RESULT1 `iResult`	RESULT2 `fResisMeasured`
`sPoints`	String containing the test points on the net to be hipotted.	**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.	Float containing the measured leakage resistance
`“AC”`	String indicating using advanced AC settings.		
`iFrequency`	Integer containing the frequency used for the DWV test. Range: 25 - 60 Hz.		
`fDWVVoltage`	Float containing the voltage applied during the DWV test. Range: 50 – 1500 V		
`fDWVCurrent`	Float containing the maximum current that can flow through insulation during hipot test. Range: .1mA – 1.5 mA		

Continued on the next page

HipotNetsTiedToPoints, *continued*

`iACDuration`	Integer containing number of cycles the voltage is applied to each net of points during the DWV test. Values: 1 – 7200 cycles		
`fIRVoltage`	Float containing the voltage applied during the Insulation Resistance Test. Range: 50 – 1500 V		
`fIRInsulRes`	Float containing the insulation resistance. It is the minimum resistance allowed between unintended connections. Values: 5,000,000 - 1,000,000,000 Ω		
`fTimeGoodFor`	Float containing the time for the duration of the Insul Res Test after the Soak Time. Range: .002 – 120 sec		
`fSoak`	Float containing the time before starting the Insul Res Test. Depends on iSoakUntilGood value. If iSoakUntilGood = off, tester waits this time before starting test. If iSoakUntilGood = on, tester up to this time before reporting an error. Range: .002 – 120 sec		
`iSoakUntilGood`	**0** (off) = Use fSoak to set a definite time before starting the Insul Res Test. **OR** **1** (on) = Use fSoak to set the longest time to wait for acceptable insul current leakage before reporting an error.		

HipotNetsTiedToPoints, *continued*

Example:

```
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 the wirelist parameters. A high voltage warning message will display during the hipot. If an error occurs, a message box will display the error.

HipotPointMask

Explanation:

This function works for software version 3.26 or greater. Use this function to add or subtract test points for the hipot test. If no input parameters are given, the function will return a string containing a list of all points to hipot.

Format:

```
sPointsToHipot, sPointsAccepted = HipotPointMask(iCommand [,
                                                  sTestPoints])
             ↑                                  ↑          ↑
          RESULTS                           FUNCTION     INPUTS
```

INPUT1 `iCommand`	INPUT2 **(not needed for command 0 & 1)** `sTestPoints`	RESULT1 `sPointsToHipot`	RESULT2 `sPointsAccepted`
Integer containing which hipot mask command to run where: **0** = Points to subtract from hipot list **1** = Points to add to hipot list **2** = Subtract all points from hipot list **3** = Add all points to hipot list	String containing the test points for command numbers 0 and 1.	String containing the points to hipot.	String containing the points accepted.

Examples:

-
  ```
  local OriginalPtsToHipot = HipotPointMask() -- get list of all points to hipot
  local AllPtsToHipot, temp1 = HipotPointMask(3) -- add all points to hipot
  local PointsToHipot, temp2 = HipotPointMask(0, PointsToRemove) -- remove these pts
  if (temp1 ~= nil) or (temp2 ~= nil) then
    MessageBox("Invalid points for HipotPointMask function:\n "..temp1.." "..temp2)
  end
  ```

  This function shows how to exclude certain points from Hipot. The Hipot point list did contain the points in the string `OriginalPtsToHipot`. The Hipot point list was reset to `AllPtsToHipot` before removing points contained in `PointsToRemove`. Finally the Hipot point list was set to contain the points in `PointsToHipot`.

-
  ```
  local OriginalPtsToHipot = HipotPointMask() -- get list of all points to hipot
  local AllPtsRemoved, temp1 = HipotPointMask(2) -- remove all points from hipot
  local PointsToHipot, temp2 = HipotPointMask(1, PointsToAdd) -- add these pts
  if (temp1 ~= nil) or (temp2 ~= nil) then
    MessageBox("Invalid points for HipotPointMask function:\n "..temp1.." "..temp2)
  end
  ```

  This function shows how to Hipot only certain points. The Hipot point list did contain the points in the string `OriginalPtsToHipot`. The Hipot point list was then reset to `AllPtsRemoved.` Finally, the points contained in `PointsToHipot` were added as the list of points to Hipot.

IsSPCDataCollectionOn

Explanation:

Use this function to determine if SPC Data Collection is on for the current wirelist. There are no inputs to this function. This function does not return the type of SPC data to be collected.

Format:

```
iDataCollectionOn = IsSPCDataCollectionOn()
       ↑                    ↑
     RESULT              FUNCTION
```

RESULT `iDataCollectionOn`
0 = Integer value indicating SPC is **off** in the current wirelist **1 =** Integer value indicating SPC is **on** in the current wirelist

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 current wirelist.

LearnCable

Explanation:

Use this function to learn a cable. The cable is learned as a child wirelist, saved to disk in the current directory, and then purged from memory. If no filename is given for the learned wirelist, the function will use "untitled.wir" for the filename. The filename for the learned cable cannot be the same name as the wirelist that is already in memory. To find the current directory, use the script function, `GetWirelistInfoAsText(2)`or `DirUtils()`. Use the function `UseChildWirelist` to operate on the learned wirelist.

If no optional parameters are given, the cable will learn with the following options:

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

All input parameter text must be in English, foreign language is not supported. Each input must have a corresponding input number along with the parameter text.

The function will return two results: a text string containing messages and a text string containing errors. If the function has no input, it will return a description of itself.

Format:

```
sResultMessage, sErrorText = LearnCable([1, sFilename,]
                                [2, sCableDescription,]
                                [3, sTestOptionsText,]
                                [4, iComponentsToLearn,]
                                [5, iDataCollectionOpts,]
                                [6, sEventScriptName,]
                                [7, sCustomComponentScriptName,]
                                [8, iLearnIfNoConnectionsOption])
          ↑                  ↑              ↑
       RESULTS           FUNCTION        INPUTS
```

INPUTS	INPUT DESCRIPTIONS	RESULT1 `sResultMessage`	RESULT2 `sErrorText`
`1, sFilename` (**optional**)	String containing the name for the learned cable. This filename cannot be the same name as the wirelist currently in memory.	**Nil** is returned if the cable learned **or** a string containing the following text: "Error learning cable"	**Nil** is returned if cable learned **or** a string containing either of the following errors: "Bad option #3" or "Error saving learned wirelist to disk."
`2, sCableDescription` (**optional**)	String containing the cable description.		

Continued on the next page

LearnCable, *continued*

`3, sTestOptionsText` (**optional**)	String containing the low voltage and high voltage text from the wirelist without the CRC and option signatures.		
`4, iComponentsToLearn` (**optional**)	Integer containing the components to learn where: 1 = resistor 2 = capacitor 4 = diode 8 = twisted pair 11 = all but diodes 15 = all		
`5, iDataCollectionOpts` (**optional**)	Integer containing the options for Data Collection where: 1 = Summary Data 2 = Measured Values 4 = Error Text **Note:** Options 2 & 4 contain Summary Data.		
`6, sEventScriptName` (**optional**)	String containing the filename for the event script		
`7,sCustomCmpScriptName` (**optional**)	String containing the filename for the custom component script		
`8,iLearnIfNoConnections` (**optional**)	Integer other than zero will learn even if no connections are found		

Examples:

- `sResultMessage, sErrorText = LearnCable(1,“batch1.wir”, 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 and 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”)`

  This example will learn the cable and save the wirelist as untitled.wir in the current directory. The cable description will be “last learned”. The cable will be learned with the supplied low and high voltage settings.

LoadWirelist

Explanation:

Use this function to load a wirelist by filename The filename should not contain an extension or directory path. The extension, .wir, is automatically appended to the file. The wirelist file must exist in the same directory where the Touch1 is currently running. To find the current directory, use the script function, `GetWirelistInfoAsText(2)`or `DirUtils()`.

If the wirelist loads correctly, the function ends in the test setup screen. If the wirelist does not exist, a message box containing "No Match Found For Filename" will be displayed and the function ends in the “Enter Filename To Find” screen for FAST FIND.

Note: 1100 testers do not support this function.

Format:

```
LoadWirelist(sWirelistName)
    ↑               ↑
 FUNCTION         INPUT
```

INPUT	DESCRIPTION
`sWirelistName`	A string containing the name of the wirelist to be loaded without the extension, .wir and the path.

Example:

```
sWirelist = "test1"
LoadWirelist(sWirelist)
```

This example will load the wirelist, "test1.wir".

log

Explanation:

The function returns the natural logarithm (base e) of x.

Format:

```
fValue = log(fInputNum)
  ↑        ↑       ↑
RESULT  FUNCTION  INPUT
```

INPUT `fInputNum`	RESULT `fValue`
Call `log` to find the natural logarithm of the float, `fInputNum`.	Float containing the natural logarithm (base e) of `fInputNum`.

Examples:

-
  ```
  fInputNum = 7.93
  fValue = log(fInputNum)
  ```

  The function will return `myValue` equal to 2.071.

- `fLogResult = log(19.1)`

  The function will return `fLogResult` equal to .9555.

log10

Explanation:

The function returns the logarithm (base 10) of x.

Format:

```
fValue = log10(fInputNum)
  ↑         ↑       ↑
RESULT   FUNCTION  INPUT
```

INPUT `fInputNum`	RESULT `fValue`
Call `log10` to find the logarithm of the float, `fInputNum`.	Float containing the logarithm (base 10) of `fInputNum`.

Examples:

-
  ```
  fInputNum = 7.93
  fValue = log10(fInputNum)
  ```
  The function will return `fValue` equal to .8993.

-
  ```
  fResult = log10(3.42)
  ```
  The function will return `fResult` equal to .5340.

LowLevelCmd

Explanation:

Use these functions to execute low level commands on the Touch1. These commands are designed for use by the very experienced programmer/engineer to extend the Touch1's testing methods. 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 LowLevelCommand(9) to return the tester to a known state. This is the appropriate cleanup function.

CRITICAL NOTE: These functions do not allow you to put more than 6 volts anywhere on the tester. If you break this rule, you will void the tester's warranty. The tester cannot damage itself so if you don't connect any outside power supplies to the tester, it is impossible for the commands listed below to damage the tester.

Format:

```
Result, Errnum = LowLevelCmd(iFunctionNum, [parameters])
       ↑             ↑                  ↑
    RESULTS       FUNCTION            INPUTS
```

The INPUT, i`FunctionNum`, contains the index number of the low level command to execute. The INPUT, parameters, contains any additional inputs needed to execute that low level command. The RESULTS, `errnum`, will be "nil" if there was no error and a two if an invalid index number is passed into the function. If the command does not have a return value, `Result` will be zero.

LowLevelCmd, *continued*

Individual Function Number Tables:

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.

INPUT1 `iFunctionNum`	INPUT2 `PointToSink`	INPUT3 `iSinkIt`	RESULT1 `Result`	RESULT2 `ErrNum`
`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.

TurnOnRelay

Why would you turn on a relay? The 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:

- 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.
- You cannot directly measure the voltage on a point when the relay is turned on for that point.
- 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.

INPUT1 `iFunctionNum`	INPUT2 `PointToTurnOnRelay`	INPUT3 `iRelayState`	RESULT1 `Result`	RESULT2 `ErrNum`
`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.

Source/ClearSourceVector

This 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.

Notes:

- 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.

INPUT1 `iFunctionNum`	INPUT2 `PointToSource`	INPUT3 `iSourceIt`	RESULT1 `Result`	RESULT2 `ErrNum`
`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.

Read/ClearReadVector

This function 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.

INPUT1 `iFunctionNum`	INPUT2 `PointToRead`	INPUT3 `iReadIt`	RESULT1 `Result`	RESULT2 `ErrNum`
`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 #

SetCurrent

This 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.

INPUT1 `iFunctionNum`	INPUT2 `iCurrent`	RESULT1 `fResult`	RESULT2 `iErrNum`
`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

MeasureVoltage

The greatest accuracy for this function 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.

INPUT1 `iFunctionNum`	INPUT2 `High Point & Low Point`	RESULT1 `fResult`	RESULT2 `ErrNum`

`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.

LowLevelCmd, *continued*

RouteCurrentToRelay

Use this command if you want to perform fourwire 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.

INPUT1 `iFunctionNum`	INPUT2 `iWhereToSendIt`	RESULT1 `iResult`	RESULT2 `ErrNum`
`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.

MasterClear

This function unsinks and turns 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).

INPUT1 `iFunctionNum`	RESULT1 `iResult`	RESULT2 `ErrNum`
`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.

SetAllDefault

This function resets the current source in addition to what MasterClear does. 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.

INPUT1 `iFunctionNum`	RESULT1 `iResult`	RESULT2 `ErrNum`
`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.

LowLevelCmd, *continued*

SetHighCurrent

This function sets 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.

INPUT1 `iFunctionNum`	INPUT2 `fCurrent`	RESULT1 `Result`	RESULT2 `ErrNum`
`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.

LuaError

Explanation:

This function is used to handle errors that would normally cause Lua to stop executing such as passing an invalid parameter to a function. If your script contains this function and an error occurs, the tester will call LuaError and pass it a string containing the error message. After LuaError exits, the script will continue to execute. If your script does not contain the LuaError function and an error occurs, a dialog box will still display the error message but the script will stop executing.

This function is useful if you accept information from the tester operator and they enter bad information that you use as a parameter in a function call. If they enter bad information and you pass it to a function listed in this manual, an error could occur that would require the use of LuaError.

Note: DO NOT call LuaError directly in functions yourself. This is a separate function written in the script that the tester calls only if there is an error in the script.

Format:

```
LuaError(sErrorText)
  ↑         ↑
FUNCTION   INPUT
```

INPUT	DESCRIPTION
`sErrorText`	A string containing the error message to be displayed in a dialog box with a CANCEL button.

Example:

-
  ```
  function LuaError(sErrorText)
     MessageBox("Error: " .. sErrorText .. " occurred. ")
  end
  ```

  In this example, if a catastrophic error occurs in your script, the error message will be displayed in a dialog box. After you press the “Cancel” button in the dialog box, the rest of your script will continue to execute instead of exiting due to the error.

max

Explanation:

The function returns the value of the largest input number. It can take an unlimited number of input numbers but there must be at least two.

Format:

```
myValue = max(myInputNum1, myInputNum2 [, myInputNum3, ...])
  ↑         ↑                    ↑
RESULT   FUNCTION              INPUTS
```

INPUTS `myInputNum1` `myInputNum2, ...`	RESULT `myValue`
At least two input numbers the maximum value will be computed on.	Contains the value of the largest input number.

Example:

```
fValue = max(7.93, 2.54, 5, 1)
```

The function will return `fValue` equal to 7.93 because 7.93 is the largest number.

MessageBox

Explanation:

Use this function to display a message box on the tester screen containing custom text and up to six custom buttons. The function returns the number of the pressed button. **Unlike dialog boxes, the tester and the script do not continue to run when a message box is displayed.**

A message box will have a default CANCEL button if no other custom button is created. Use the function DialogOpen() if no CANCEL button is needed.

For the 1100 testers, message boxes replace the entire screen. For the Touch1, message boxes were designed so part of the touch screen is always displayed and the message box is centered on the screen. The message box automatically sizes to fit the required custom text and the buttons automatically align.

For the Touch1, the buttons do not automatically size to fit the button text. The touch screen is 320 x 240 pixels and a character is 8 x 6 pixels. A button cell is 32 x 30 pixels which allows four characters per row & five characters per column. MessageBox buttons are three button cells wide so eleven characters fit into one button cell.

Format:

```
iPressedButtonNum = MessageBox ([iFromColor, iToColor,]
                                sMessageText [, sButtonText1]
                                [, sButtonText2] [, ButtonText3]
                                [, sButtonText4]
                                [, ButtonText5][, sButtonText6])
       ↑                ↑                  ↑

     RESULT          FUNCTION           INPUTS  ( [ ] = optional )
```

INPUTS	INPUT DESCRIPTIONS	RESULT `iPressedButtonNum`
`iFromColor,` `iToColor` **(optional pair, Touch1 only)** **To see color, a monitor must be attached to the Touch1.**	where: 0 = BLACK 1 = BLUE 2 = GREEN 3 = CYAN 4 = RED 5 = MAGENTA 6 = BROWN 7 = LIGHTGRAY 8 = DARKGRAY 9 = LIGHTBLUE 10= LIGHTGREEN 11 = LIGHTCYAN 12 = LIGHTRED 13 = LIGHTMAGENTA 14 = YELLOW 15 = WHITE	Integer containing the number of the button that was pressed.
`sMessageText`	String containing the text for message box. You cannot input arrays, only strings are allowed.	
`sButtonText1` – `sButtonText6` **(optional)**	**Optional** string containing the text for custom buttons 1-6. Text should be in quotes.	

MessageBox, *continued*

Miscellaneous Prompt Box Syntax

- **String Concatenation:**

Use "**..**" to concatenate strings and/or variables together for message text.

- **Titles:**

You can define titles to add to dialog and message boxes. For 1100 testers, the title will be centered on the top line. The codes are as follows:
Start Title = ~
Stop Title = ~
The title can be embedded in the message string as follows: sMsg = "~title~message"

Example:

```
local iBtnNum = MessageBox("~Serial Number Entry Error~" .. "Serial numbers
must be numeric.", "Cancel")
```

This example displays a message box with a Cancel button and a title "Serial Number Entry Error". The text of the message box is "Serial numbers must be numeric".

There are several control sequences to enhance dialog and message box displays. They are as follows:

- **Hot keys:**

You can define *ShowHotKey* and *HideHotKey* to add hot keys (keyboard shortcuts) to dialog and message boxes. The codes are as follows:
Show Hot Key = <K *x*> where *x* is the letter of the hot key **or** use the code: "\26\02\01"
Hide Hot Key = <KH *x*> where *x* is the letter of the hot key **or** use the code: "\26\02\02"
Cancel Key = \27
Enter Key = \13

Example:

```
ShowHotKey = "\26\02\01"
sMessage = "Hot Key Test"
local iBtnNum = MessageBox(sMessage, "S" .. ShowHotKey "kip","<KH \27>Cancel")
```

This example displays a message box with Skip and Cancel buttons. The "S" key is a defined hot key that activates when pressed. The ESC is not a hot key so will not activate when pressed. The ".." means concatenate the text together as one string.

- **Underlining:**

You can define *StartUnderline* and *StopUnderline* to add underlining to dialog and message boxes. The codes are as follows:
Start underline = <U> **or** "\25\01\01"
Stop underline = **<\\U>or** "\25\01\02"

Example:

```
StopUnderline = "\25\01\02"
MessageBox("No Underline" .. "<U>Underlined" ..StopUnderline ..
          "No Underline", "CANCEL")
```

This example displays a message box with a Cancel button. The text "No Underline" is not underlined followed by the text "Underlined" which is underlined and "No Underline" is not underlined. The ".." means concatenate the text together as one string.

- **Font Changes:**

You can define fonts to display text for dialog and message boxes in different touch1 fonts. The codes are as follows:
Big Font = <B>
Big Font with multiplier = <B 2> **or** <B 3> where <B 2> is twice & <B3> is triple the size of <B>
Small Font = <S>
Small Font with multiplier = <S 2> **or** <S 3> where <S 2> is twice & <S3> is triple the size of <S>
Very Big Font = "\28\02\03\01"

Example:

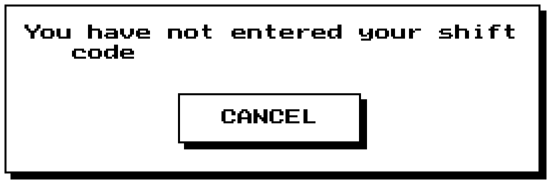
```
VeryBigFontCode = "\28\02\03\01"
local iBtnNum = MessageBox("Regular Font" .. VeryBigFontCode ..
                           "Very Big Font" .. "<S>Small Font", "Cancel")
```

This example displays a message box with a Cancel button. The text "Regular Font" is in the normal Touch1 big font. The text "Very Big Font" is in very big font and then the text "Small Font" is in the Touch1's smallest font.

MessageBox, *continued*

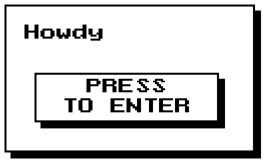
Examples:

- `iButtonNum = MessageBox(“You have not entered your shift code.”)`



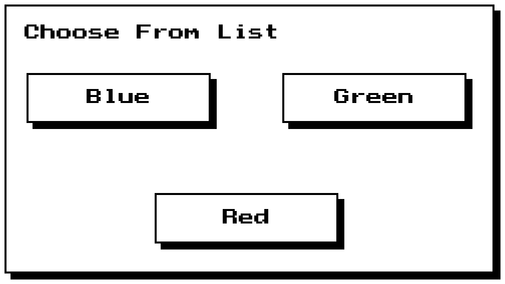
This message box is sized around the text. It has a default "CANCEL" button. If the user pressed "CANCEL", the function would return a 1 and store it in `iButtonNum` since the "CANCEL" button is the only button.

- `MessageBox("Howdy", "PRESS\nTO ENTER")`



This message box uses the default location and automatic sizing. It has a custom button with the text on two lines. The “\n” is a linefeed.

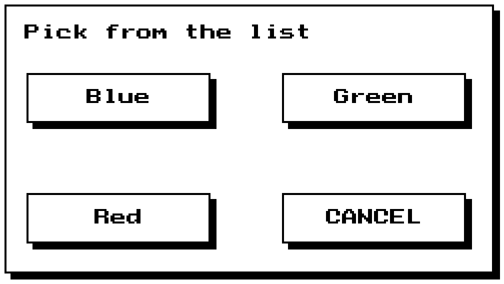
- iButtonNum = MessageBox("Choose From List", "Blue”, “Green", "Red")



This message box uses the default location and automatic sizing. It has three buttons: “Blue”, "Green", and “Red". If the user pressed "Red", the function would return a 3 and store it in `iButtonNum` since the "Red" button is the third button.

MessageBox, *continued*

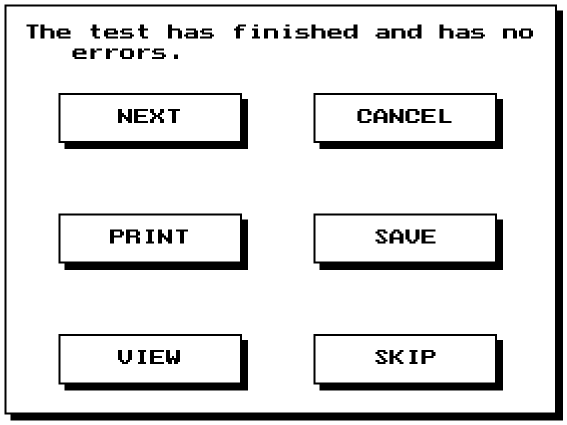
-
  ```
  sButtonName1 = "Blue"
  sButtonName2 = "Green"
  sButtonName3 = "Red"
  iButtonNum = MessageBox("Pick from the list", sButtonName2,
                          sButtonName3, sButtonName1, "CANCEL")
  ```



```
Pick from the list
```

This message box uses the default location and automatic sizing. It has four buttons: “Blue”, "Green", "Red", and "CANCEL". If the user pressed "Red", the function would return a 3 and store it in `iButtonNum` since the "Red" button is the third button.

-
  ```
  sMessageText = "The test has finished and has no errors."
  sButtonText1 = "NEXT"
  sButtonText2 = "CANCEL"
  sButtonText3 = "PRINT"
  sButtonText4 = "SAVE"
  sButtonText5 = "VIEW"
  sButtonText6 = "SKIP"
  iPressedButtonNum = MessageBox(sMessageText, sButtonText1,
                                 sButtonText2, sButtonText3,
                                 sButtonText4, sButtonText5,
                                 sButtonText6)
  ```



```
The test has finished and has no
   errors.
```

This message box has six custom buttons: "NEXT", "CANCEL", "PRINT", "SAVE", "VIEW", and "SKIP". If the user pressed "NEXT", `iPressedButtonNum` would return a 1 since the "NEXT" button is the first button.

MicroLan

Explanation:

Use this 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

Format:

```
MyResult = MicroLan(iInputNum, [parameters])
   ↑          ↑                ↑
 RESULT    FUNCTION          INPUT
```

INPUT `iInputNum & [parameters]`	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.	Returns 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:

-
  ```
  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
  ```

  This 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, *continued*

-
  ```
  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
  ```

  This 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.

min

Explanation:

The function returns the value of the smallest input number. It can take an unlimited number of input numbers, but there must be at least two.

Format:

```
myValue = min(myInputNum1, myInputNum2, myInputNum3, ...)
  ↑        ↑                    ↑
RESULT   FUNCTION             INPUTS
```

INPUTS `myInputNum1, myInputNum2, ...`	RESULT `myValue`
At least two input numbers the minimum value will be computed on.	Contains the minimum value of all the input numbers.

Example:

```
fMinValue = min(3.1, 2.0, 10.32, 5.0)
```

The function will return `fMinValue` equal to 2 because 2 is the smallest of the group of input numbers.

mod

Explanation:

The function computes the floating-point remainder of `myX/myY`.

Format:

```
fValue = mod(myX, myY)
  ↑       ↑      ↑
RESULT FUNCTION INPUTS
```

INPUTS `myX, myY`	RESULT `fValue`
Numbers comprising the fraction you want to find the remainder on.	Contains the floating-point remainder of `myX/myY`.

Example:

`fValue = mod(4.5, 2.0):` The function will return `fValue` equal to 0.5.

outtextxy

Explanation:

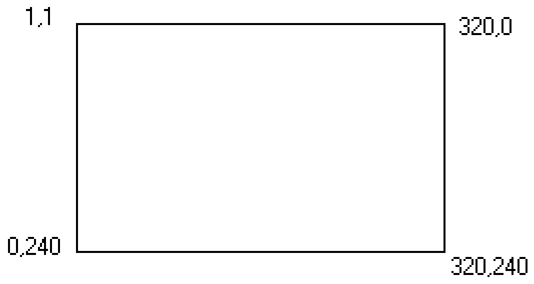
Use this function when you want to display text on the touch screen:

- without using a message box that requires a CANCEL button.
- without using dialog box functions that can have CANCEL button.
- without using the print function where text can scroll off the screen.

This function is especially useful for debugging. It is necessary to call this function every time variables are updated.

The integer values, `iXCoordinate` and `iYCoordinate` denote the starting x and y pixel positions where the text will be placed on the touch screen. A touch screen is 320 x 240 pixels where a character is 8 x 6 pixels and a button cell is 32 x 30 pixels. Attaching a monitor to the tester can expand these coordinates to the size of the monitor. Use the function `tostring` to convert variables to a string and then display using this function.

Note: 1100 testers do not support this function



Format:

```
outtextxy(iXCoordinate, iYCoordinate, sDisplayText)
   ↑                          ↑
```
FUNCTION INPUTS

INPUTS	INPUT DESCRIPTIONS	RESULT
`iXCoordinate`	Integer indicating the tester's x pixel position for the text.	`The text contained in sDisplayText` is displayed on the touch screen at the given x and y pixel coordinates, `iXCoordinate` and `iYCoordinate`.
`iYCoordinate`	Integer indicating the tester's y pixel position for the text.	
`sDisplayText`	String containing the text to display on touch screen.	

Example:

```
• outtextxy( 9, 8,"                        ")
  outtextxy( 9, 9," ********************** ")
  outtextxy( 9,10," *                    * ")
  outtextxy( 9,11," *  RELAY TEST FAILED  * ")
  outtextxy( 9,12," *                    * ")
  outtextxy( 9,13," ********************** ")
  outtextxy( 9,14,"                        ")
  SetDelayTimeInMilliseconds(1000)
```

The text, `RELAY TEST` FAILED will be centered on the touch screen.

PlaySound

Explanation:

Use this function to play a sound. The volume is dependent upon the Touch1's volume setting under System Setup. For a maximum sound, set the Touch1 volume at its maximum setting.

Format:

```
PlaySound(iPitch, iDuration, iVolume)
   ↑                  ↑
FUNCTION            INPUTS
```

INPUT	DESCRIPTION
`iPitch`	An integer indicating the pitch of the sound in Hertz.
`iDuration`	An integer indicating the duration of the sound in seconds.
`iVolume`	An integer from 1 to 100 indicating the volume of the sound where 100 is the loudest. **Note:** This parameter is dependent upon the Touch1's volume setting under System Setup.

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.

print

Explanation:

Use this function to quickly display a value for debugging and error messages. You cannot position the text using this function as you can with the `outtextxy` function. The `print` function causes the text on the touch screen to scroll if there is more text than fits on the screen. It is recommended a monitor be connected to the Touch 1. With a monitor attached the screen displays in the upper left-hand corner. Use the functions `SendTextToParallelPrinter`, `outtextxy`, and `WriteToSerial` if you need formatted text to output.

Format:

```
print(myValue1, myValue2, myValue3, ...)
  ↑              ↑
FUNCTION       INPUTS
```

INPUT	DESCRIPTION
`myValue1, myValue2, etc.`	Values to be printed out for quick debugging.

Example:

-
  ```
  myValue = "500"
  print(10, "Big Red", "HOWDY", myValue)
  ```
  This example displays the following:
  ```
  10
  Big Red
  HOWDY
  500
  ```

PrintLabel

Explanation:

Use this function to print a custom label.

Format:

PrintLabel ("label definition tag")

INPUT1	INPUT2 (optional)	INPUTS DESCRIPTION	RESULT
`Label Definition Tag`	Custom1, Custom2, etc. Refer to pages 40-41.	String corresponding to the label definition sequence for the label required.	This function does not return a result.

Examples:

Note: The following example was automatically created by the Label Report Script Generation Utility (see page 40-41).

```
-- $** USES_LABELS **$

function DoCustomReport()
local scablesignature=GetWirelistInfoAsText(4)

  PrintLabel("LABEL1",scablesignature,"signature")
end

-- $** LABEL_DEFINITION **$
-- $FORMAT: 01$
-- $COMMENT: C:\Temp\Zebra folder\T300 proof.prn
-- $LABEL: LABEL1$
-- 5E58417E54413030307E4A534E5E4C54305E4D4E575E4D54545E504F4E5E
-- 504D4E5E4C48302C305E4A4D415E5052322C325E4D44305E4A55535E4C52
-- 4E5E4349305E585A0D0A5E58410D0A5E4D4D540D0A5E4C4C303430360D0A
-- 5E50573630390D0A5E4C53300D0A5E465436342C34395E41304E2C32382C
-- 33365E46485C5E464442696C6C20616E642054656427732045786365656C6C
-- 656E74204361626C65735E46530D0A5E4259322C332C3136305E46543639
-- 2C3331315E42434E2C2C592C4E0D0A5E46443E3A##0120##023E37373737
-- 345E46530D0A5E465437372C3130345E41304E2C32382C32385E46485C5E
-- 4644##C15E46530D0A5E46543239322C3130335E41304E2C32382C32385E
-- 46485C5E4644##C25E46530D0A5E5051312C302C312C595E585A0D0A##FF
```

Label Definition Tag

PromptForUserInformation

Explanation:

Use this function to create a prompt window on the Touch 1. The window is an alphanumeric, a numeric, or a password prompt. The password window hides the input by displaying asterisks and otherwise acts like the alphanumeric window.

All screens come automatically with DELETE, ← (cursor left), → (cursor right), CLEAR, OK, and CANCEL buttons. The alphanumeric screen also comes with a dash "–", period, underscore, space, and colon buttons. The numeric screen only accepts integers. Use the function *tonumber* to convert a variable to an integer. You set the prompt message and the maximum number of characters to enter in the entry box. You can also supply an optional default value that initially displays in the entry box. The function returns the text the user enters in response to the prompt. The response can be used as the input for functions that output data.

When using this function with the SPC (Statistical Process Control) Data collection feature, the response is called a user-defined field. Examples of user-defined fields include: company name, part number, shift code, revision number. This user-defined data can be saved to SPC data collection using the function, `SaveSPCData`.

Format:

```
sPromptResponseText = PromptForUserInformation(iPromptType,
                                               sFirstLineText,
                                               sSecondLineText,
                                               iMaxChars
                                               [, defaultValue])
        ↑                      ↑                      ↑
      RESULT                FUNCTION            INPUTS  ( [ ] = optional )
```

INPUTS	INPUTS DESCRIPTION	RESULT `sPromptResponseText`
`iPromptType`	Integer containing the type of prompt box where: **1** = alphanumeric **2** = numeric (integers) **5** = alphanumeric password where*'s hide input	A string containing the text the user enters. **or** **nil** = User presses the CANCEL button without entering a value. Always test for CANCEL and replace nil with a valid string so subsequent routines will not break when they expect a string.
`sFirstLineText`	String containing text for the prompt that will display on first line of title. Can be up to 30 characters.	
`sSecondLineText`	String containing additional text that will display on second line of title. Can be up to 30 characters.	
`iMaxChars`	Integer containing the maximum number of characters the user can enter. Range: 1-30	

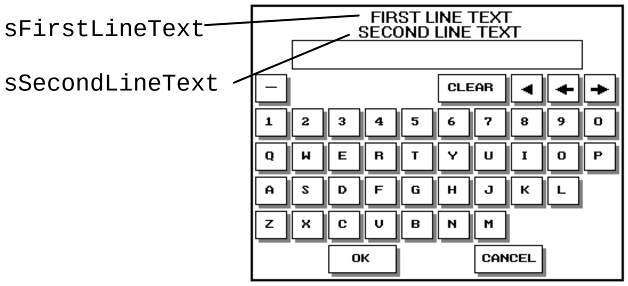
`DefaultValue` **(optional)**	A string or integer depending on iPromptType that contains an initial value for entry box. This input will display upon entry to the prompt box.	

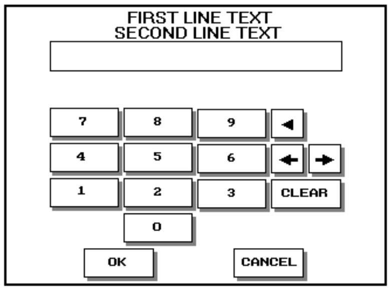
PromptForUserInformation, *continued*

Prompt Type Screens:

`iPromptType`: Alphanumeric(1) `or` Password(5)

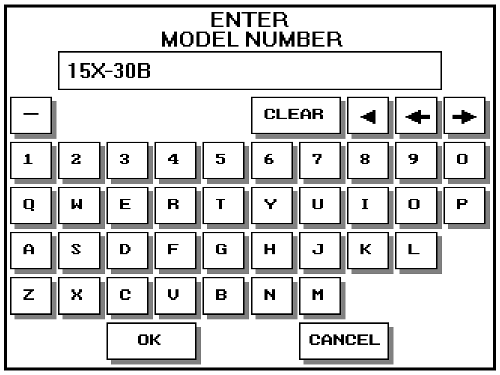
`iPromptType`: Numeric(2)





Examples:

-
  ```
  iPromptType = 1
  sFirstLineText = "ENTER"
  sSecondLineText = "MODEL  NUMBER"
  iMaxChars = 30
  sDefaultValue = "15X-30B"
  sPromptResponseText = PromptForUserInformation(iPromptType,
                                                  sFirstLineText,
                                                  sSecondLineText,
                                                  iMaxChars, sDefaultValue)
  ```



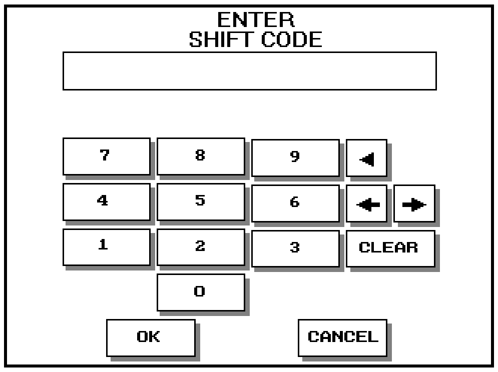
This example displays an alphanumeric prompt box on the Touch 1 screen. The title box displays "ENTER" on the first line and "MODEL NUMBER" on the second line. A default value of "15X-30B" is displayed upon entry to the screen. The user can enter up to 30 characters for the model number or accept the default value.

-
  ```
  iPromptType = 5
  sFirstLineText = "ENTER PASSWORD"
  iMaxChars = 8
  sPromptResponseText = PromptForUserInformation(iPromptType,
                                                        sFirstLineText,
                                                        iMaxChars)
  ```

This example prompts the user for a password on the Touch 1 screen. The title box displays "ENTER PASSWORD" on the first line. Asterisks are displayed on the screen as the user enters the password. The password can be up to eight characters as defined by the variable `iMaxChars`.

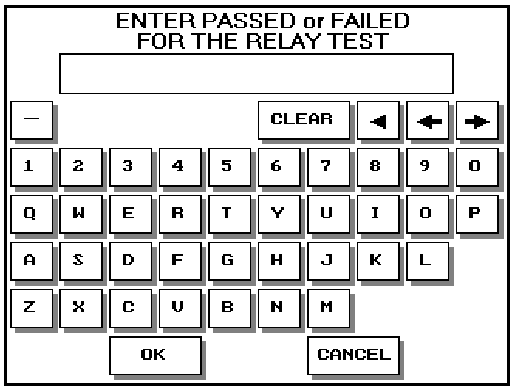
PromptForUserInformation, *continued*

-
  ```
  iPromptType = 2
  sFirstLineText = "ENTER"
  sSecondLineText = "SHIFT CODE"
  iMaxChars = 4
  sPromptResponseText = PromptForUserInformation(iPromptType,
                                                  sFirstLineText,
                                                  sSecondLineText,
                                                  iMaxChars)
  ```



This displays a numeric prompt box on the Touch 1 screen. The title box displays "ENTER" on the first line and "SHIFT CODE" on the second line. No default value is given so the entry box will be empty upon entry to the screen. The user can enter up to four numbers for the shift code as defined by the variable, `iMaxChars`.

-
  ```
  sPromptResponseText = PromptForUserInformation(1, "ENTER PASSED or
                                                 FAILED", "FOR THE RELAY
                                                 TEST", 6)
  ```



This displays a alphanumeric prompt box on the Touch 1 screen. The title box displays "ENTER PASSED or FAILED" on the first line and "FOR THE RELAY TEST" on the second line. No default value is given so the entry box will be empty upon entry to the screen. The user can enter up to six characters for the response.

- `functionDescription = PromptForUserInformation()`

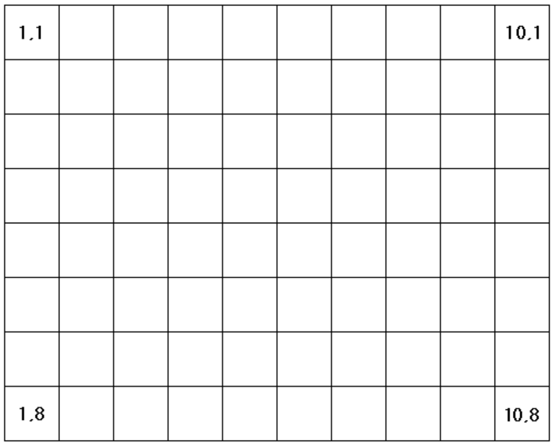
The function returns the function description as text in the RESULT, `functionDescription`.

PushKeyPress

Explanation:

Use this function when you want to control the touch screen without touching the screen. For example, this function can be used to start a script on power up or to retrieve a wirelist. The keypress will be executed at the end of each DoOnTestEvent() or DoIt() loop or at the end of a custom component test. The inputs are defined by hotkey characters or by the touch screen cells. The touch screen is 10 cells wide by 8 cells long. The hotkey characters are the underlined characters displayed on the touch screen.

Note: 1100 testers do not support this function



Format:

```
PushKeyPress(iXCell, iYCell)
   ↑                ↑
FUNCTION         INPUTS
```

INPUTS `iXCell, iYCell`	RESULT
Integer pair containing the row and column of the button to press. 1,1 = upper left cell 10,8 = lower right cell **OR** Text string containing the hotkey character of the button to press. The hotkey is the underlined character on the Touch 1. Return or Enter key = \r ESC key = \27	Activates the button corresponding to the input cells or hotkey character.

Examples:

- `PushKeyPress(1,1,5,7) -- go to main screen then test cable`

  This example will press the home button to go to the main screen (1,1) and then press the test cable button (5,7).
- `PushKeyPress("trfftest7\r")`
  `PushKeyPress("r")`

  This example will press the test setup button and then retrieve the wirelist "test7.wir".
- `PushKeyPress(1,1,"tvhc")`

  This example will go to the main screen (1,1) and then to the hipot settings screen.

random

Explanation:

Use this function to return a pseudo-random integer in the range 0 to 32767. The sequence can be started at different values by calling the `randomseed` function.

Format:

```
iValue = random()
  ↑        ↑
RESULT   FUNCTION
```

RESULT	DESCRIPTION
`iValue`	Contains a random integer in the range 0 to 32767.

Example:

```
iValue = random()
```

The function will return `iValue` equal to an integer from 0 to 32767.

randomseed

Explanation:

This function uses a numeric argument, `mySeedNum`, to start a new sequence of pseudo-random integers to be returned by subsequent calls to random. A particular sequence of pseudo-random integers can be repeated by calling `randomseed` with the same seed value. The default sequence of pseudo-random integers is selected with a seed value of 1.

Note: To get real random numbers, use a concatenated system date and time for the seed.

Format:

```
randomseed(mySeedNum)
   ↑           ↑
FUNCTION     INPUT
```

INPUT `mySeedNum`	RESULT
Contains the value where the random sequence will be started for the function `random`.	The function sets up the starting sequence of random integers for the function, `random`.

Example:

-
  ```
  randomseed(3)
  myValue = random()
  ```

  This example will start a new sequence of random integers at 3 to run with the function `random`.

read

Explanation:

Use this function to read a file according to a read pattern. The file must be opened using the function, `readfrom`. If you call the function without a `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. Use the function, `readfrom`, to close the file when finished reading.

Format:

```
sCharsRead = read([sReadpattern])
     ↑            ↑          ↑
  RESULT       FUNCTION    INPUT
```

INPUT `sReadpattern` **`(optional)`**	DESCRIPTION	RESULT `sCharsRead`
`"."`	This string pattern returns the next character or nil on end of file.	A string containing the characters read **OR** nil = nothing read
`".*"`	This string pattern reads the entire file.	
`"[^\n]*{\n}"` `(default pattern)`	This string pattern 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.	
`"{%s*}%S%S*"`	This string pattern returns the next word skipping spaces if necessary or nil on end of file.	
`"{%s*}[+-]?%d%d"`	This string pattern returns the next integer or nil if the next characters are not integers.	

Example:

-
  ```
  readfrom(myFile)       -- This opens the file for reading
  local tempstr = read() -- This reads the first line of the file
  readfrom()             -- This closes the file
  ```

  This example opens the file, `myFile`. Since no input is given, the default pattern is used. The function reads the first line of the file into a temporary local variable, `tempstr`. The file is then closed using the function `readfrom`.

ReadBlockFromSerial

Explanation:

Use this function to read a block of data from the serial port that was initialized by the function `SetSerialParms`.

Format:

```
sDataRead, iResult = ReadBlockFromSerial([sPortNumber,]
                                         [fFirstCharTimeout,
                                         fTransmitTimeout,]
                                         iBlockSize)
         ↑                   ↑                    ↑
      RESULTS            FUNCTION         INPUTS  ( [ ] = optional )
```

INPUTS	INPUT DESCRIPTIONS	RESULT1 `sDataRead`	RESULT2 `iResult`
`sPortNumber` **(optional)**	A string containing which COM port (“**com1”** or “**com2”**) to use. If the input is not used, the default is “com1”.	nil **OR** Text string containing the data read from the port.	**-999** = timeout **-998** = no port **-997** = port in use **OR** Integer containing the number of characters read
`fFirstCharTimeout` `fTransmitTimeout` **(optional pair)**	Floats containing the timeout values in seconds. The first timeout indicates the time to wait for the first character to be transmitted. The second timeout indicates the time to wait before a timeout during transmission. If the inputs are not used, the First CharacterTimeout default is 1.0 second and the Transmit Timeout is .1 seconds.		
`iBlockSize`	Integer indicating the block size of the data to retrieve. It is used to recognize when the transmission is complete.		

Example:

- `sDataRead, iResult = ReadBlockFromSerial("com1", 2.0, 1.0, 1024)`
  This example reads a 1024 block of data from the "com1" serial port. It waits two seconds before transmitting the first character and one second before timing out during a transmission.

readfrom

Explanation:

Use this function to open the named file or close a file after reading from it. The path should be included with the filename. The function returns the value of the file handle. If the function fails to open the file, it returns a nil plus a string describing the error. When called with a filename, this function **does not** close the current input file upon return. When the function is called without a filename, it closes the file.

Format:

```
canReadFile, sErrMsg = readfrom(sFilename)
         ↑                 ↑         ↑
      RESULTS          FUNCTION    INPUT
```

INPUT `sFilename`	RESULT1 `canReadFile`	RESULT2 `sErrMsg`
String containing the path and filename for the file to be opened. **OR** When the function is called without a filename, it closes the file.	Contains the file handle for the filename **OR** **nil** = open failed	A string describing the error if the function fails such as ”No file or directory” **OR** **nil** = file opened OK

Example:

```
• sFilename = "/DOS/touch1/storage.dat"
  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 storage.dat file. If the function fails, it will return a nil in `canReadFile` and an error string in s`ErrMsg`. If the file opens, the first line will be read into the string, s`FirstLine` using the function, `read`. The second line will be read into the string, `sSecondLine` using the function, `read`. The file is then closed calling the function `readfrom` again.

ReadFromSerial

Explanation:

Use this function to read data from the serial port that was initialized by the function `SetSerialParms`. Use this function along with `WriteToSerial` to read and write to the serial port.

Format:

```
sResult, iErrorNum = ReadFromSerial([sPortNumber],
                                    [sTimeBeforeStart,
                                    fSecBeforeTimeout],
                                    sTerminatingChar)
   ↑              ↑                               ↑
RESULTS        FUNCTION                     INPUTS  ( [ ] = optional )
```

INPUTS	INPUT DESCRIPTIONS	RESULT1 `sResult`	RESULT2 `iErrorNum`
`sPortNumber` **(optional)**	String containing which COM port to use. “**com1**” or “**com2**” The default is “com1”.	Text string containing the data read from the port.	**-999** = timeout **-998** = no port **-997** = port in use **OR** **0** = no error
`FTimeBeforeStart (default = 1.0)` `fSecBeforeTimeout (default = .01)` **(optional pair)**	Floats containing timeout values in seconds. The first timeout indicates the time to wait for the first character to be transmitted. The second timeout indicates the time to wait before a timeout during transmission.		
`sTerminatingChar`	Character used to recognize when the transmission is complete.		

Example:

- `myResult, ErrNum = ReadFromSerial("com1", 5, 1, "\n")`

  This example reads the data from the "com1" serial port until a linefeed, '\n' is reached. It waits five seconds before transmitting the first character and one second before timing out during a transmission.

ReadUserInputStates

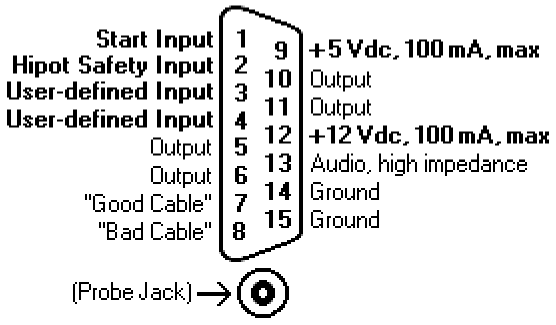
Explanation:

Use this function to get the state of the digital inputs on the user digital I/O port. The settings for External Start Switch and Hipot Safety Switch in the Test Control screen on the Touch 1 affect this function if they are turned on. The function reads the inputs directly from the digital I/O port. You can use any combination of the four inputs and the function will return their respective states, low or high.

Format:

```
iInputState1, iInputState2,
iInputState3, iInputState4 = ReadUserInputStates(iDigitalInput1,
                                                 iDigitalInput2,
                                                 iDigitalInput3,
                                                 iDigitalInput4)
         ↑                                ↑           ↑
      RESULTS                         FUNCTION     INPUTS
```

INPUTS `iDigitalInputX`	RESULT `iInputStateX`
1 = External Start Switch (Pin 1)	Integer containing one of the following codes: **0** = on or conducting **OR** **1** = off or not conducting
2 = Hipot Safety Switch (Pin 2)	
3 = User-Defined (Pin 3)	
4 = User-Defined (Pin 4)	



Examples:

- `externalStartSwitchState = ReadUserInputStates(1)`

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

ReadUserInputStates, *continued*

Example:

-
  ```
  function TestOutputs()
      local io1, io2, io3, io4, sStr
      local iCount = 1
      while iCount ~= 0 do
          SetUserOutputStates(1,1,2,1,3,1,4,1,5,1,6,1)
          SetUserOutputStates(iCount,0)
          iCount = iCount + 1
          if iCount > 6 then
              iCount = 1
          end
          io1,io2,io3,io4 = ReadUserInputStates(1,2,3,4)
          sStr = format("input 1 = %i, 2 = %i, 3 = %i, 4 = %i"
                           ,io1,io2,io3,io4)
          outtextxy(1,7,"                                              ")
          outtextxy(5,8,sStr)
          outtextxy(1,9,"                                              ")
          Delay(0.5)
          local ioSum = io1+io2+io3+io4
          if ioSum == 0 or ioSum == 4 then
              iCount = 0
          end
      end
  end
  ```

  The function will set an output and read the input states and then display the input states on the touch screen. It will continue doing this for all the output states.

RemountDrive

Explanation:

Use this function to remount a drive on a Touch 1 running Linux. The function returns a description of itself if it is called without any input; otherwise it does not return any results.

Note: 1100 testers do not support this function

Format:

```
RemountDrive(sDriveLetter)
     ↑             ↑
 FUNCTION        INPUT
```

INPUT `sDriveLetter`
String containing the drive letter to remount. The drive letter is not case sensitive.

Example:

- `RemountDrive(“a”)`

  This example will remount the floppy drive.

remove

Explanation:

Use this function to delete a file. If the function fails to delete the file, it returns a nil plus a string describing the error.

Format:

```
error, sErrMsg = remove(sFilename)
     ↑              ↑        ↑
  RESULTS        FUNCTION  INPUT
```

INPUT `sFilename`	RESULT1 `error`	RESULT2 `sErrMsg`
String containing the path and name of the file to be deleted.	Contains a nil if the file cannot be deleted **or** a value other than nil if the file is deleted.	A string describing the error if the function fails **or** a nil if the file is deleted.

Example:

-
  ```
  sFilename = "/DOS/touch1/temp.fil"
  error, sErrMsg = remove(sFilename)
  ```

  This example will delete the file, "`temp.fil`. If the function fails, it will return a nil in `error` and an error string such as `"No such file or directory"` in `sErrMsg`.

rename

Explanation:

Use this function to rename a file. If the function fails to rename the file, it returns a nil plus a string describing the error.

Format:

```
error, sErrMsg = rename(sOldFilename, sNewFilename2)
     ↑              ↑             ↑
  RESULTS        FUNCTION       INPUTS
```

INPUT1 `sOldFilename`	INPUT2 `sNewFilename`	RESULT1 `error`	RESULT2 `sErrMsg`
String containing the path and filename of the file to rename.	String containing the path and the new filename after being renamed.	nil = file cannot be renamed **OR** value other than nil if the file is renamed.	A string describing the error if the function fails **OR** nil = file was renamed

Example:

-
  ```
  sOldFilename = "/DOS/touch1/temp.fil"
  sNewFilename = "/DOS/touch1/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 s`ErrMsg`.

SaveSPCData

Explanation:

Use this function to save user-defined data or other Touch 1 data not automatically saved by SPC Data Collection. It must be called for each user-defined field you want to save to SPC data. The user-defined data can be obtained from the result of the PromptForUserInformation function. The function will return a description of itself if there are no input parameters.

SPC data collection must be turned ON for the current wirelist. Use the function `IsSPCDataCollectionOn` to check for the current state.

Format:

```
SaveSPCData(iWhenDataIsCollected, sDataNameText, sDataValueText)
    ↑                             ↑
FUNCTION                       INPUTS
```

INPUTS	INPUTS DESCRIPTION	RESULT
`iWhenDataIsCollected`	Integer containing how to collect the SPC data: **1** = Per cable **2** = Per test run	Saves the inputted data name and value to SPC Data Collection.
`sDataNameText`	String containing the text for SPC data collection user-defined field name.	
`sDataValueText`	String containing the data to collect. This value can be the response from the `PromptForUserInformation` function	

Examples:

-
  ```
  iWhenDataIsCollected = 2
  sDataNameText = "Company Name"
  sDataValueText = "XYZ COMPANY"
  SaveSPCData(iWhenDataIsCollected, sDataNameText, sDataValueText)
  ```
  The function will save the company name, "XYZ COMPANY", to SPC Data Collection on every test run.

-
  ```
  sRelayTestResult = "PASSED" or "FAILED"
  SaveSPCData(1, "Relay Test" , sRelayTestResult)
  ```
  The function will save the data name text, "Relay Test", and its value, "PASSED" or "FAILED" which depends on the outcome of the relay test, to SPC Data Collection for every cable.

-
  ```
  functionDescription = SaveSPCData()
  ```
  The function returns the function description as text in the RESULT, `functionDescription`

SendTextToParallelPrinter

Explanation:

Use this function to send text out to the printer through the parallel port. If the function has no input, it will return a description of itself.

This function will override the Touch 1's automatic print after test if it is turned on.

Format:

```
iNumCharsPrinted, iPrinterStatus = SendTextToParallelPrinter(
                                             sTextToPrint
                                             [,iAddFormatting])
       ↑                 ↑                  ↑                  ↑
    RESULTS          FUNCTION            INPUT1             INPUT2
```

INPUT1 `sTextToPrint`	INPUT2 `iAddFormatting` **`(optional)`**	RESULT1 `iNumCharsPrinted`	RESULT2 `iPrinterStatus`
String containing text to be sent to parallel printer.	Nonzero = Add carriage returns Defaults to zero.	Integer containing the number of characters printed.	**0** = Ready
			1 = Printer Not Selected
			2 = Timeout
			3 = I/O Error
			4 = Out of Paper
			5 = Printer Busy

Formatting Codes:

- `\13\10 = carriage return and linefeed`
- `\12    = formfeed`
- `\r     = back up to beginning of current line. Can be used for underlining. See an example under the Format function, page 75.`

Examples:

-
  ```
  sTextToPrint = "Text string to be printed"
  iNumCharsPrinted, iStatus = SendTextToParallelPrinter(sTextToPrint)
  ```
  The function will print "Text string to printed" on the parallel 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")`

  The function will print "GOOD CABLE" on the parallel printer. No return values are being used.

- `iPrinterStatus = SendTextToParallelPrinter("^XA^FDhello^FS^XZ")`

  The function will print "hello" on a zebra printer. The control characters are defined as follows: ^XA = start format, ^XZ = end format, ^FD = start of field data, ^FS = end of field data. A Zebra printer needs all the text as one big string. Sending a report one line at a time will cause the Zebra printer to timeout. See your manual for your specific zebra printer.

- `sFunctionDescription = SendTextToParallelPrinter()`

  The function will return the function description as text in the variable, s`FunctionDescription`.

SendTextToParallelPrinter, *continued*

-
  ```
  local theReport =
  {
     { "                                    GOOD CABLE REPORT" },
     { "\r" },
     { "                                    _________________\n" },
     { "\n\n\n" },
     { "Company:  %.30s\n", company },
     { "%60s", "Date:  " },
     { "%s\n", GetDateAsText(5) },
     { "%68s", format("Operator:  %.30s\n", operator) },
     { "%66s", format("Shift:  %.4s\n", shiftCode) },
     { "\n\n" },
     {"                  GOOD CABLE"},
     { "\n\n" },
     { "Cable Signature:  %.12s\n", cableSignature },
     { "Cable Description:  %.30s\n", cableDescription },
     { "Cable Serial #:  %.30s\n", myLongTermCableSerialNumber },
     { "\x0C" },     -- formfeed to skip to next report
     { "" }
  }

  iIndex = 1  -- start at item 1
  while theReport[iIndex] ~= nil do
    local sText = call( format, theReport[iIndex]) – format text
    SendTextToParallelPrinter(sText)
    iIndex = iIndex + 1  -- go to next item in theReport
  end
  ```

```
                              GOOD CABLE REPORT


Company:   CIRRIS
                                                            Date:  7/27/1998
                                                            Operator:  BOB
                                                            Shift:  SHF1
                                 GOOD CABLE

Cable Signature:  984123-2XT9H
Cable Description:  Last Learned
Cable Serial Number:  45453
```

This example will print a good cable report on a dot matrix printer attached to the parallel port. It calls the built in format function to format the text and put it into the variable, `sText`. The formatted text is then sent to the printer.

SetCableSerialNumber

Explanation:

Use this function to save the cable serial number to SPC data collection. SPC data collection must be turned ON for the current wirelist. Use the function `IsSPCDataCollectionOn` to check for the current state. If the function is called with no input parameters, it will return a description of itself.

Format:

```
SetCableSerialNumber(sInputText)
          ↑                 ↑
       FUNCTION           INPUT
```

INPUT `sInputText`	RESULT
A text string containing the cable serial number to save to SPC data.	Saves cable serial number to SPC Data Collection

Examples:

-
  ```
  sInputText = "12345678"
  SetCableSerialNumber(sInputText)
  ```

  The function will save the cable serial number, 12345678, to SPC data collection.

-
  ```
  if IsSPCDataCollectionOn() == 1 then
      SetCableSerialNumber("6443AX92")
  else
       MessageBox(“SPC Data Collection is OFF!”)
  end
  ```

  The function will save the cable serial number, 6443AX92, to SPC data collection if it is turned on. Otherwise, a message box will display with text it is off.

-
  ```
  sFunctionDescription = SetCableSerialNumber()
  ```

  The function will return the function description as text in the variable, `sFunctionDescription`.

SetDelayTimeInMilliseconds

Explanation:

Use this function to set a delay time in milliseconds. If no input is given, the function will return a description of itself. This function can be used for serial printers that need a delay time. Also, see the function, `Delay(),` for setting a delay time in seconds.

Format:

```
SetDelayTimeInMilliseconds(iInputNum)
            ↑                  ↑
         FUNCTION            INPUT
```

INPUT `iInputNum`	RESULT
An integer corresponding to the number of milliseconds of delay time.	Delay for the number of milliseconds equal to `iInputNum`.

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.

- `functionDescription = SetDelayTimeInMilliseconds()`

  The function will return the function description as text in the RESULT, `functionDescription`.

SetOperatorName()

Explanation:

This function applies to the Signature 100 series tester. Use this function to apply a string to the tester's memory that can be used in SPC and other functions that use the opertor's name.

Format:

SetOperatorName(sName)

INPUT `sName`
String containing text known as operator name.

Example:

```
sName = PromptForUserInformation(1,"Enter your ","Name",20)
SetOperatorName(sName)
```

In this example, we use the PromptForUserInformation function to obtain the text and assign it to the variable sName, then we assign the text in sName to the SetOperatorName function.

SetSerialParams

Explanation:

Use this function to initialize the serial printer so you can write or read data from the serial port. Use this function before using the function `WriteToSerial`.

Available Baud Rates are: 9600, 19200, 57600, and 115200

Format:

```
SetSerialParams([sPortNumber,] sSerialParamString)
      ↑                      ↑
   FUNCTION              INPUTS  ( [ ] = optional )
```

INPUT	DESCRIPTION	RESULT2 `iErrorCode`
`sPortNumber` **(optional)**	String containing which COM port: **com1** or **com2** Function defaults to port COM1 if not specified.	**-999** = timeout **-1000** = no port **-1001** = port in use **-1002** = bad word size **-1003** = bad parity **-1004** = invalid baud **-1005** = bad stop bits
`sSerialParamString`	A text string corresponding to the baud rate, data bits, parity, and stop bits for your serial printer. "<baud rate>:<data bits><parity><stop bits>" data bits = **7** or **8** parity = even (**e**), odd (**o**), or none (**n**) stop bits = **1** or **2**	

Examples:

-
  ```
  sPortNumber = “com2”
  sSerialParamString = "9600:8n1"
  SetSerialParams(sPortNumber, sSerialParamString)
  ```
  The function will set up the serial printer to use the following: port = COM2 , baud rate = 9600, data bits = 8, parity = none, stop bits = 1.

- `SetSerialParams("19200:8n1")`

  The function will set up the serial printer to use the following: baud rate = 19200, data bits = 8, parity = none, stop bits = 1. The port defaults to COM1 or can be set in the `WriteToSerial` function.

SetUserOutputStates

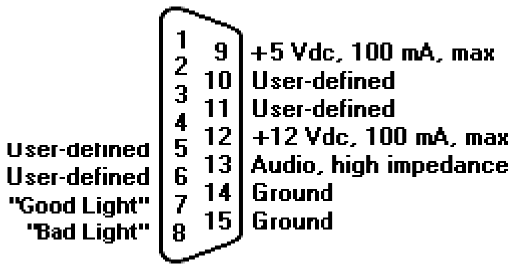
Explanation:

Use this function to set the digital outputs on the user I//O port to a particular logic state. `SetUserOutputStates` takes up to six parameter pairs. In each pair, the first value is the output number and the second value is the new logic state of the output (low or high). For example, a user-connected light could turn ON when the output is logic LOW (0). You can use any combination of the six output pairs and the function will set their respective logic states: low (conducting) or high (not conducting). See the function `GetUserOutputStates` to read the state of the outputs.

Note:

- On the Touch1, outputs 5 and 6 might get overridden by anything that turns on/off the good/bad LED.

Pinout:



Format:

```
SetUserOutputStates(iDigitalOutput1, iOutputState1,
                    iDigitalOutput2, iOutputState2,
                    iDigitalOutput3, iOutputState3,
                    iDigitalOutput4, iOutputState4,
                    iDigitalOutput5, iOutputState5,
                    iDigitalOutput6, iOutputState6,
                    iDigitalOutput7, iOutputState7,
                    iDigitalOutput8, iOutputState8)
      ↑                        ↑
   FUNCTION                  INPUTS
```

INPUT1 `iDigitalOutputX`	INPUT2 `iOutputStateX`
1 = Pin 5	Integer containing one of the following codes: **0** = On or conducting **1** = Off or not conducting
2 = Pin 6	
3 = Pin 10	
4 = Pin 11	
5 = Good Light & Pin 7 (Touch1) **5** = Good Light Only (1100)	
6 = Bad Light & Pin 8 (Touch1) **6** = Bad Light Only (1100)	
7 = Pin 7 (1100 only)	
8 = Pin 8 (1100 only)	

SetUserOutputStates, *continued*

Examples:

-
  ```
  iDigitalOutput1 = 1
  iOutputState1 = 0
  iDigitalOutput2 = 2
  iOutputState2 = 1
  iDigitalOutput3 = 3
  iOutputState3 = 0
  iDigitalOutput4 = 4
  iOutputState4 = 0
  iDigitalOutput5 = 5
  iOutputState5 = 0
  iDigitalOutput6 = 6
  iOutputState6 = 0
  SetUserOutputStates(iDigitalOutput1, iOutputState1,
                      iDigitalOutput2, iOutputState2,
                      iDigitalOutput3, iOutputState3,
                      iDigitalOutput4, iOutputState4 ,
                      iDigitalOutput5, iOutputState5,
                      iDigitalOutput6, iOutputState6)
  ```

  The function will set Pin 6 off and all others on (Pins 5, 10, 11, Good Light, Bad Light).

- `SetUserOutputStates(6,0)`

  The function will turn on the Bad Light led.

sin

Explanation:

Use this function to return the sine of a number. Angles are in degrees and not radians.

Format:

```
myValue = sin(myInputNum)
  ↑         ↑       ↑
RESULT   FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `myValue`
A numeric value to compute the sine on.	A numeric value containing the sine value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = 5.3
  myValue = sin(myInputNum)
  ```

  The function will return `myValue` equal to .09237.

- `sinResult = sin(3.4)`

  The function will return `sinResult` equal to .0534.

sqrt

Explanation:

Use this function to return the square root of a number.

Format:

```
myValue = sqrt(myInputNum)
   ↑          ↑       ↑
RESULT    FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `MyValue`
Call `sqrt` to find the square root of `myInputNum`.	Contains the square root value for the number, `myInputNum`.

Examples:

-
  ```
  myInputNum = 9
  myValue = sqrt(myInputNum)
  ```
  The function will return myValue equal to 3.

-
  ```
  iResult = sqrt(4)
  ```
  The function will return `iResult` equal to 2.

strfind

Explanation:

Use this function to find the first match to a pattern in a string. This function is case sensitive. If a match is found, it returns the starting index and the ending index of the string for the pattern. If no match is found, the function returns a nil.

Note: When indexing a string, the first character is at position 1, not zero, as in C.

Format:

```
iStartResult, iEndIndex = strfind(sString, sPattern
                                           [, iStartIndex])
          ↑                    ↑                    ↑
       RESULTS              FUNCTION              INPUTS
```

INPUT1 `sString`	INPUT2 `sPattern`	INPUT3 `iStartIndex` **(optional)**	RESULT1 `iStartResult`	RESULT2 `iEndIndex`
String to search in for the pattern to find.	String containing a sequence of characters to find in the input string, `sString`. A ^ at the beginning of a pattern anchors the match at the beginning of the string. A $ at the end of a pattern anchors the match at the end of the subject string. A pattern may contain sub patterns enclosed in parentheses. Each sub pattern is a capture and is returned as one of the values from the search.	Integer indicating where to start the search. If `init` is not supplied, the search will begin at the first character.	Integer indicating where the first character of the search pattern was found.	Integer indicating where the last character of the search pattern was found.

Pattern Details:

Character Class Representation	DESCRIPTION
x	x is any character not in the list: ^$()%.[]*+-?
.	Represents all characters
%a	Represents all letters
%c	Represents all control characters
%d	Represents all digits
%l	Represents all lower case letters
%p	Represents all punctuation characters
%s	Represents all space characters
%u	Represents upper case letters
%w	Represents all alphanumeric characters
%x	Represents all hexadecimal digits
%z	Represents the character with representation 0
%x	Represents any non-alphanumeric character. This is the way to escape the following characters: ^$()%.[]*+-?
For all classes represented by a single letter (%a, %c, etc.), the corresponding uppercase letter represents its complement. For example, %S represents all non-space characters.	

strfind, *continued*

Pattern Item	DESCRIPTION
`*`	Matches 0 or more repetitions of characters in the class. It matches the **longest** possible sequence.
`+`	Matches 1 or more repetitions of characters in the class. It matches the **longest** possible sequence.
`-`	Matches 0 or more repetitions of characters in the class. It matches the **shortest** possible sequence.
`?`	Matches 0 or 1 occurrence of a character in the class.
`%n`	Matches a substring equal to the **n**th captured string where **n** is between 1 and 9.
`%bxy`	Matches strings that start with x and end with y where x and y are balanced. For example, %b() matches expressions with balanced parentheses.

Examples:

-
  ```
  sString = "ABCDDDEFG"
  local iStartIndex, iEndIndex = strfind(sString, "DDD")
  ```

  This example returns i`StartIndex` = 4 and i`EndIndex` = 6 because the first 'D' is the fourth character and the last 'D' is the sixth character in the pattern, "DDD."

-
  ```
  sString = "GOOD CABLE #9"
  local iStartIndex, iEndIndex = strfind(sString, "#10", 12)
  ```

  This example returns i`StartIndex` and i`EndIndex` = nil because the pattern `"#10"` was not found in the input string, `"GOOD CABLE #9"`.

-
  ```
  local iPos, iEnd
  iPos, iValue = strfind(sString, "^%s*([^%s].*)$") -- trim off spaces on left
  iEnd, iValue = strfind(sString, "^(.*[^%s])%s*$") -- trim off spaces on right
  MessageBox("Pos :" .. tostring(iPos)..":\nValue :"..tostring(iValue)..":")
  ```

  This example takes a string, `sString`, and strips off leading and trailing spaces. It then displays the results in a message box. It uses the following codes:

  '^': the next thing to look for is at the beginning of the string
  '$': the last thing to look for is at the end of the string
  '%s': any kind of whitespace, including tab, space, newline, etc.
  '%s*': any number of chars that are whitespace. Works if there is no whitespace at all.
  '.*': any character at all and any number of those characters. Works if there are no chars.
  '^%s*': all leading spaces
  '([^%s.*])$': return all characters that start with a non-whitespace character

strlen

Explanation:

Use this function to get the length of a string returned as an integer.

Format:

```
iStringLength = strlen(sString)
     ↑            ↑      ↑
 RESULT       FUNCTION  INPUT
```

INPUT `sString`	RESULT `iStringLength`
String you want to find the length on.	Integer containing the length of the string, `sString`.

Example:

- `iStringLength = strlen("This is a very long cable")`

  This example returns `iStringLength` = 25 because `"This is a very long cable"` is 25 characters long including spaces.

strlower

Explanation:

Use this function to change a string to all lower case letters. The function will convert all upper case characters and change them to lower case. All other characters remain the same.

Format:

```
sLowerCaseString = strlower(sString)
     ↑                  ↑       ↑
 RESULT             FUNCTION  INPUT
```

INPUT `sString`	RESULT `sLowerCaseString`
Text containing upper case characters you want to convert to lower case.	Text containing the new string in all lower case letters.

Example:

- `sLowerCaseString = strlower("1 = REMOVE THE CABLE")`

  This example returns sLowerCaseString containing: "1 = remove the cable ".

strrep

Explanation:

Use this function to create a new string that is a concatenation of a user-selected number of copies of the original input string. This function is useful when formatting text.

Format:

```
sNewString = strrep(sString, iNumCopies)
     ↑            ↑           ↑
  RESULT      FUNCTION      INPUTS
```

INPUT1 `sString`	INPUT2 `iNumCopies`	RESULT `sNewString`
String to copy and put into the new string, `sNewString`.	Integer containing the number of copies of the input string.	String containing the new concatenated string.

Examples:

- `sNewString = strrep("PASSED ", 3)`

  This example returns `sNewString` = "PASSED PASSED PASSED ".

- `sNewString = strrep(" ", iNumSpacesWanted)`

  This example puts spaces in the string to format text on the page. It returns the number of spaces defined by `iNumSpacesWanted` in `sNewString`.

strsub

Explanation:

Use this function to return a new string that is a subset of the input string.

Format:

```
sNewString = strsub(sString, iStartIndex [, iEndIndex])
     ↑             ↑                ↑
  RESULT       FUNCTION          INPUTS
```

INPUT1 `sString`	INPUT2 `iStartIndex`	INPUT3 iEndIndex **(optional)**	RESULT `sNewString`
String containing text for the new string.	Integer containing the starting index for the new string.	Integer containing the ending index for the new string. If not used, the new string will be the string length from the starting index.	String containing the new string.

Example:

- `sNewString = strsub("This is XYZ COMPANY'S cable", 9, 19)`

  This example returns `sNewString = "This is XYZ COMPANY"` because 'X' is the ninth character and 'Y' is the nineteenth character in the string, `"This is XYZ COMPANY'S cable"`.

strupper

Explanation:

Use this function to change a string to all upper case letters. The function takes all lower case characters and converts them to upper case characters. All other characters remain the same.

Format:

```
sUpperCaseString = strupper(sString)
     ↑                  ↑        ↑
  RESULT            FUNCTION   INPUT
```

INPUT `sString`	RESULT `sUpperCaseString`
String to convert to upper case letters.	String containing the new string in all upper case letters.

Example:

- `sUpperCaseString = strupper("15 = press the green button!")`

  This example returns `sUpperCaseString` = "15 = PRESS THE GREEN BUTTON!".

tan

Explanation:

Use this function to return the tangent of a number. Angles are in degrees and not radians.

Format:

```
myValue = tan(myInputNum)
   ↑         ↑      ↑
 RESULT  FUNCTION  INPUT
```

INPUT `myInputNum`	RESULT `myValue`
Call `tan` to find the tangent of `myInputNum`.	Contains the tangent value for the number, `myInputNum`.

Examples:

- `myInputNum = 3.563`
  `myValue = tan(myInputNum)`

  The function will return `myValue` equal to .06227.

- `myResult = tan(5.43)`

  The function will return `myResult` equal to .0855.

TestPreference

Explanation:

Use this function to get and set test preference settings. If the input is only the preference id, the function will return the current setting. If the inputs are a preference id and a parameter setting, the function will change the setting and return the new setting.

Format:

```
iIDSetting = TestPreference(iTestPrefID, [iPrefParameter])
   ↑                ↑                      ↑
 RESULT          FUNCTION                INPUTS
```

INPUT1 `iTestPrefID`	INPUT2 **(optional)** `iPrefParameter`	RESULT `iIDSetting`
Integer containing the test preference ID. Where: **1** = Single Test **2** = Auto Hipot **3** = Hipot Delay **4** = High Speed Hipot **5** = Auto Fault Location	Integer containing the parameter for the test preference ID input. If INPUT1 is Single Test(1): **0** = Continuous or **1**= Single If INPUT1 is Auto Hipot(2): **0** = Manual or **1**= Automatic If INPUT1 is Hipot Delay(3): **iIDSetting** = Delay in seconds If INPUT1 is High Speed Hipot(4): **0** = OFF or **1**= ON If INPUT1 is Auto Fault Location(5): **0** = OFF or **1**= ON	Integer containing the setting for the test preference ID input. If INPUT1 was Single Test(1): **0** = Continuous or **1**= Single If INPUT1 was Auto Hipot(2): **0** = Manual or **1**= Automatic If INPUT1 was Hipot Delay(3): **iIDSetting** = Delay in seconds If INPUT1 was High Speed Hipot(4): **0** = OFF or **1**= ON If INPUT1 was Auto Fault Location(5): **0** = OFF or **1**= ON

Examples:

- `local iIDSetting = TestPreference(2)`

  The function will return the current setting for the Automatic Hipot test preference setting. If automatic hipot is on, the function will return `iIDSetting = 1 or` it will return `iIDSetting = 0` if manual hipot is on.

- `local iIDSetting = TestPreference(5, 1)`

  The function will set the Automatic Fault Location test preference setting to on. It will return `iIDSetting = 1` indicating automatic fault location is turned on.

TestWirelist

Explanation:

Use this function to perform tests on the current wirelist. The high voltage test will perform the test on all the nets. For testing a single net, use the function `HipotNetTiedToPoint`. Use the function `UseChildWirelist` to load the child wirelist before calling this function to test. Use the function `TWLGetErrorText` to retrieve the error text if the test fails. This function does not affect SPC Data Collection or the cable test counters.

Format:

```
iTestVal = TestWirelist(iTestNum)
    ↑            ↑              ↑
 RESULT      FUNCTION        INPUT
```

INPUT `iTestNum`	RESULT `iTestVal`
Integer containing one of the following codes: **3** = LV test only **4** = HV test only (All Nets) **255** = LV, Components, & HV tests	Integer containing one of the following codes: **Nonzero** = Test failed **3** = LV test failed (includes components test) **4** = HV test failed

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 the `UseChildWirelist` example.

TimePassed

Explanation:

Use this function to get the time passed since the timer was opened or reset. The timer should be created using the function TimerOpen.

Format:

```
iTimePassed = TimePassed(iTimerHandle)
     ↑              ↑          ↑
   RESULT       FUNCTION     INPUT
```

INPUT `iTimerHandle`	RESULT `iTimePassed`
Integer containing the handle for the created timer.	Integer containing the time passed (msec) since the timer was opened or reset.

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 be displayed. The timer will then be destroyed.

TimerClose

Explanation:

Use this function to destroy the timer object and free one of the ten available timers. The timers are created using the function TimerOpen.

Format:

```
TimerClose(iTimerHandle)
     ↑           ↑
  FUNCTION     INPUT
```

INPUT `iTimerHandle`
Integer containing the handle for the created timer.

Example:

See the example under the *TimerOpen* function.

TimerDone

Explanation:

Use this function to check if the timer has timed out or there is more time remaining. The timer should be created using the function TimerOpen.

Format:

```
iTimerDone = TimerDone(iTimerHandle)
     ↑            ↑          ↑
   RESULT      FUNCTION    INPUT
```

INPUT `iTimerHandle`	RESULT `iTimerDone`
Integer containing the handle for the created timer.	Integer containing the status of the timer. **0** = Timer is done **1** = More time remaining

Example:

See the example under the *TimerOpen* function.

TimerOpen

Explanation:

Use this function to create a timer with a timeout delay. This timer cannot be used on 1100 hardware while hipotting. The maximum number of available timers at any one time is ten. If too many timers are opened, use the function TimerClose to destroy an existing timer before creating a new one.

Format:

```
iTimerHandle, iTimerStatus = TimerOpen(iDelayMSec)
          ↑                           ↑          ↑
       RESULTS                    FUNCTION     INPUT
```

INPUT `iDelayMSec`	RESULT1 `iTimerHandle`	RESULT2 `iTimerStatus`
Integer containing the timeout delay for the created timer.	Integer containing the handle for the created timer. Use this value for all the other timer commands: TimePassed, TimerDone, TimerReset, and TimerClose.	Integer containing the status of the created timer where: **0** = Timer okay **1** = Too many timers opened (Max open timers = 10) **2** = Bad delay parameter (not a number or is nil) **3** = Invalid timeout: delay is < 0 **or** delay > 4 seconds (1100 hardware only) **4** = Hipot timer conflict (This timer cannot be used on 1100 hardware while hipoting.)

Example:

```
local iTimerHandle = TimerOpen(100)
while TimerDone(iTimerHandle) == 0 do
      sProbedPoint = GetProbedPin()
      if sProbedPoint ~= nil then
            TimerReset(iTimerHandle)
      end
end
TimerClose(iTimerHandle)
```

This example will create a timer with a 100 millisecond timeout. If the timer has not timed out, all the points touched by the probe will be returned in `sProbedPoint`. The timer will be reset if no points are probed. At the end, the timer is destroyed.

TimerReset

Explanation:

Use this function to restart the timer using the original timeout delay passed in when the timer was created. The timer should be created using the function TimerOpen.

Format:

```
TimerReset(iTimerHandle)
     ↑           ↑
  FUNCTION     INPUT
```

INPUT `iTimerHandle`
Integer containing the handle for the created timer.

Example:

```
See the example under the TimerOpen function.
```

tmpname

Explanation:

Use this function to get a filename that can safely be used for a temporary file. The filename is returned as a text string. This function does not open the file, and the file must be deleted when it is no longer needed.

Note: 1100 testers do not support this function

Format:

```
sTmpFileName = tmpname()
    ↑             ↑
 RESULT        FUNCTION
```

RESULT
`sTmpFileName =` String containing the filename of the temporary file.

Example:

-
  ```
  sTmpFileName = tmpname()
  writeto(sTmpFileName)
  write("GOOD")
  writeto()
  remove(sTmpFileName)
  ```

  This example returns a temporary filename as a string in `sTmpFileName.` The file is opened for writing by the function, `writeto`. The text "GOOD" is written to the file and then the file is closed using the functions, `write` and `writeto`. Finally, the temporary file is deleted.

tonumber

Explanation:

Use this function to convert an argument of any kind to a number. If the function cannot do the conversion, it returns a nil. Use this function to convert input for the numeric PromptForUserInformation box or to convert values returned from measurements. See the function `type` to get the kind of value.

Format:

```
myNumber = tonumber(myVariable)
   ↑           ↑          ↑
RESULT      FUNCTION    INPUT
```

INPUT `myVariable`	RESULT `myNumber`
The variable which will be converted to a number.	The number that was converted from the input, `myVariable` **OR** nil = conversion failed

Example:

-
  ```
  sString = "10"
  myNumber = tonumber(sString)
  ```
  This example returns a 10 in `myNumber`

tostring

Explanation:

Use this function to convert an argument of any kind to a string type. This function is useful for debugging along with the function `outtextxy` where the conversion can be displayed on the screen. See the function `type` to get the kind of value.

Format:

```
sResult = tostring(myValue)
   ↑           ↑        ↑
RESULT     FUNCTION   INPUT
```

INPUT `myValue`	RESULT `sResult`
The value which will be converted to a string.	The string that was converted from the input number, `myValue`.

Example:

-
  ```
  iInteger = 4
  sResult = tostring(iInteger)
  ```
  This example returns a "4" in `sResult`.

TWLGetErrorText

Explanation:

Use this function to get error text from the last test after the function `TestWirelist` is called. It is the ONLY reliable way to get error text when testing wirelists using the function `TestWirelist`. Also, see the function `UseChildWirelist` to load a wirelist for testing.

Format:

```
sResult = TWLGetErrorText(iTestVal)
   ↑            ↑                 ↑
RESULT       FUNCTION          INPUT
```

INPUT `iTestVal`	RESULT `sResult`
Integer containing one of the following codes: **3** = LV & component errors only **4** = HV errors only **255** = LV, component, & HV errors.	String containing the error text.

Example:

-
  ```
  iTestVal = TestWirelist(255)
  if iTestVal > 0 then
        sErrText = TWLGetErrorText(iTestVal)
  end
  ```

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

- See the `UseChildWirelist` example.

type

Explanation:

Use this function to get the input's kind of value. It returns one of five specific strings for the value type. The functions `tonumber` and `tostring` can be used to change the type.

Format:

```
sType = type(myValue)
  ↑         ↑     ↑
RESULT   FUNCTION INPUT
```

INPUT `MyValue`	RESULT `sType`
The type is wanted for this variable.	String containing one of the following codes: **"nil"** = false (0) **"number"** = real floating point number **"string"** = text in quotes **"table"** = an array that can be indexed by any value not just numbers. **"function"** = a function that can be called.

Examples:

-
  ```
  myValue = 100
  sType = type (myValue)
  ```

  This example returns the string, `"number"` in `sType` because 100 is a number.

-
  ```
  theType = type("XYZ COMPANY")
  ```

  This example returns the string, `"string"` in `theType` because `XYZ COMPANY` is text.

UseChildWirelist

Explanation:

Use this function to load a child wirelist on top of the current wirelist in lastlrnd. The function will load the wirelist from memory or disk. The function `TestWirelist` can be called to test the wirelist. Call `UseChildWirelist` again and it will then stop using the wirelist but leave it in memory or stop using the wirelist and free its memory. Use this function as a fast way to replace child wirelists by loading them and leaving them in memory. Test all the wirelists and after all testing has finished, remove all of the wirelists from memory by calling this function again.

Note:

There is only one instance of an evtEvent script so a parent wirelist cannot call a child wirelist that has an event script in it. The child's events will not be called because the parent's events are still in memory.

Format:

```
iResult = UseChildWirelist(sChildWLFilename)
                  THEN
iResult = UseChildWirelist(iStopNumber)
   ↑                ↑                 ↑
 RESULT          FUNCTION           INPUT
```

INPUT `sChildWLFilename`	RESULT `iResult`
A string containing an optional path and the name of the child wirelist which will be loaded.	Integer containing one of the following: **0** = Loaded OK **nonzero** = Failed loading child wirelist

THEN

INPUT `iStopNumber`
Integer containing one of the following: **0** = Stop using the current child wirelist but leave it in memory. **1** = Stop using the current child wirelist and free the memory.

Examples:

- `iCouldntLoad = UseChildWirelist(sChildWLFilename)`

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

Continued on the next page

UseChildWirelist, *continued*

-
  ```
  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.

WiresAttached

Explanation:

Use this function to determine if a cable is present. It checks only the adapters listed in the currently loaded wirelist. There are no inputs to this function.

Format:

```
iWiresAttached = WiresAttached()
      ↑                    ↑
   RESULT               FUNCTION
```

RESULT `iWiresAreAttached`
Integer containing one of the following codes: **nil** = nothing is attached **1** = wires are attached

Example:

- `iWiresAreAttached = WiresAttached()`

  `iWiresAreAttached` will contain a nil if no wires are attached or a one if wires are attached.

write

Explanation:

Use this function to write to a file that has been opened or created using the function `writeto`. Call the function `writeto` to close the file when you are finished writing.

Format:

```
error, sErrMsg = write(iFileHandle, value1, value2, ...)
       ↑              ↑           ↑
   RESULTS        FUNCTION     INPUTS
```

INPUT1 **(optional)** `iFileHandle`	INPUTS `value1, value2, value3, ...`	RESULT1 `error`	RESULT2 `sErrMsg`
A number containing the file handle obtained from the writeto() function.	String or integer value that will be written to the file. To write other value types, use the `tostring` or `format` functions to convert the value to a string or number.	nil = error writing to file **OR** A value other than nil if the file is written to correctly.	A string describing the error if the function fails such as “no file or directory”. **OR** nil = file is written to correctly

Examples:

-
  ```
  iFileHandle = writeto("/DOS/touch1/temp.fil")-- Open temp.fil
  error, sErrMsg = write(iFileHandle, "Company XYZ")
  write(" Plant #5")
  write(12345678, "\n5:15")
  writeto()         -- Close the file
  ```

  This example writes the following to the temp.fil file.

  Company XYZ Plant #5

  ```
  12345678
  5:15
  ```

  If the function fails writing to the file, it will return a nil in `error` and an error string in `sErrMsg`.

-
  ```
  local sTempstr = tostring(12345678) -- Convert number to string
  write(sTempstr)   -- Write string to file
  ```

  This example converts the number, 12345678 to a string and then writes the string, "12345678" to the file.

WriteBlockToSerial

Explanation:

Use this function to send data to a serial device. For example, use it to send a custom report out to the serial printer. The function `SetSerialParams` must be called before using this function. The port defaults to COM1 if it was not specified in `SetSerialParams` or `WriteBlockToSerial`.

Format:

```
iNumCharsSent, iErrorCode = WriteBlockToSerial([sPortNumber,]
                                                sTextToSend,
                                                iNumCharsToSend)
           ↑                        ↑                   ↑
        RESULTS                  FUNCTION            INPUTS  ( [ ] = optional )
```

INPUT1 **(optional)** `sPortNumber`	INPUT2 `sTextToSend`	INPUT3 `iNumCharsToSend`	RESULT1 `iCharsSent`	RESULT2 `iErrorCode`
A string containing which COM port to use. “**com1”** or **com2**” If the input is not used, the default is com1.	A string containing the data to send.	An integer containing the block size of the text to send.	An integer containing the number of characters sent.	Integer containing one of the following codes: **-999** = timeout **-998** = no port **-997** = port in use **-996** = bad word size **-995** = bad parity **-994** = invalid baud **-993** = bad stop bit **OR** **nil** = good

Example:

```
sTextToSend = "Text string to be sent."
iNumCharsSent, iErrorCode = WriteBlockToSerial(sTextToSend, 23)
```

The function will send "Text string to be sent." out the default port COM1. `iNumCharsSent` will contain the number of characters sent if there are no errors. Otherwise, `iErrorCode` will contain an error code.

writeto

Explanation:

Use this function to open a new file so something can be written in it. Include the path and the extension for the filename. If the file already exists, it will be completely erased when calling this function.

Format:

```
canWriteFile, sErrMsg = writeto([sFilename])
              ↑                 ↑         ↑
           RESULTS          FUNCTION    INPUT
```

INPUT `sFilename`	RESULT1 `canWriteFile`	RESULT2 `sErrMsg`
A string containing the name of the file to be created and opened for writing. Include the filename extension and path. **OR** When the function is called without a filename, it closes the file.	Contains the file handle for the filename **OR** nil = open failed	A string describing the error if the function fails.

Example:

-
  ```
  sFile = "/DOS/touch1/storage.dat"
  iCanWriteFile, errMsg = writeto(sFile)
  if iCanWriteFile then
     write("Company XYZ")
     writeto()      -- Close the file
  end
  ```

  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.

WriteToSerial

Explanation:

Use this function to send data to a serial device. For example, you can send text out to the serial printer. The function, `SetSerialParams`, must be called before using this function to initialize the serial port. The port defaults to COM1 if it was not specified in `SetSerialParams` or `WriteToSerial` functions.

Format:

```
iNumCharsSent, iErrorCode = WriteToSerial([sPortNumber,]
                                                 sTextToSend)
           ↑                        ↑             ↑
        RESULTS                 FUNCTION       INPUTS  ( [ ] = optional )
```

INPUT1 **(optional)** `sPortNumber`	INPUT2 `sTextToSend`	RESULT1 `iNumCharsSent`	RESULT2 `iErrorCode`
String containing which COM port to use. “**com1”** or “**com2**” Default = com1	String containing text to be sent to the serial device.	Integer containing the number of characters sent.	Integer containing one of the following codes: **-999** = timeout **-998** = no port **-997** = port in use **-996** = bad word size **-995** = bad parity **-994** = invalid baud **-993** = bad stop bit **OR** **nil** = good

Examples:

-
  ```
  local theReport =
  {
     { "COMPANY NAME        %s\r", GetDateAsText(5) },
     { "Cable Serial #:  %i\r", sLongTermCableSerialNumber },
     { "\r\r\r" }-- three linefeeds to skip to the next label
  }

  SetSerialParams("9600:8n1")
  iIndex = 1              -- start at item 1
  while theReport[iIndex] ~= nil do
     local sText = call( format, theReport[iIndex])
     WriteToSerial(sText)-- Send the text to the serial printer
     iIndex = iIndex + 1-- go to next item in theReport
  end
  ```

  This example will print a cable serial label on a serial printer. It calls the SetSerialParams function to set up the serial port. The built in format function is called to format the text and put it into the string, `sText`. The formatted text, `sText,` is then sent to the serial printer.

```
COMPANY NAME                7/27/98
Cable Serial #:  4545766
```

WriteToSerial, *continued*

-
  ```
  sPortNumber = "com2"
  sTextToSend = "Text string to be sent"
  iNumCharsSent, iStatus = WriteToSerial(iPortNumber, sTextToSend)
  ```

  This example will print "Text string to be sent" on the serial printer from port COM2. `iNumCharsSent` will contain the number of characters sent and `iStatus` will contain a nil if all the characters are sent.

Glossary

Cable Serial Number	Using scripting, you can assign serial numbers to the cables you test. The serial number can then be saved as part of your Statistical Process Control (SPC) data (if you've purchased the SPC package for your Touch 1) and/or displayed on a custom report or a label.
Custom Reports	Using scripting, you can program your Touch 1 to print out reports containing special information you've requested. The default test event script files `badrpt.lua` and `goodrpt.lua` are examples of scripts that instruct the Touch 1 to print custom reports. The default script files: `autobad.rpt`, `autogood.rpt`, `errors.rpt`, `testsum.rpt`, and `wirelist.rpt` are also custom reports.
Feature Access Code	A six character **Feature Access Code** is used to install the scripting feature on the Touch 1. This code must be entered as part of the script installation process.
LUA language	The programming language used to write scripts for use on your Touch 1.
Sample Cable	A **Sample Cable** is a kind of cable you want to test which you know is built correctly. This known good cable is learned on the Touch 1 and then is used as the standard against all cables of that type you test. If a cable under test is built the same as the Sample Cable, the Touch 1 will report a good cable.
Script	A **script** in the Touch 1 is a file containing a set of instructions that let the Touch 1 do things that are not included as a standard part of the software. Several default scripts are included as part of the software in the Scripting package. To run a script file, it must be attached to the wirelist you're testing using that script. Each script is written using commands from the LUA programming language.
Scripting	**Scripting** on the Touch 1 lets you add features, or customize the behavior of your Touch 1. Scripting lets you set up the Touch 1 to perform functions that are not shipped as a standard part of the software. The scripting language in the Touch 1 uses the LUA programming language.
Serial Number	Each Touch 1 tester has a unique **serial number**. The serial number for your analyzer appears in the Main Menu Help screen.
Test Group / Run	A unique number assigned to a cable testing session used for SPC data collection. A **test group** is created each time a test window is opened from the Main Menu screen or the Test Setup screen on the Touch 1. Each test group records the number of tests performed. This may not be the same as the number of cables tested if the same cable is tested more than once. A test group is closed and the test count is ended on returning to the Main Menu screen.
User-defined field(s)	Using scripting on your Touch 1, you can design your own customized fields (referred to as **user-defined fields**) to appear in reports, or as part of your Statistical Process Control (SPC) data. Examples of such fields are your company name, the Touch 1 operator's shift code, or each cable's serial number.

Index