SSI Test Executive v6.36

User Manual

Document No: 315RTE9410-SLV63
February 17, 2007

Serendipity Systems, Inc.
PO Box 774507
Steamboat Springs, CO 80477

TEL: (720) 246-8925

www.serendipsys.com

productinfo@serendipsys.com

PRODUCT INFORMATION:

SSI Test Executive Software (Referred to as “the Software” or “this Software”)

PO Box 774507
Steamboat Springs, CO 80477

1. Limited Liability

IF INSTALLED AND OPERATED AS REQUIRED, THE SOFTWARE SHOULD PERFORM AS DESCRIBED IN THE DOCUMENTATION ENCLOSED HEREWITH. ALL OTHER ASPECTS THE SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY OTHER WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED.

YOU ARE NOT GRANTED ANY IMPLIED WARRANTIES OR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU EXCEPT AS SET FORTH IN PARAGRAPH 5 BELOW. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT SSI OR ANY AUTHORIZED SSI DEALER) ASSUME THE ENTIRE RESPONSIBILITY AND COST OF ALL NECESSARY OR INCIDENTAL RESULTS PRODUCED, AS WELL AS ANY DAMAGES OF ANY KIND AND ANY SERVICE, REPAIR OR CORRECTION THAT MAY BE REQUIRED.

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

Serendipity Systems, Inc. (SSI hereinafter) does not warrant that any of the functions contained in the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free.

SSI warrants that the distribution media on which the Software is furnished to be free from defects in material or workmanship under its intended use, for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

SSI shall not be responsible for any adverse effects caused by Acts of God or any other cause beyond SSI's reasonable control.

2. Limitation of Remedies

SSI's entire liability and your exclusive remedy shall be limited to the replacement within (30) days, for you (the original acquirer), of any distribution media not meeting SSI's Limited Warranty which is returned to SSI or any authorized dealer with a clearly legible copy of your receipt.

IN NO EVENT WILL SSI BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, LOST OPPORTUNITIES, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH SOFTWARE, EVEN IF SSI OR AN AUTHORIZED DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MY NOT APPLY TO YOU.

ABOUT THIS DOCUMENT

This document contains operating instructions and development information for the Serendipity Systems' Test Executive. This information is also contained in an online help file that is included with the product.

The information contained in this document is believed to be correct as of the date of publication. The authors assume no liability for any loss that might occur as a result of errors or omission of any information. Please report any errors to the authors for correction in later revisions of this document.

Table of Contents

1. INTRODUCTION 1

2. TEST EXECUTIVE 3

2.1 TEST LIST 4
2.1.1 Test List Status 5
2.1.2 Viewing Test Information 6
2.1.3 Test Program Help 7
2.2 TEST EXECUTION 8
2.2.1 Control Buttons 9
2.2.2 Execution Status 10
2.2.3 Power Monitor Display 11
2.2.4 Custom Logo Display 11
2.2.5 Test Control Options 12
2.3 STATUS BAR 13
2.4 TOOLBAR 13
2.5 MENUS 14
2.5.1 File Menu 14
2.5.2 Run Menu 16
2.5.3 Options Menu 18
2.5.4 Tools Menu 19
2.5.5 Window Menu 20
2.5.6 Help Menu 21
2.6 DIALOGS AND WINDOWS 22
2.6.1 Test Program List 22
2.6.2 Test Configuration 23
2.6.3 Breakpoint Control 25
2.6.4 Test Results 26
2.6.5 Debug Log 27

3. REPORT GENERATOR 29

3.1 TEST LIST 30
3.2 TEST LIST STATUS 31
3.3 TEST RESULTS 32
3.4 STATUS BAR 33
3.5 TOOLBAR 33
3.6 MENUS 34
3.6.1 File Menu 34
3.6.2 Options Menu 36
3.7 DATASHEETS 38
3.7.1 Printing a Datasheet 38
3.7.2 Printing to a File 39
3.7.3 Detail Datasheet Example 40
3.7.4 Summary Datasheet Example 41

4. PRIVILEGE EDITOR 43

4.1 PRIVILEGE LEVELS 44
4.2 MENUS 45
4.2.1 File Menu 45
4.2.2 Edit Menu 46

5. DEVELOPER APPLICATION NOTES 47

5.1 TEST EXECUTIVE ARCHITECTURE 47
5.2 TEST PROGRAM DESIGN 48
5.3 TEST OBJECT DESIGN 49
5.4 TEST OBJECT IMPLEMENTATION 51
5.5 TEST OBJECT OPERATION 52
5.6 AGENT INTERFACE 53
5.7 COMMAND LINE ARGUMENTS 54
5.8 COMPONENT VALIDATION 55
5.8.1 TPD Directory Structure 56
5.8.2 File Existence Checking 57
5.8.3 Version Checking 58
5.9 OPERATOR PRIVILEGES 59
5.10 POWER MONITORING 59
5.11 INDEPENDENT BREAKPOINT BEHAVIOR 59
5.12 PREREQUISITE TESTS 59
5.13 CONDITIONAL BRANCHING 59
5.14 SPLASH SCREEN 60

6. TEST EXECUTIVE FILES 61

6.1 ERROR LOG FILE 62
6.2 DEBUG LOG FILE 62
6.3 TESTEXEC INITIALIZATION FILE 62
6.4 TESTEXEC LOG FILE 63
6.5 TEST PROGRAM DEFINITION FILE 65
6.5.1 Test Program Definition Format 67
6.5.2 Identification 68
6.5.2.1 Identification Syntax 68
6.5.3 Options 69
6.5.3.1 Options Syntax 70
6.5.3.2 User Control Options 72
6.5.3.3 Report Printer 73
6.5.3.4 Test Type List 74
6.5.3.5 Breakpoint Settings 75
6.5.3.6 Custom Datasheet Header/Footer 75
6.5.4 Setup Objects 76
6.5.4.1 Setup Objects Syntax 78
6.5.5 Test Objects 79
6.5.5.1 Test Objects Syntax 80
6.5.5.2 Test Description 81
6.5.5.3 Test Object Command Line Arguments 82
6.5.5.4 Prerequisites 83
6.5.5.5 Test Branching 84
6.5.6 Action Objects 86
6.5.6.1 Action Objects Syntax 87
6.5.7 Power Monitor 88
6.5.7.1 Power Monitor Syntax 88
6.5.7.2 Power Monitor DLL 90
6.5.8 Other Files 91
6.5.8.1 Other Files Syntax 92
6.5.9 Configuration 93
6.5.9.1 Configuration Syntax 94
6.5.10 Configuration Labels 97
6.5.10.1 Configuration Labels Syntax 97
6.5.11 User Defined 99
6.5.12 Sample TPD Files 100

6.5.12.1 Simple TPD File 100
6.5.12.2 Complex TPD File 101
6.6 Test Object Definition File 103
6.6.1 Test Object Definition Format 103
6.6.2 TOD Identification 104
6.6.3 TOD Verifies 105
6.7 Test Program Configuration File 106
6.8 Test Executive Configuration File 107
6.8.1 Test Station 108
6.8.2 Options 109
6.8.3 Log File Management 110
6.8.4 Security 111
6.8.5 Test Program List 112
6.9 Microsoft Access Database File 113
6.9.1 Test Program Configuration Table 114
6.9.2 Test Sequence Table 115
6.9.3 Test Object Event Table 115
6.9.4 Verify Event Table 116
6.10 Parametric Data Log File (IEEE-1545) 117

7. PROGRAMMING INTERFACE 119

7.1 Test Object Programming Interface 119
7.2 TEXDLL Summary 120
7.3 Access Functions 122
7.3.1 Behavior Modes 123
7.3.2 Command Flags 124
7.3.3 TPD Reload 125
7.3.3.1 Abort Recovery 125
7.3.3.2 UUT-Adaptation 126
7.3.3.3 Detailed Behavior 127
7.4 Verify/Compare Functions 128
7.4.1 Verify Breakpoints 129
7.4.2 Test Criteria 130
7.4.3 Invalid Test Criteria 131
7.4.4 Significant Digits 132
7.4.5 Test Methods 133
7.5 User Interface Functions 134
7.5.1 Custom Dialog Titles 135
7.6 Information Functions 136
7.7 Other Functions 137
7.7.1 Embedded Breakpoints 137
7.8 TOD Files 138
7.9 Test Object Help Files 138
7.10 Test Objects and Microsoft Windows 139
7.10.1 Windows Compliance for Test Objects 139
7.10.2 Windows GUI and Multitasking 140
7.10.3 Aborting a Test 141
7.10.4 Aborting a Standalone Test Object 142
7.10.5 Abort Delay 144
7.11 DLL Test & Setup Objects 145
7.11.1 DLL Loading and Execution 145
7.11.2 DLL Dependencies 146
7.11.3 DLL Startup 147
7.11.4 DLL Abort 148
7.11.5 DLL Debugging 149

7.11.6 DLL Error Recovery 150
7.12 ACTIVEX DLL TEST OBJECTS 151
7.12.1 Building an ActiveX Test Object 151
7.12.2 Naming an ActiveX Object 152
7.12.3 Object Creation and Shared Memory 152
7.12.4 Aborting an ActiveX DLL 153
7.12.5 Debugging an ActiveX DLL 154
7.12.6 Registering ActiveX DLLs 155
7.12.7 ActiveX Test Object Syntax 155

8. TEXDLL FUNCTION DEFINITIONS 157

8.1 TEXABORT 158
8.2 TEXCOMPARE 159
8.3 TEXCOMPARETOD 161
8.4 TEXEMBEDBREAKPOINT 162
8.5 TEXERRORBOX 163
8.6 TEXFINISH 164
8.7 TEXGETTODVERIFY 165
8.8 TEXGETTPC 167
8.9 TEXGETTPD 168
8.10 TEXINPUTNUMBER 170
8.11 TEXINPUTSTRING 171
8.12 TEXLOGPDEL 172
8.13 TEXMESSAGEBOX 173
8.14 TEXPOWERMONITOR 174
8.15 TEXPRINT 175
8.16 TEXSCALE 176
8.17 TEXSETABORTDELAY 177
8.18 TEXSTART 178
8.19 TEXSTARTACTIVE 180
8.20 TEXVERIFY 181
8.21 TEXVERIFYTOD 183
8.22 TEXWARNINGBOX 184
8.23 TEXYESNOBOX 185

9. APPENDIX A - FREQUENTLY ASKED QUESTIONS 187

10. APPENDIX B - UNITS AND PREFIXES 193

11. APPENDIX C - PROBLEMS AND NOTES 195

11.1 REPORT GENERATOR & PRIVILEGE EDITOR CRASH ON EXIT 195
11.2 DATABASE "OUT OF MEMORY" ERROR 196

12. APPENDIX D - GLOSSARY OF TERMS 197

13. APPENDIX E - FORMATTING MASK 199

14. APPENDIX F - VERSION CHANGES 201

15. APPENDIX G - SAMPLE TEST OBJECT 215

16. SERENDIPITY SYSTEMS, INC 217

1. Introduction

This describes the operation of the Test Executive, developed by Serendipity Systems, Inc. (SSI). The SSI Test Executive is a Windows application for controlling test program configuration, test execution, data logging and reporting.

The Test Executive has an easy-to-use graphic interface for presenting tests and controlling test selection, execution, and behavior options. Novice Windows users with no programming experience can use the Test Executive. User input is provided by point-and-click interfaces; file browsers and selection lists. The user interface can be resized, minimized and maximized using standard Windows controls; and it supports a configuration file that automatically saves and restores window settings.

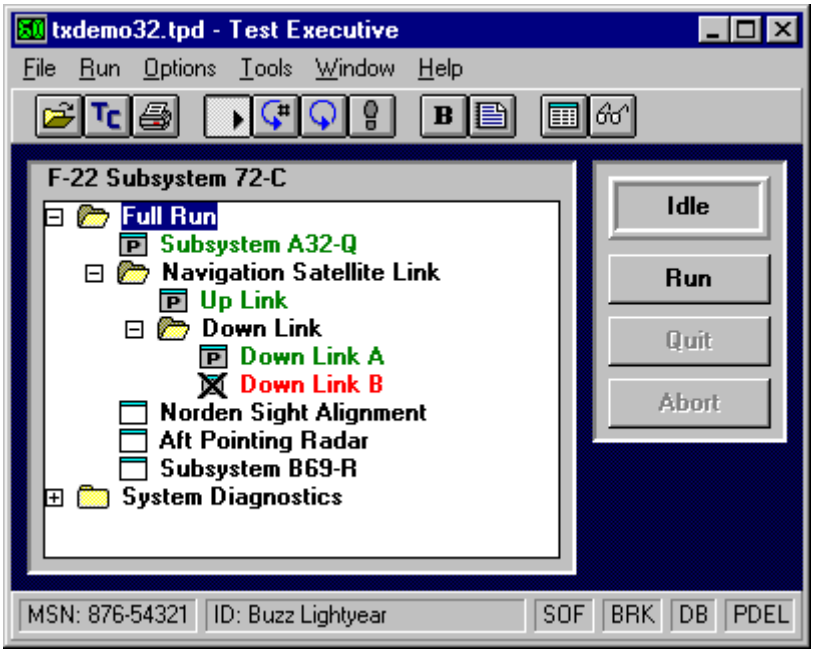
Tests are displayed for selection in a hierarchical list where each entry represents one test. An end-to-end test is represented as a set of subordinate tests under one parent in the hierarchy. Selecting and executing a parent test automatically runs the end-to-end test. Subordinate tests can be independently selected and executed as well. Behavior options allow the user to specify looping and stepwise execution, stop-on-fail behavior, and breakpoint control. An Execute button starts execution of the current test, a Quit button interrupts the current execution, and an Abort button resets the current execution and runs a specified recovery procedure.

A companion application, the Report Generator, provides for viewing and printing logged test results.

The behavior of the Test Executive is largely defined by the Test Program Definition (TPD) file, which is created by a test developer. Test objects (i.e. executables) are written using functions provided by the Test Executive's TEXDLL function set.

2. Test Executive

The Test Executive has an easy-to-use interface for controlling a set of integrated tests. With it, a test operator can load a test program, select a test for execution, set breakpoint options, start a test, view test results, print a datasheet and run external utilities.

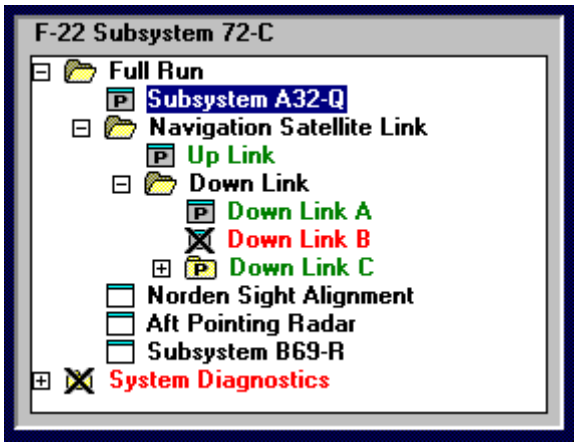


Test program loading, and data sheet printing, are controlled from the File Menu or toolbar. The test hierarchy is presented in the Test List window where tests are selected for execution. Test execution is handled by the Run Menu and a set of control buttons on the right side of the window. Diagnostic control is available from the Breakpoint dialog. Data logging options are managed by the Options Menu and their settings are displayed on the window's status bar. The Window Menu manages auxiliary windows for viewing test results and debug information. The Report Generator is accessed from the Tools Menu along with TPD-specific external utilities. Finally, the Help Menu accesses test program and Test Executive help files.

> **Note: Some Test Executive capabilities may be disabled if operator privileges are active.**

2.1 Test List

The Test List window displays all tests available for execution in an ordered, hierarchical list. This list is composed of executable test objects and test groups represented by status icons. Test groups contain (own) one or more test objects and display a result status when collapsed (i.e. pass, fail, and not all tested). Test groups are denoted by various file folder icons. Test objects have "executable window" icons, similar to those in the Windows' Explorer. Each test group can have one or more subordinate tests below it in the hierarchy and these are shown by indentation. If the entire list cannot be viewed at once, a vertical scroll bar is automatically provided to traverse the list.



A selected test object is run by double clicking on the test, pressing F5, or by pressing the Execute control button. Execution of a selected test group automatically includes all of its subordinate tests. If the selected test has no subordinates, then only one test is run.

Subordinate tests are expanded (made visible) and collapsed (hidden) by pointing to the test group name and double-clicking with the left mouse button. Alternately, a single click on the plus (+) or minus (-) symbol expands or collapses a test group.

Additional test information for an item in the Test List is viewed by holding the right mouse button down while over it. Context-sensitive help for a selected test, if available, is accessed by pressing the **F1** function key.

The Test List also supports selection and execution of multiple test objects. Individual test objects are selected in two different ways. Use the mouse to select (click and hold) the first test object and then drag across subsequent test objects; or press the *Ctrl* key and left-mouse click all desired test objects. When the required test objects are selected, press the run button.

> **Note: Multiple test selection is enabled or disabled based on the contents of the currently loaded TPD file.**

2.1.1 Test List Status

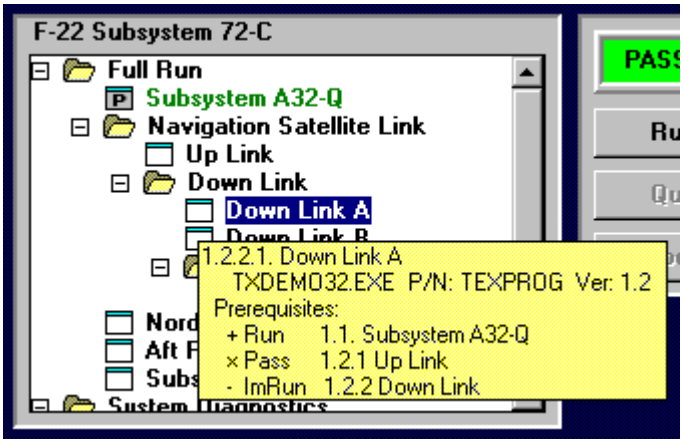
The Test List contains a hierarchical list of test groups and test objects. During operation, the icons used for these items represent their current test status. The test status is reset whenever a new TPD is loaded, a new UUT MSN is entered or the operator chooses (via the Run Menu). The following shows the individual icons for test groups and test objects along with their respective pass or fail status.

	Test Group- contains (owns) one or more test objects.
	Test Group- contains (owns) one or more test objects and all test objects completed execution with a “Pass” result.
	Test Group- contains (owns) one or more test objects that completed execution with a “Fail” result.
	Test Group- in an expanded state (with test objects visible).
	Test Object - not executed.
	Test Object- executed with a Pass result.
	Test Object- executed with a Fail result.
	Test Object- executed with an Error.

2.1.2 Viewing Test Information

Information about a specific item in the Test List is viewed by pressing and holding the right mouse button on a test or test group. This causes a window to appear that contains the test name, paragraph number, executable file, TOD file, command line arguments and prerequisites. If defined, version information is displayed for the specified files.

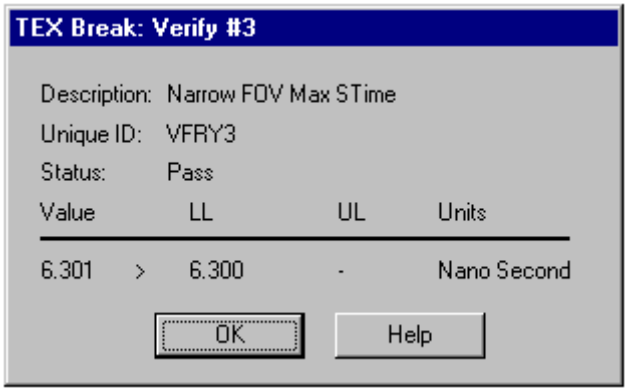
The prerequisite list has all the tests that must pass (Pass) or be executed (Run) before the selected test is run. A leading symbol on each line indicates whether the prerequisite has been met (+) or not (x). Test objects also inherit prerequisites from parent test groups. Immediate prerequisites (ImPass and ImRun) are dynamically evaluated during the execution of a test sequence. Consequently, they are always represented by a minus sign (-).



2.1.3 Test Program Help

If a Window's Help file is defined for the test program it can be accessed in three ways. The first way is through the Help Menu. The second way is by a context-sensitive link through the Test List. When **F1** is pressed, the currently selected test becomes a link into the test program help file. This allows the operator to jump immediately to a help topic of interest.

The third way to access test program help is through a verify breakpoint dialog. When a verify breakpoint occurs, the dialog has a Help button that is a context-sensitive link into the test program help file. The unique verify id is the context link used for accessing help. This allows immediate access to information specific to the current verify. The Help button is hidden when test program help is not available.



2.2 Test Execution

Activities for executing a test include the following:

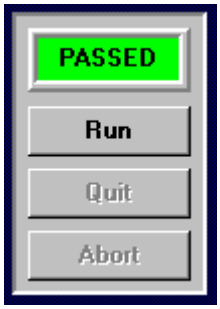
- A test is selected for execution in the Test List window.
- Test behavior options are selected using the Run Menu. Alternately, four toolbar buttons support selecting Run, Loop#, Loop, and Single Step modes.
- Breakpoint control is accessed via the Options Menu.
- Pressing the Execute button, double clicking a test object, or pressing **F5** starts the run process for the selected item.
- The Quit button is used to halt execution once the current test has completed.
- The Abort button interrupts the current test and resets the test equipment by executing any abort setup objects.

The Test Executive shows execution status at all times in a button-sized display pane located in the window's upper, right-hand corner. Test output is scrolled in the Test Results window. The Test Results window can be displayed using the Window Menu or by clicking on the Test Results toolbar button.

When executing a test, two conditions may cause the Test Configuration window to be presented. This occurs when the TPC data is incomplete and when the test configuration prompt option is enabled in the TPD file.

2.2.1 Control Buttons

The Test Executive has three control buttons that are used to start a test, stop a test, and interrupt a test. The control buttons appear in the upper, right-hand corner of the window. They are immediately below the Execution Status display and above the optional Power Monitor display.



Execute is the top command button. It is used to start execution of the selected test, using the current control parameters and breakpoints. To execute selected tests, press **F5** or use the left mouse button to click on the Execute button. The label on the Execute button changes to display the current execution mode: **Run**, **Loop#**, **Loop**, **Step**, or **Next Step**.

Quit is the middle command button. This button stops test execution, without resetting any instrumentation, by allowing the currently selected test object to finish executing. Quit is enabled only while a test is executing.

Abort is the bottom command button. The abort button interrupts the current test execution and runs the abort executables that are defined by the TPD file. Typically this will reset/reinitialize instrumentation. Test results are not logged if an abort occurs.

When executing a test, two conditions may cause the Test Configuration window to be presented. This occurs when the TPC data is incomplete and when the optional test configuration prompt is enabled in the TPD file.

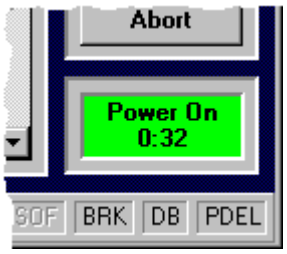
2.2.2 Execution Status

The Execution Status display appears in the upper, right-hand corner of the Test Executive window. The display is immediately above the three Control Buttons. The status display presents one of the following states:

Idle	No test executing. Pass/fail status is unknown.
Running	Executing a single pass.
Setup	Running Pre or Post execution setup objects.
Looping #	Executing in a loop a user-defined number of times.
Looping	Executing in a loop.
Stepping	Executing stepwise.
PASSED	Test completed; no failure reported.
FAILED	Test completed with failure or halted due to failure.
ABORT	Abort procedure is being executed.
Error	Error occurred in execution. Test did not complete.

2.2.3 Power Monitor Display

The Power Monitor display is located below the Control Buttons on the right-hand side of the Test Executive interface. This display is only visible when power monitoring has been enabled by the current TPD file. The display identifies the power condition (e.g. Power On) and the time remaining for operation. The background of the display is green while power is on, and gray when power is off.



During normal operation, the time display indicates the remaining time available for applying power to the current UUT. The time value decrements while power is on. When the remaining time is less than 30 seconds, the background of the display is yellow. If the time decrements to zero, an abort operation is initiated to halt the current test. During this the Power Monitor display has a red background and the power condition is **SHUTDOWN**. Once power has been turned off, typically by an abort setup object, the power condition is **Recover** and the background color is purple. During a recovery phase the time is displayed within angle brackets (e.g. < 2:42 >) and it indicates the time remaining before testing can resume.

2.2.4 Custom Logo Display

Beneath the control buttons, and optional power monitor display, is an area that displays a custom graphic. The graphic information is read from a file (**TexLogo.bmp**) that can reside in the TPD directory, its parent directory or the application directory (typically C:\Program Files\SSI Test Executive). The graphic file is searched for in that order, so different on-screen graphics can be associated with different TPDs. If no graphic file is found, nothing is displayed.

2.2.5 Test Control Options

The Test Executive supports four options for controlling test behavior. Execution mode and fault behavior options are selected using the Run Menu.

Execution mode is selected using the **Run Mode**, **Loop# Mode**, **Loop Mode**, and **Single Step** options in the Run Menu. Four toolbar buttons also provide convenient selection of the execution mode. The current execution mode is displayed as the label of the Execution button.

	Run Mode performs one execution of the selected test and all subordinates. Execution stops automatically after the final test, or due to Stop-on-Fail behavior.
	Loop# Mode executes the selected test and all subordinates a certain number of times as defined by the operator.
	Loop Mode continuously executes the selected test and all subordinates. After the final test, execution begins again at the first test. Looped execution continues until stopped by the Quit or Abort button.
	Step Mode invokes stepwise execution of the selected test and all subordinates. Each time the Execution button is pressed, the selected test is executed and selection is automatically advanced to the next subordinate test. After the final test, selection is reset to the initial selected test. When stepping through an execution, certain menu items, and the toolbar, are disabled. Use the Quit button to end stepwise execution.

In all four execution modes, hidden subordinate test levels are automatically expanded as needed to advance selection of the next test.

Fault behavior is selected using the **Stop-on-Fail** option in the Run Menu. When selected, test execution halts when a failing test object is encountered. The stop-on-fail behavior mode is displayed at the bottom of the screen, in the third pane of the status bar.

When operator privileges are active, operators without a Flow privilege are limited to Run Mode only.

2.3 Status Bar

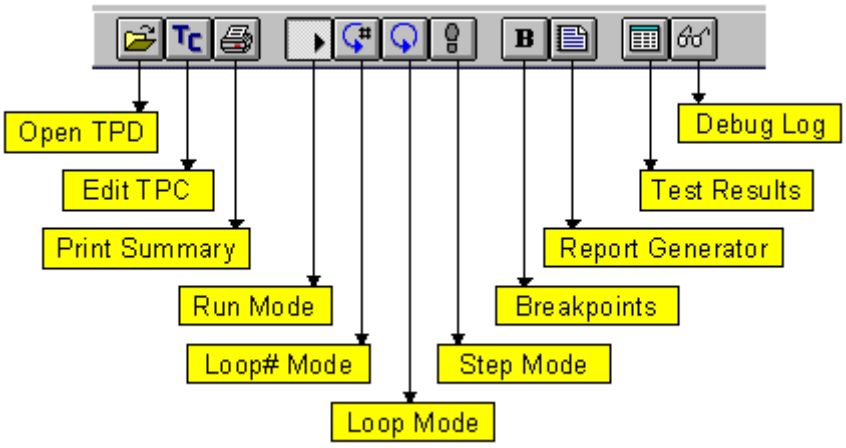
The status bar is a group of six display panes located at the bottom of the Test Executive window. The left-most (first) pane displays the current MSN. The second pane displays the current test operator's identification. The third pane reflects whether or not Stop-on-Fail has been enabled. The fourth pane reflects whether or not breakpoints have been enabled. The fifth pane is enabled if the Output Database option has been selected from the Options Menu. Finally the sixth (right most) pane is enabled if the output PDL option has been selected from the Options Menu. It also indicates the type of ASCII data logging that is active (PDEL or PDL).

2.4 Toolbar

The Test Executive toolbar displays a row of control buttons that represent frequently used menu options. This convenient mechanism is often preferred for performing much repeated tasks and always appears directly below the menus.

Each Toolbar button has three states: up (normal), down (when clicked), and disabled. When the cursor is held over an enabled button, a short help message is displayed next to the cursor. The following diagram is a close-up of the Toolbar showing the menu option associated with each button.

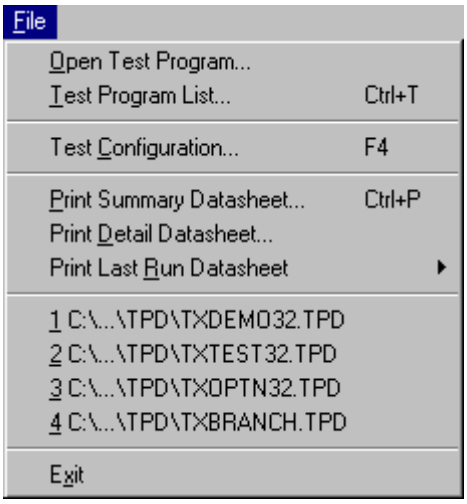
The "Print Datasheet" button is disabled if a printer is not installed.



2.5 Menus

2.5.1 File Menu

The Test Executive File Menu supports loading TPD files, setting Test Program Configuration (TPC) information, entering operator identification and printing summary or detail datasheets. A file history list permits quick reloading of recently accessed TPD files. Operator identification and the Test Program List are only available if specified in the Test Executive configuration file.

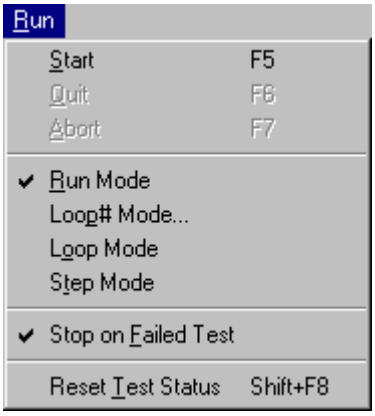


Menu Option	Description
Open Test Program...	Opens a file browser for selecting a TPD file. The selected file is loaded and displayed in the Test List Window.
Test Program List...	Opens a test selection dialog for choosing a test program. The test programs are listed by description, rather than by file name.
Test Configuration…	Displays the Test Configuration dialog for entering test-related information.
Operator ID…	Presents a dialog for entering operator identification.
Print Summary Datasheet…	Prints a datasheet summarizing the test results for the current UUT; or runs a custom report program if defined by the TPD file. This option is disabled if a printer is not installed.
Print Detail Datasheet…	Prints a detail datasheet for the current UUT. The datasheet contains the most recent test results for each of the items in the Test List. This option is disabled if a printer is not installed.
Print Last Run Datasheet	Prints a detail or summary datasheet on the most recent test sequence. This datasheet contains all test objects that were executed, including all repetitions and loops. Database logging must be active for this option.
File History	This provides a quick selection from the last four TPD files opened. The list is updated each time a new file is opened.
Exit	Closes the Test Executive. All remote applications are halted. The current window settings, positions, and control selections are saved in an initialization (INI) file.

2.5.2 Run Menu

The Test Executive Run Menu allows a user to start, quit, and abort tests, specify options to control test execution, set Stop-on-Fail mode and Reset Test Status.

Note that the run modes are mutually exclusive.

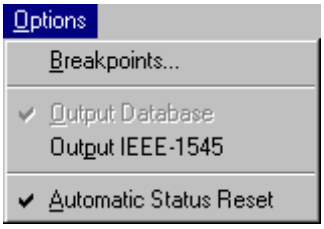


Menu Option	Description
Start	Starts the selected test(s). Also accomplished by pressing **F5** or the execute button.
Quit	Stops a running test sequence after the current test object completes its execution. Also accomplished by pressing **F6** or the Quit button.
Abort	Aborts a running test sequence, terminates current test object and resets test equipment. Also accomplished by pressing **F7** or the Abort button.
Run Mode	The selected test and all subordinate tests are run once when the Run button is pressed.
Loop# Mode…	The selected test and all subordinate tests are run a specified number of times in a loop, when the Run button is pressed. Use the Quit or Abort buttons to stop execution.
Loop Mode	The selected test and all subordinate tests are run continuously, in a loop, when the Run button is pressed. Use the Quit or Abort buttons to stop execution.
Step Mode	The selected test and all subordinate tests are run, one at a time, for each press of the Run button.
Stop on Failed Test	Stops the testing sequence when a test fails.
Reset Test Status	Resets the status of all test groups and test objects to *not executed*. This does not affect previously logged test results. Also accomplished by pressing **Shift+F8**

2.5.3 Options Menu

The Test Executive Options Menu allows the user to set breakpoints, control data logging and select display status reset. The status bar indicates the state of most of these options. The current TPD file may force data logging to be always on (grayed & checked), never on (grayed only), or left to the user's choice. Automatic Status Reset is similarly defined by the currently loaded TPD file.

The default ASCII logging format for the Test Executive is IEEE-1545. The logging format can be defined in the Test Executive Configuration file for a test system, or in a TPD file for an individual test program. The active logging type is displayed on this menu and the status bar (PDEL or PDL). By default, logged data is discarded when an Abort or Quit occurs. This behavior can be overridden by the TPD file.



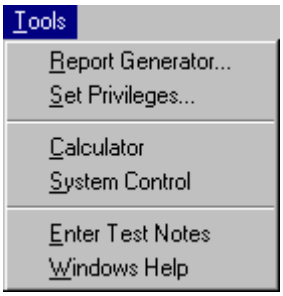
Menu Option	Description
Breakpoints...	Displays the Breakpoint Control dialog which allows various breakpoint options to be set.
Output Database	Enables/disables test result output to a Microsoft Access database. One database is created for each UUT MSN. These databases are used by the Report Generator.
Output IEEE-1545	Enables/disables logging to a Parametric Data Log (PDL) file. When enabled, a PDL file is created, in the TPD's PDEL subdirectory, for each test execution. Each file contains a set of test conditions and accumulated measurements from the test objects.
Automatic Status Reset	Enables/disables automatic resetting of the displayed test status. When enabled, the test status is reset each time the Run button is pressed.

2.5.4 Tools Menu

The Tools Menu provides access to Test Executive utilities and third-party applications. The third-party tools are defined by the currently loaded TPD file. The test developer uses the TPD file to configure the Tools menu, designating menu shortcut keys and grouping tools by function.

> **Note: The Set Privileges menu item is only visible for an operator with Administration privileges.**

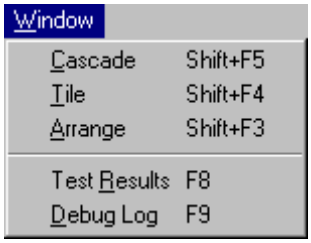
Selecting a tool from the menu executes the corresponding program. Tools can be set to be disabled while a test is running.



Menu Option	Description
Report Generator...	Starts the Report Generator for viewing and printing detailed or summary UUT datasheets.
Set Privileges...	Starts the Privilege Editor for creating and modifying Test Executive operator privileges.

2.5.5 Window Menu

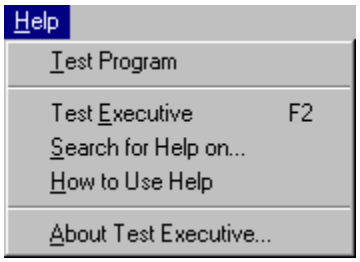
The Window Menu controls the placement of the Test Executive interface, Test Results, and Debug Log windows in cascaded, tiled or arranged modes. This menu also displays, or activates, the Test Results and Debug Log windows.



Menu Option	Description
Cascade	Displays a cascade of all open Test Executive windows.
Tile	Displays all open Test Executive windows in a horizontal tile format
Arrange	Causes all open Test Executive windows to be grouped in the middle of the screen.
Test Results	Toggles between the Test Results window and the Test Executive. Function key **F8** also performs this task.
Debug Log	Toggles between the Debug Log window and the Test Executive. Function key **F9** also performs this task.

2.5.6 Help Menu

The Help Menu provides access to Test Program help, Test Executive help and application information. The Test Program help file, if available, provides information about the currently loaded TPD. It can also be activated by pressing the **F1** function key.

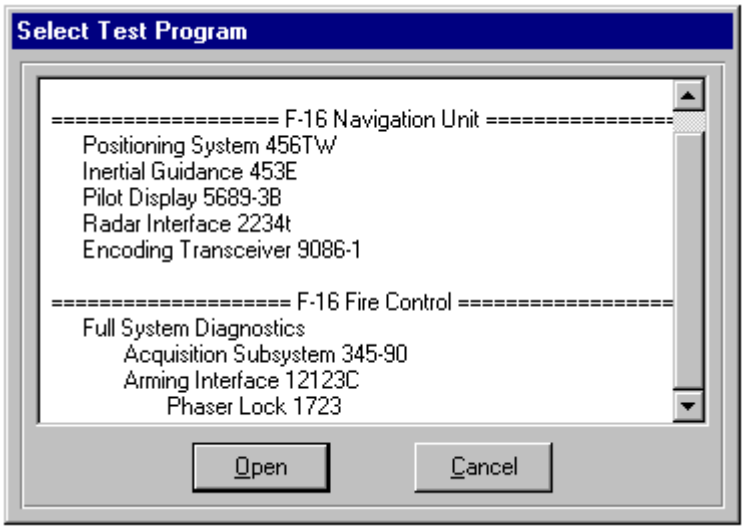


Menu Option	Description
Test Program	Opens Test Program help, if specified in the TPD file, to its contents page. Pressing function key **F1** activates context-sensitive help based on the test selected in the Test List.
Test Executive	Opens the Test Executive Help file at the Table of Contents.
Search for Help on…	Opens a Help Search dialog box. This provides access according to an alphabetical search of keywords and topics.
How to Use Help	Opens a How to Use Help contents dialog box.
About Test Executive…	Displays a dialog box reporting the Test Executive version, copyright and serial number.

2.6 Dialogs and Windows

2.6.1 Test Program List

The Test Program List is a simple, visual interface for selecting test programs to load. It is accessed from the Test Executive File Menu. Test programs in the list are typically identified by descriptive phrases or sentences. This makes locating and loading a test program much easier than navigating a directory structure, with a file dialog box, while looking for a specific TPD file.



Note: The Test Program List is only available if it has been defined in the Test Executive Configuration File.

A test program is selected with a left mouse click, or by scrolling with the vertical arrow keys. Pressing the Open button, or the Enter key, causes the test program to be loaded. Pressing the Cancel button, or the Esc key, causes the dialog to close without loading a test program.

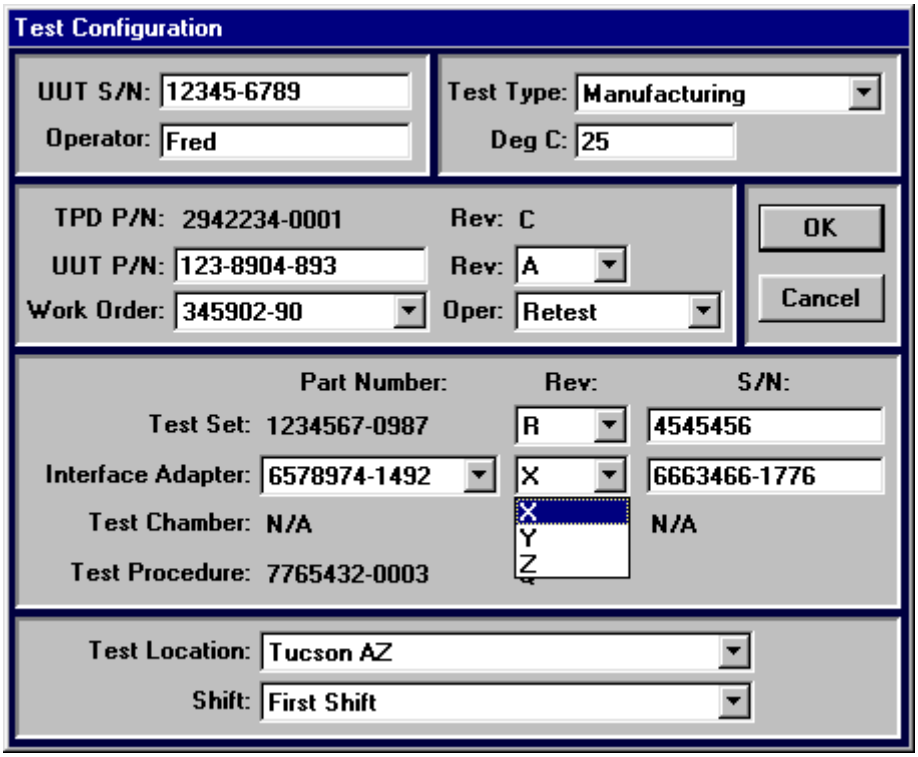
The Test Program List is moveable and sizable according to standard Windows conventions. It can be automatically displayed, when the Test Executive starts, by specifying a command line argument (**/tpl**).

2.6.2 Test Configuration

The Test Configuration dialog is activated from the File Menu or toolbar. This dialog has fields for defining all of the parameters associated with a test. The parameter values are stored, with the test results, in PDL and/or database files. The Test Executive does not execute a test unless all editable fields have been populated.

The parameter fields are configured by the current TPD file. Each field can be static (gray background), editable (white background), a drop-down list or ignored (N/A). Additionally, editable fields may have a specific format for the data that can be entered. The exact appearance of the Test Configuration dialog may differ from that shown below. Some of the parameter fields can be renamed in the TPD file to match local data requirements.

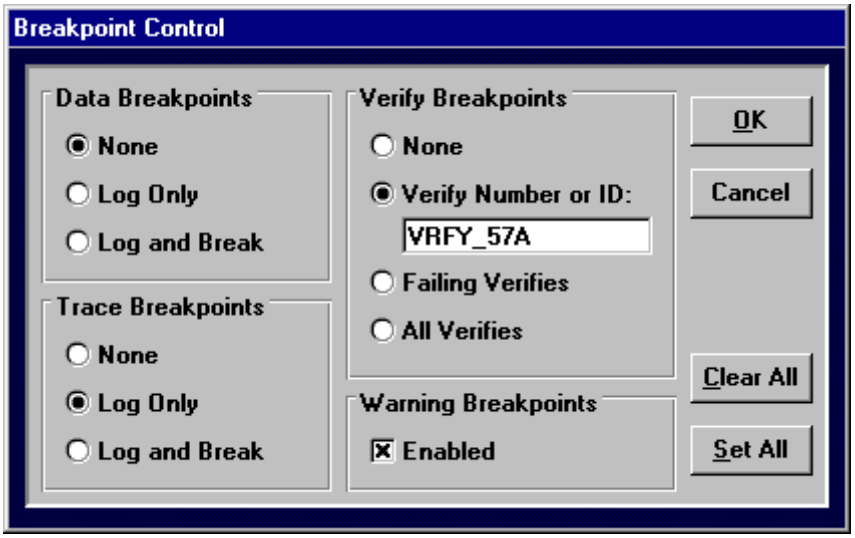
When selected, a field with a required data format is denoted by underscore placeholders and literal characters (dashes, parenthesis, etc.). When unselected, an incomplete data format field has a dark background. **Note: When a field requires a specific data format, underscore characters are not accepted as input.**



Control Option	Description
UUT S/N	Allows for entry of a UUT S/N (Serial Number). When a new UUT S/N is entered, the test status resets, the Test Result and Debug Log windows clear, a new database is created for use by the Report Generator and the PreMSN and PostMSN procedures are executed (if defined in the TPD file).
Operator ID	Allows the user to input name or ID number.
Test Type	Allows the user to select applicable test types. *(Acceptance, Manufacturing, Calibration, Engineering,* or user defined)
Deg C	Allows input of a required or suggested test environment temperature.
TPD P/N and **Rev**	This static information always comes from the TPD file.
UUT P/N and **Rev**	Allows input of UUT P/N and Revision. Can be editable or static.
Work Order # and **Operation**	Allows input of work order # and operation. Can be editable, static, or N/A.
S/N	Allows input of S/N corresponding to data set label (Test Set, Test Chamber, Test Procedure, Test Adapter). Can be editable, static, or N/A.
Part Number	Allows input of Part Number corresponding to data set label (Test Set, Test Chamber, Test Procedure, Test Adapter). Can be editable, static, or N/A.
Rev	Allows input of Revision corresponding to data set label (Test Set, Test Chamber, Test Procedure, Test Adapter). Can be editable, static, or N/A.
OK button	Closes the Test Configuration window and saves all entered information. Configuration data is written out to a TPC file for use the next time the TPD file is loaded.
Cancel Button	Closes the Test Configuration window and ignores all changes to the data.

2.6.3 Breakpoint Control

The Breakpoint Control dialog is used to set break options on all tests. When a breakpoint is encountered, a window appears with information relative to the cause of the breakpoint. Test execution is suspended until the operator responds. This control is accessed via the Options Menu or the toolbar. When operator privileges are active, operators without a diagnostic privilege are only allowed to view the breakpoint settings..



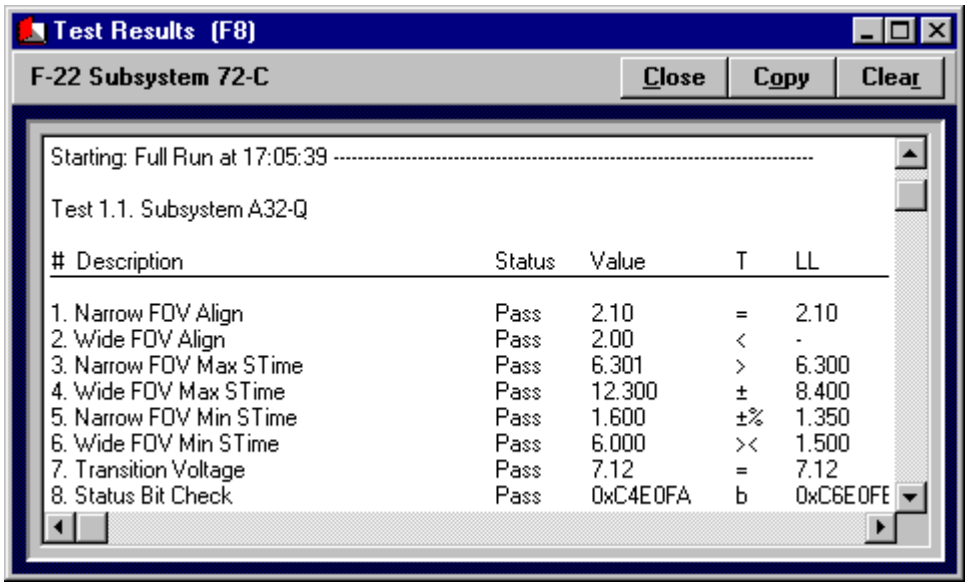
Control Option	Description
Data Breakpoints	Allows the operator to log (Debug Log) only or to log and break at program defined embedded data points in the test program.
Trace Breakpoints	Allows the operator to log (Debug Log) only or to log and break at program defined embedded trace points in the test program.
Verify Breakpoints	Allows the operator to set breakpoints on specific verifies, failing verifies, or all verifies.
Warning Breakpoints	Allows the operator to enable breakpoints for all warnings during the test. Warnings from the test program are always logged automatically to the Debug Log.
Clear All	Clears all breakpoint controls.
Set All	Sets all breakpoint controls.

2.6.4 Test Results

The Test Results window displays a history of test execution. The user is provided with vertical and horizontal scroll bars for reviewing this information. Key commands (PgUp, PgDn, Home, etc.) can also be used to move around the display. While the Test Executive is open, a test execution history is accumulated and stored in a large text buffer. When the history buffer becomes full, the oldest portion of the contents is deleted to make room for newer information.

In the Test Results window, the beginning and end of each execution is automatically reported by a header line that includes the selected test name and a time stamp. If subordinate tests are included in the execution, one header line is reported at the start of each subordinate test. Headers for subordinate tests do not include a time stamp. The Test Results window is automatically cleared when a new UUT MSN is entered or a new TPD is loaded. The results can be selected and copied to the clipboard, or cleared manually.

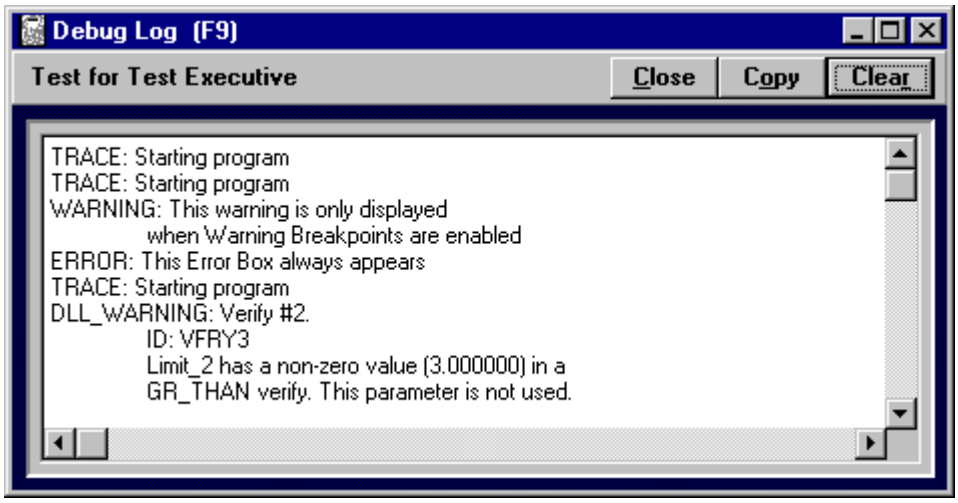
Function key **F8** toggles between the Test Executive interface and the Test Results window.



2.6.5 Debug Log

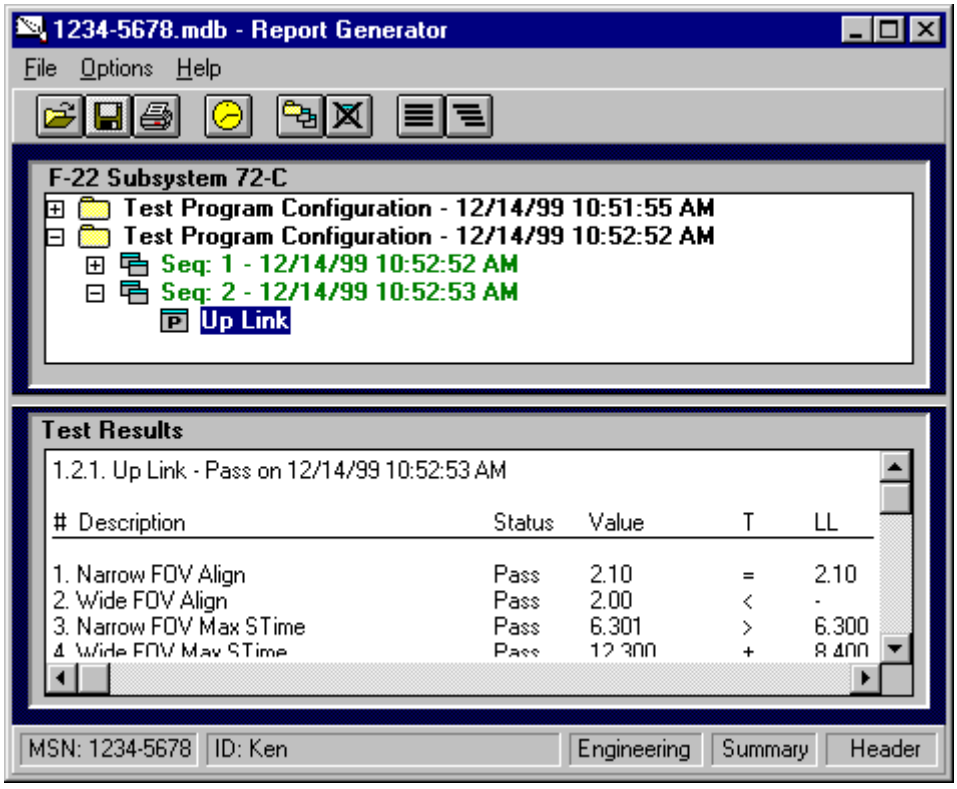
The Debug Log window behaves in a similar manner as the Test Results window. It displays logged data, warnings, and errors from the test object executions. The window is automatically cleared when a new UUT MSN is entered or a new TPD is loaded. A full copy of the Debug Log is maintained in the file, DEBUG.LOG (located in the Test Executive execution directory), until a new TPD or UUT MSN is entered. The results from the Debug Log pane can be selected and copied to the clipboard, or cleared manually.

Function key **F9** toggles between the Test Executive interface and the Debug Log window.



3. Report Generator

The Report Generator is an application for selecting, viewing and printing test result databases generated by the SSI Test Executive. In the Report Generator, test sequences are displayed as a hierarchy in the Test List. The time and date of each test sequence is listed chronologically under a test configuration. The names of the tests contained in the sequence are displayed as indented entries under the sequence heading.

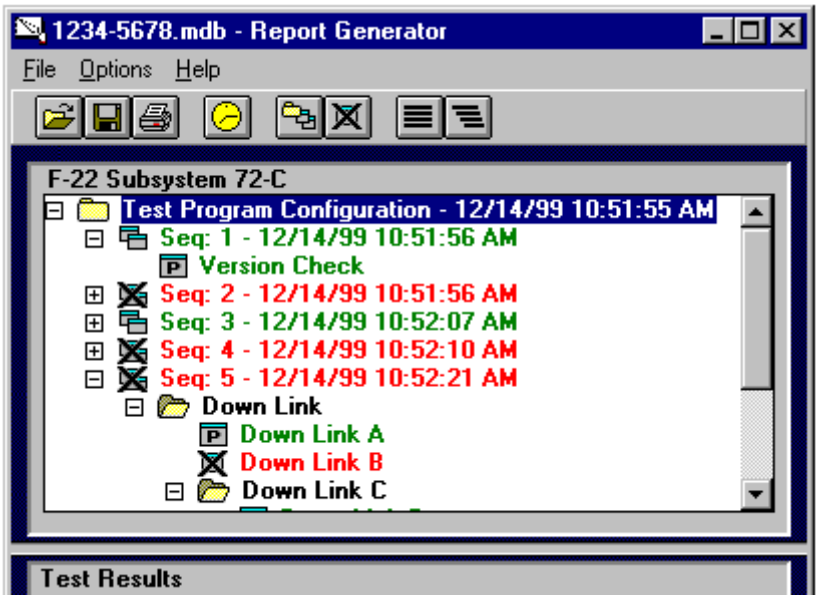


The Report Generator has the following capabilities:

- Open a database file for an individual UUT using the File Menu.
- Display test sequences and run times in the Test List.
- Display test results and configuration in the Test Results.
- Select summary or detail datasheets from the Options Menu.
- Print datasheets comprised of the selected items in the Test List.
- Use split bar to divide space between the Test List and Test Results pane.
- View file version checking and control results.

3.1 Test List

The Report Generator Test List presents a chronological hierarchy of test configurations, test sequences and individual tests. The status and type of each item is represented by icons and color coding. The hierarchy can be expanded and collapsed, by double-clicking, to change the detail level of the display. When a single item is selected, corresponding information is displayed in the Test Results pane.



The Test List is used to select the items desired for a printed datasheet. Multiple items are selected in a manner similar to the Windows Explorer. Individual items are selected, or deselected, by holding the Control key down while single-clicking with the left mouse button. Groups of items are selected by holding the Shift key down while single-clicking with the left mouse button, or by holding the left mouse button down and dragging over the items to select.

The height of the Test List is adjusted by dragging the **split bar** up or down. The split bar is the horizontal line that appears between the Test List and the Test Results pane. The current Test Program name is displayed above the Test List.

A test object labeled "**Version Check**" denotes the results of version checking the contents of the corresponding TPD file. The data displayed and printed is similar in format to the verify information from a test execution. The leftmost column contains the file that was version checked. This is followed by the check results (equal or greater), file date, expected date limits and version number.

> **Note that a sequence with a passing status may contain a failing test object if that test object was executed multiple times and eventually passed (e.g. "If Fail Then Repeat(5) While Pass").**

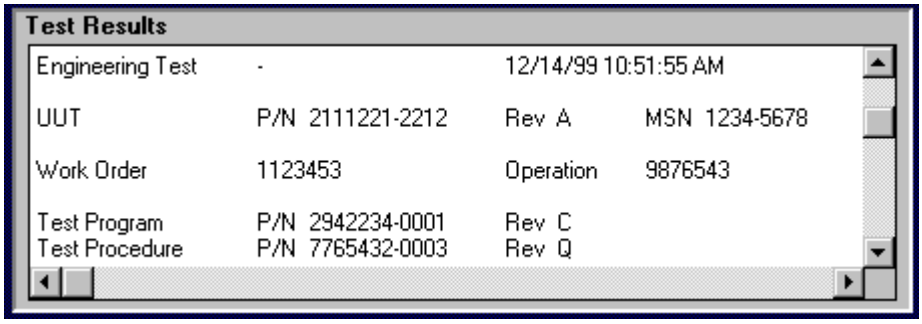
3.2 Test List Status

All of the items in the Report Generator Test List use icons and colors to denote status. Passing test sequences, test groups, and test objects have green text. Failing ones have red text. Test groups reflect the status of their child test objects when they are collapsed.

	Test Configuration - contains (owns) one or more executed test sequences.
	Test Sequence - contains (owns) one or more test objects and all test objects completed execution with a “Pass” result.
	Test Sequence - contains (owns) one or more test objects that completed execution with a “Fail” result.
	Test Group - in an expanded state (with test object(s) visible).
	Test Group - in a collapsed state, contains (owns) one or more test objects that have all passed.
	Test Group - in a collapsed state, contains (owns) one or more test objects that completed execution with a “Fail” result.
	Test Object - executed with a Pass result.
	Test Object - executed with a Fail result.
	Test Object - executed with an Error reported.

3.3 Test Results

Test information is displayed in the Test Results pane by single-clicking an item in the Test List. Single-clicking on a test group, or test sequence, displays information about Test Program Configuration. Select an individual test to display verify data from its execution. The Test Results pane is cleared when multiple items in the Test List are selected. Vertical and horizontal scroll bars are provided for viewing the information. Use **Ctrl-C** to copy selected text, in this pane, to the Windows clipboard.



3.4 Status Bar

The Status bar is a group of five display panes located at the bottom of the Report Generator window. The left-most (first) pane displays the UUT MSN for the opened database. The second pane displays the operator's ID. The third pane shows the type of test execution for the currently selected item in the Test List. The fourth pane reflects the selected print option, Summary or Detail. The fifth pane is enabled if the Include Header option has been selected.

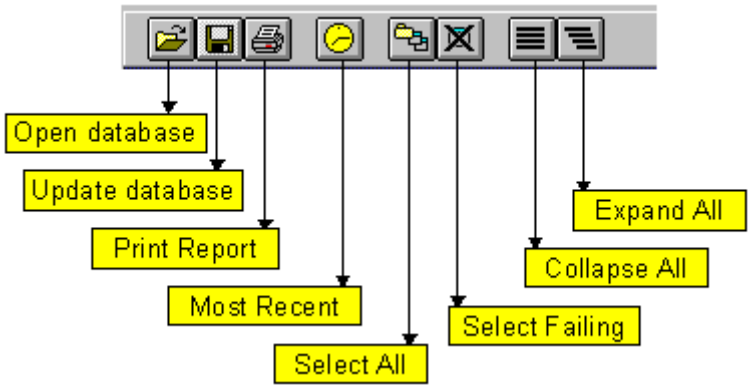
3.5 Toolbar

The Report Generator toolbar displays a row of control buttons that represent frequently used menu options. This convenient mechanism is often preferred for performing much repeated tasks and it always appears directly below the menus.

Each toolbar button has three states: up (normal), down (when clicked), and disabled. When the cursor is held over a button, a short help message is displayed next to the cursor. The following is a close-up of the toolbar buttons and description of their functionality.

The update database icon is disabled if the database has not changed since loading. The database is checked for changes every time a menu is opened or an item is selected in the Test List.

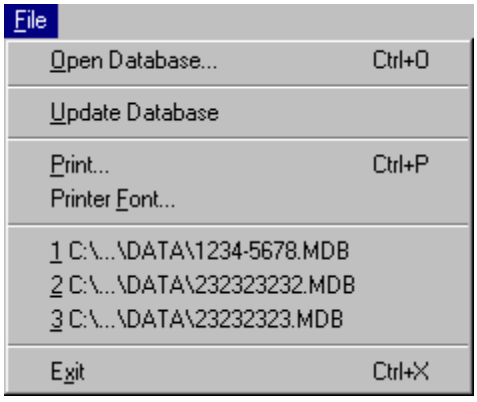
The "Print selected test results" button is disabled if a printer is not installed.



3.6 Menus

3.6.1 File Menu

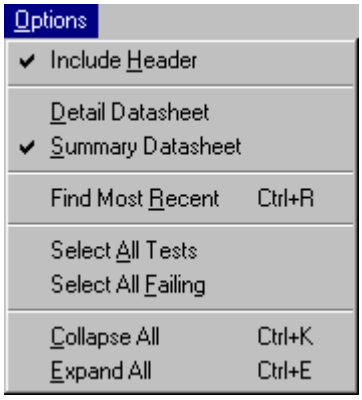
The Report Generator File Menu supports database loading and updating; datasheet printing and printer configuration. An historical list of the last four databases accessed is provided for quick retrieval. An Exit selection closes the Report Generator.



Menu Option	Description
Open Database	Opens a file browser for selecting a UUT database file. Once the selected file is loaded, the Test List displays the test configurations and sequences that were run on the UUT.
Update Database	Reloads the current database to update the Test List with additional test information. This option is only enabled when the current database has changed.
Print	Prints a datasheet containing selected items and any hidden child subitems in the Test List. The datasheet format is controlled by the Options Menu. This option is disabled if a printer is not installed.
Printer Font	Displays a Font dialog box which allows the user to select a font and font size for printed reports. This option is disabled if a printer is not installed. For convenience, the selected font and point size are stored in the initialization file.
File List	Displays up to the last four database files that have been opened. A single-click on a file name opens the corresponding database file.
Exit	Closes the application.

3.6.2 Options Menu

The Report Generator Options Menu provides control over the format and content of printed datasheets. The format options selected on this menu are also shown on the status bar. Many of the selection options are also available on the toolbar or by shortcut key.



Menu Option	Description
Include Header	Enables information from the Test Program Configuration table to be included on the first page of a printed datasheet. A two-line header is printed on top of the remaining pages. A single-click on this menu item selects or deselects this option. If Include Header is not selected, a two-line header is printed on top of each page.
Detail Datasheet	Selects a detailed format for printed datasheets. This format contains information on each verify in a test execution. This includes verify number, description, status, value, lower limit (LL), upper limit (UL), and units. A single-click on this menu item selects this option.
Summary Datasheet	Selects a summary format for printed datasheets. This format includes test paragraph number, test status, test name, and summary status. A single-click on this menu item selects this option.
Find Most Recent	Locates and selects the most recent test sequence (i.e. the last one executed). This aids an operator in quickly navigating to the item of most common interest.
Select All Tests	Selects all of the tests displayed in the Test List. Selected tests are used to create printed reports.
Select All Failing	Selects all of the tests displayed in the Test List that have a status of “Fail”. Selected tests are used to create printed reports.
Collapse All	Compacts the hierarchy of the Test List to only show the Test Program Configuration items.
Expand All	Unfolds the hierarchy of the Test List to show all test sequences and individual tests.

3.7 Datasheets

3.7.1 Printing a Datasheet

There are several different types of datasheets that can be printed by the Report Generator. A Summary datasheet lists the status of the selected tests without the details of individual verifies. A Detail datasheet includes the verifies for each selected test. A datasheet can have a minimal two line heading or the heading can include all of the test configuration information. The contents of a datasheet are built from the selected items in the Test List

Using a Mouse to Select Sequences and Tests

You can use the left mouse button to select one or more sequences or tests for printing. Before you can select items, they must be visible in the Test List.

To select a sequence or test

- Click the sequence or test name that you want to select.

To select two or more items in order

1. Click the first test or sequence that you want to select.
2. Press and hold down SHIFT while you click the last test or sequence in the group. Selecting two or more items in order is also known as extending a selection.

You can also extend a selection by clicking on the first item and holding the left mouse button down while dragging over the additional items to select.

To select two or more items out of order

- Press and hold down CTRL while you click each test or sequence.

To cancel a selection

- Press and hold down CTRL while you click the selected test or sequence.

Note that selecting a test group or sequence with hidden sub-items automatically tags each of the hidden items for printing.

When all of the report options have been selected, a datasheet is printed by selecting Print from the File Menu, or by pressing CTRL+P, or by pressing the printer toolbar button.

3.7.2 Printing to a File

You can easily print a datasheet to a file from either the Report Generator or the Test Executive. To accomplish this, there first must be a printer on your computer that has **File** as its output port. To redirect a printer driver to a file, do the following:

1. Double-click the My Computer icon to open it.
2. Double-click to open the Printers folder.
3. Click the icon for the printer you wish to change.
4. On the File menu, click Properties.
5. Click the Details tab, and then change the port to **File** in the Print To The Following Port box.

Then, when printing a datasheet, select the redirected printer from within the Print dialog box. You are prompted for a file name after you press the OK button. Depending on the printer you initially selected, the resulting file may contain formatting or font commands. To avoid this, you may install a Generic/Text Only printer driver and redirect its output to a file. A new printer is installed by the following steps:

1. Double-click the My Computer icon to open it.
2. Double-click to open the Printers folder.
3. Double-click the icon for Add Printer.
4. Follow the Wizard instructions to add a local, Generic, Text Only printer with a **File** port (You may need your Windows installation disk during this process).
5. Assign a unique name to the newly installed “file” printer.

The file generated by the Generic/Text Only printer driver is best viewed with Microsoft WordPad. Microsoft Notepad does not properly display the file’s line terminating characters.

3.7.3 Detail Datasheet Example

The detail datasheet lists each verify operation performed by the selected test objects. If a sequence is selected, it is included at the appropriate point with its date and time. If the verify data does not fit the page width, the printer font size is automatically reduced to make it fit (if possible). If the items selected for a report have different test configurations, a new page and heading is printed for each unique configuration. The datasheet heading is set on the Options Menu. It can be either a two-line minimum heading (see example below) or one that contains the complete test configuration information. Multiple page datasheets use the minimum heading after the first page. Custom datasheet headers, and footers, are optionally defined in the TPD file. These can include multiple lines, simple formatting and an embedded graphic image.

F-22 Subsystem 72-C Page 1

MSN: 1423578 Engineering Test Datasheet

Seq: 2 - 7/18/99 4:30:36 PM

1.2.2. Down Link - Group

1.2.2.1. Down Link A - Pass on 7/18/99 4:30:40 PM

#	Description	Status	Value	T	LL	UL	Units
1.	Narrow FOV Align	Pass	2.10	=	2.10	2.10	uV
2.	Wide FOV Align	Pass	2.00	<	-	2.20	uV
3.	Narrow FOV Max Stime	Pass	6.301	>	6.300	-	ns
4.	Wide FOV Max Stime	Pass	12.300	±	8.400	2.400	ns
5.	Narrow FOV Min Stime	Pass	1.600	±%	1.350	1.650	ns
6.	Wide FOV Max Stime	Pass	6.000	><	1.500	6.110	s
7.	Transition Voltage	Pass	7.12	=	7.12	7.12	mV
8.	Status Bit Check	Pass	0xC4E0FA	b	0xC6E0FB	0xF0001	-
9.	String Compare Result	Pass	-	p/f	-	-	-
10.	Maximum RAM Address	Pass	0xAAAAAAAA	=	0xAAAAAAAA	0xAAAAAAA	-

1.2.2.2. Down Link B - FAIL on 7/18/99 4:30:45 PM

#	Description	Status	Value	T	LL	UL	Units
1.	Narrow FOV Align	Pass	2.10	=	2.10	2.10	uV
2.	Wide FOV Align	Pass	2.00	<	-	2.20	uV
3.	Narrow FOV Max Stime	Pass	6.301	>	6.300	-	ns
4.	Wide FOV Max Stime	Pass	12.300	±	8.400	2.400	ns
5.	Narrow FOV Min Stime	Pass	1.600	±%	1.350	1.650	ns
6.	Wide FOV Max Stime	Pass	6.000	><	1.500	6.110	s
7.	Transition Voltage	Pass	7.12	=	7.12	7.12	mV
8.	Status Bit Check	Pass	0xC4E0FA	b	0xC6E0FB	0xF0001	-
9.	String Compare Result	FAIL	-	p/f	-	-	-
10.	Maximum RAM Address	Pass	0xAAAAAAAA	=	0xAAAAAAAA	0xAAAAAAA	-

3.7.4 Summary Datasheet Example

The summary datasheet lists the pass/fail status of the selected test objects. Sequence headings are included at the appropriate points with their date and time. If the items selected for a report have different test configurations, a new page and heading is printed for each unique configuration. The type of datasheet heading is set from the Options Menu. It can be either a two-line minimum heading or one that contains the complete test configuration information (as shown here).

F-22 Subsystem 72-C
Engineering Test Datasheet

UUT P/N 2111221-2212 Rev D
MSN 1423578

Work Order 1123453, Operation 9876543

Test Program P/N 2942234-0001 Rev C
Test Procedure P/N 7765432-0003 Rev Q

Test Set	P/N 1234567-0987	Rev Y	MSN 4545456
Interface Adapter	P/N 6578974-1492	Rev X	MSN 466-1776

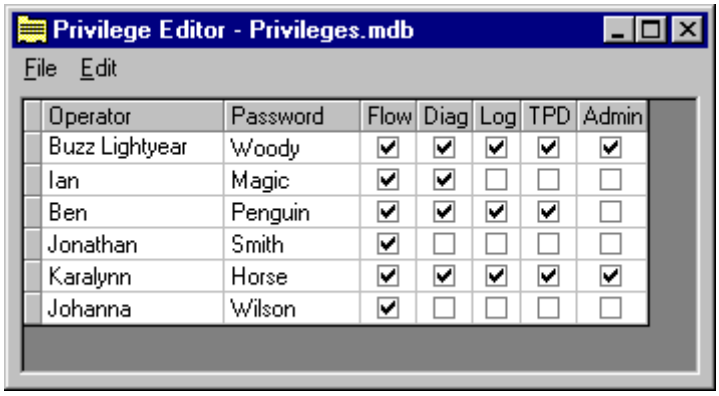
Temperature: 25

Created by Operator M. Mouse

1/21/00 9:55:46 AM	**Sequence 1**		
	Test 1.	-	Full Run
1/21/00 9:55:49 AM	Test 1.1.	Pass	Subsystem A32-Q
	Test 1.2.	-	Navigation Satellite Link
1/21/00 9:55:57 AM	Test 1.2.1.	FAIL	Up Link
	Test 1.2.2.	-	Down Link
1/21/00 9:56:12 AM	Test 1.2.2.1.	Pass	Down Link A
1/21/00 9:56:23 AM	Test 1.2.2.2.	Pass	Down Link B
1/21/00 9:56:34 AM	Test 1.3.	Pass	Norden Sight Alignment
1/21/00 9:56:42 AM	Test 1.4.	Pass	Aft Pointing Radar
	SUMMARY	FAIL	

4. Privilege Editor

The Privilege Editor is used to manage Test Executive operator privileges. It allows an administrator to assign access privileges and passwords for Test Executive operators. The simple grid format has a row for each operator and the columns are dedicated to operator name, password and privilege settings. The columns of the grid can be sorted by a left mouse click on the column heading.



A left mouse click selects a cell for editing. Pressing the **Enter** key moves editing down to the next row. An **Enter** in the last row causes a new row to be added. A row is selected by a left mouse click on the leftmost gray cell of the row. Select multiple rows by holding the left mouse button down and dragging it along the left side of the grid. Row contents are manipulated by the Edit Menu. Function key **F1** accesses online help.

> **Note: Operator privileges are enabled, or disabled, in the security section of the Test Executive Configuration File (TestExec32.cfg).**

Privilege information is stored in an encrypted Microsoft Access database. The database is optionally password-protected, when created, based on an entry in the security section of the Test Executive Configuration file.

For documentation purposes, an image of the Privilege Editor contents is easily captured by pressing ALT+PRINT SCREEN. This places the image in the Windows Clipboard where it can be pasted into a word processor or graphic program.

4.1 Privilege Levels

Test Executive operator privileges are mutually exclusive. Any combination of privileges may be assigned to an operator. All operators have the default privilege to load a TPD file from the Test Program List and execute a Go/No-Go test. This list describes the capabilities associated with each column in the Privilege Editor.

Column	Description
Flow	Allowed to select an execution mode and failure behavior (SOF).
Diag	Allowed to set or clear diagnostic breakpoints and logging.
Log	Allowed to enable or disable data logging.
TPD	Allowed to load TPD files using a file dialog or the file history.
Admin	Allowed to administer the privileges database with the Privilege Editor.

An operator without **Flow** control is only allowed to run the topmost item in the test list. This is typically a test group that includes all of the necessary test objects for validating a UUT. If a lower item is selected, when the Run button is pressed, the topmost item is selected and the operator is notified that he may only run the top item.

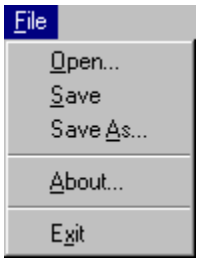
An operator without **Log** control is never prompted to delete log files. He is prompted if the Test Executive is configured to notify without removal. See the section on *Log File Management* for more information.

4.2 Menus

4.2.1 File Menu

The File Menu, for the Privilege Editor, is used to load and save the contents of a privileges database. Typically, the Privilege Editor is started from the Test Executive Tools Menu and it automatically loads the relevant database. Other user scenarios include network management of a common privilege database and creating privilege databases for deployment.

Note: You must have administrator privileges to open a privilege database.

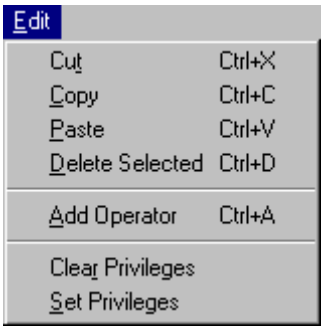


Menu Option	Description
Open	Opens a file dialog for selecting a privileges database. Enter a new file name in order to create an empty privilege database.
Save	Writes the contents of the Privilege Editor grid to the current privilege database.
Save As	Writes the contents of the Privilege Editor grid to a new privilege database.
About	Displays a dialog box reporting the Privilege Editor version and copyright notice.
Exit	Closes the application.

4.2.2 Edit Menu

The Edit Menu, for the Privilege Editor, is used to modify the contents of the privilege grid. The edit commands are similar to those of many Windows applications. Rows are selected by a left mouse click on the leftmost (gray) cell.

> **Note that pressing Enter in the last row of the grid automatically adds an additional row.**

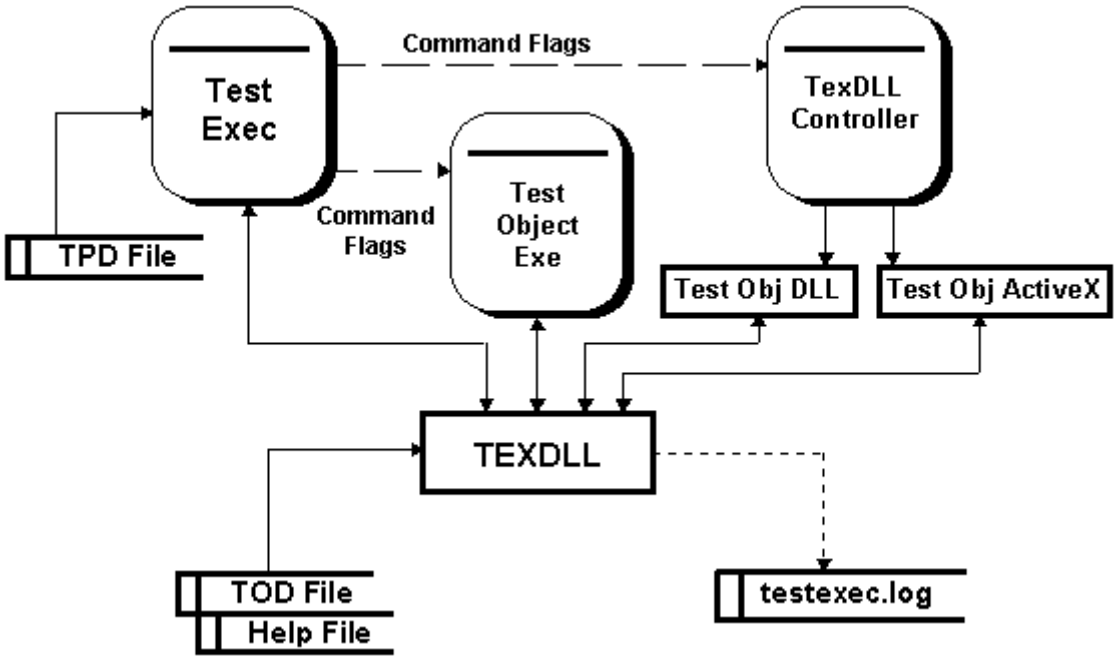


Menu Option	Description
Cut	Copies selected text, or rows, to the Windows Clipboard and removes it from the table.
Copy	Copies selected text, or rows, to the Windows Clipboard.
Paste	Copies the contents of the Windows Clipboard to the cursor position, selected text, or selected rows.
Delete Selected	Deletes the selected rows in the table.
Add Operator	Adds an additional row to the bottom of the table.
Clear Privileges	Resets (unchecks) all privileges for the selected rows.
Set Privileges	Sets (checks) all privileges for the selected rows.

5. Developer Application Notes

5.1 Test Executive Architecture

The Test Executive manages test object execution based on the contents of a Test Program Definition (TPD) file and various user interface selections (e.g. run modes and breakpoints). The TPD file defines a test hierarchy, behavior options and the operational parameters for test objects and setup objects. In general, setup objects have many of the same behavior and interface options as test objects. See the following *Test Program Design* section for an overview on creating a TPD file.



Test and setup objects use a programming interface for synchronizing and communicating with the Test Executive. The functions available to a test object are contained in the TEXDLL (TEXDLL32.DLL). The Test Executive uses the TEXDLL to monitor a test object's activity. If the Test Executive is not active, a log file (TESTEXEC.LOG) is created to track test object operation. This log file can be used during debugging to diagnose test object behavior.

There are three types of test objects that can be run by the Test Executive. An executable test object is run as an independent process. DLL and ActiveX test objects are run as threads in the TexDLL Controller (TexDLLControl.exe) process. By operating in a common process, DLL and ActiveX test objects can share resources. See the following *Test Object Design* section for more information on test object types.

When the Test Executive runs a test object it passes along command flags to direct its behavior. These flags are derived from user selections (e.g. breakpoints), operational circumstances (e.g. repeat count) and TPD defined entries. Some of these flags control the TEXDLL via an input parameter to texStart(), the other flags are intended for use by the test object.

5.2 Test Program Design

When designing a test for the Test Executive it is important to take into consideration all of the capabilities of the system before beginning. Many of these capabilities are closely linked to the Test Program Definition (TPD) file.

Systems that 'evolve' tend to take longer to build than those that are designed. With this in mind, it is a good idea to design the test hierarchy for the Test Executive before creating the individual test objects. This focuses the design on what the test operator will be using to test/verify the Unit Under Test (UUT). This view then structures the division of test requirements. For example, if presenting more than 30 different tests is too confusing, you know immediately not to write 42 test objects.

The decisions to make during this phase include: what tests are to be grouped together; what tests should be run individually; what are the relationships/requirements between tests; what subsystems should be tested as a sequence; how many verifies should be included in each test.

All of these considerations can be easily prototyped with a TPD file and a single 'stub' test object. The sample test object shipped with the Test Executive (TXDEMO32) easily serves this purpose. Command line arguments are used to elicit various behaviors (e.g. pass, fail, error, etc.) from it. The source is included for customization to various requirements. For more information, see *Appendix G - Sample Test Object*.

Once the test is configured in a TPD file, it becomes easier to detect common behaviors that might be delegated to a custom DLL or reusable code modules. It also begins to suggest the necessary behavior for pre and post setup objects. For example, if several test objects need to share data, a DLL can be kept loaded during a test sequence as a data repository.

The relationships between the test objects also dictates the prerequisites that will be required in the final system. It is important at this point to determine if the prerequisites are consistent and necessary (i.e. it is possible to create a tangle of prerequisites that allow no tests to be run). If the prerequisites are too restrictive it might be appropriate to redesign the test sequences or to design the test objects to operate more independently. This is also when conditional branching can be considered for the test list. Again, it is possible to hopelessly complicate testing with convoluted branching. Branching should be avoided when possible and thoroughly tested, with the sample test object (and "**/ucf**"), when implemented.

When the structure of the test has taken shape, it is then possible to consider the design of the test program's help mechanism. For relatively simple test programs, a physical document may be all that is required. Where appropriate, the physical document can be augmented, or replaced, by online help files. The Test Executive supports context sensitive help from the test hierarchy and verify breakpoints. These links to the help files then form the basis for documenting the test program. Consequently, the help system could be created concurrent to the test development and additionally serve as design documentation.

Help files are created using Microsoft Word and are compiled with a help compiler that comes with many of Microsoft's development environments (e.g. Visual Basic, Visual C++). The source for a simple Help project is included with the Test Executive. There are also many third party tools available for developing Windows Help files. Investing in one or more help development tools is strongly suggested. The savings in time and frustration will quickly pay for them.

5.3 Test Object Design

Once a test program structure has been established, the test object requirements are starting to form. These include behavior, shared resources, relation to setup objects and possible prerequisites. All of these are factors in deciding how the test objects should be built. A test object can be a standalone executable (*.exe), a dynamic link library (*.dll) or an ActiveX object (*.dll). Each of these has certain advantages and disadvantages during development and runtime. These should be carefully weighed before deciding on an implementation plan.

Test Object	Programming Language	Pro	Con
Standalone Executable	Any that can call a DLL and run as an executable.	Easy to debug Choice of languages Simplest Abort handling	No resource sharing Slower execution
Dynamic Link Library	Typically C or C++	Resource sharing Faster execution	Harder to develop/debug More complex Abort handling
ActiveX Object	Visual Basic	Resource sharing Easy to develop and debug	Harder to share variables between test objects Most restrictive Abort handling

Standalone executable test objects are typically the easiest to create. A large variety of programming languages and development environments are suitable for building this type of test object. Debugging is handled completely within the development environment and the resultant executable can be run directly by the operating system. Because these test objects operate in their own process, they cannot share certain resources (e.g. instrument handles) and they often run slower due to process creation and DLL loading.

There are two significant advantages to developing test object DLLs. One is that a DLL can share resources such as instrument handles with other test and setup objects. This ensures a continuity of instrument control and avoids the execution time and development effort required to initialize instruments within an individual test object. The other advantage is that test object DLLs, and their support DLLs (e.g. instrument drivers), can be preloaded as setup objects. This permits almost instantaneous execution by the Test Executive with very little system overhead. The downside of resource sharing and preloading is that creating and debugging a test object DLL is more difficult. Shared resources need to be allocated and released at appropriate times and their state is often dependent on test sequencing (e.g. is the data array initialized? is UUT power on?). Abort response for a test object DLL is also more complicated because resources often must be released and/or reset before a test object exits. More information on test object DLLs can be found at *DLL Test & Setup Objects*.

By supporting ActiveX objects, the Test Executive allows a test object to be coded in Visual Basic while also benefiting from the resource sharing that is available to DLLs. Visual Basic is a powerful high-level language with a rich development environment. Debugging is quick (i.e. no compile/link steps) and the language itself prevents many of the allocation and pointer errors that so often plague C programs. Unfortunately the 'protected' nature of Visual Basic makes it more difficult to share variables between test objects and ActiveX abort handling has some very strict restrictions. For more information see *ActiveX DLL Test Objects*.

There is no need to make all the test objects the same type. In fact, there may be certain advantages to using different types together in one TPD. For example, Go/NoGo testing might be implemented with DLLs in order to capitalize on their speed. Infrequently run diagnostics, instrument calibrations or operator interfaces could be suitably built as standalone executables. Frequent benchmarking is highly recommended during development to ensure that the execution speed of the final system is sufficient for the required throughput.

> **Note: changing a standalone executable test object to a DLL is very time consuming, especially if the original language or development system doesn't support creating DLLs. Therefore, be very careful when choosing to build standalone executable test objects and check often to ensure their execution speed is sufficient to the task.**

5.4 Test Object Implementation

Now it is time to focus on the individual test object functionality. Of primary consideration at this point is how the test objects will implement verifies. The two choices are direct verifies (i.e. **texVerify**) that include all parameters and indirect verifies (i.e. **texVerifyTOD**) that operate off of separate Test Object Definition (TOD) files. The direct verifies don't require an additional file and they are easier to debug in an interactive development environment. The indirect verifies have the outstanding feature that they can be configured for different test requirements while using the same test object executable. This is useful when test parameters change or when different test circumstances (e.g. environmental vs. acceptance) need to be supported.

Once the test object functionality has been determined, it is time to allocate that functionality to executable modules. Probably the simplest way to do this is to have one executable per test object. Unfortunately this can result in a large number of files to maintain and track. Another approach is to combine several test objects into one executable. The individual test object functionality can still be exercised by calling different functions in a single test object DLL. Standalone executables can use command line arguments to control which test function to run. The added advantage to this approach is the ability to share common code among several test objects.

Similarly, if TOD files are to be used, they should be considered from a quantity and maintenance perspective. One TOD file can be used to hold the verify parameters for multiple test objects. Generally the fewer TOD files the better, but too many verifies in a single TOD file can create a different maintenance problem. Fortunately the link between a test object and a TOD file is made in the TPD file, so the usage can be easily changed for improved maintenance or data independence.

By defining all of these elements in advance, it is easier to create consistent naming conventions for files and test/verify ID's. Consistent and unique ID's are particularly important for logging test results and linking to test program help files.

Note: Verify commands can also be used to display, and log, additional information about test conditions. For example, if a string compare fails, a follow-up verify could include the failing string. Or, a series of verifies could be used to log instrument serial numbers and calibration dates (if available).

5.5 Test Object Operation

With all of the preceding design complete, it is then time to start coding the test objects. During this process consideration must be giving to error handling, user interface requirements, debugging and abort handling. The Test Executive DLL (TEXDLL) provides a variety of functions that help to address these factors.

For errors and warnings there are the **texErrorBox** and **texWarningBox** functions. These behave similarly, the difference between them is that the warning box is only displayed if warning breakpoints have been enabled (from the command line or Test Executive). Both functions automatically log information to the Test Executive. It is important to remember that destructive error conditions should be corrected in the program, if possible, before alerting the operator with these functions. Otherwise the program is suspended while waiting for an operator response (as the test system melts).

Simple user interface requirements can be accomplished with **texMessageBox**, **texYesNoBox**, **texInputNumber** and **texInputString**. Information can also be sent to the Test Result window with **texPrint**. Sparse use of these functions is recommended to minimize operator irritability.

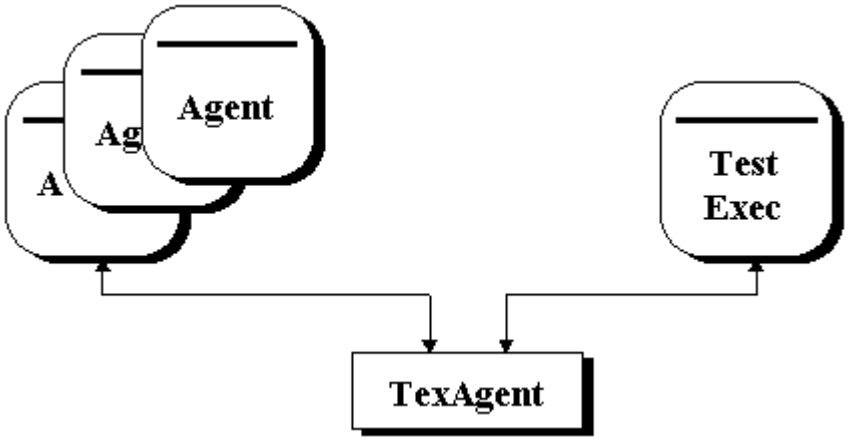
Debugging is greatly enhanced by the thoughtful use of embedded breakpoints (i.e. **texEmbedBreakpoint**). The two types of embedded breakpoints supported are TRACE and DATA. Typically, TRACE breakpoints are used to track the execution flow through various functions and modules in a program. DATA breakpoints are used to view intermediate values of measurements or calculations. This distinction is entirely at the discretion of the individual program developer. Breakpoints are externally controlled through command line arguments. They can be disabled, enabled for logging only and enabled for logging and display. By using them at a test object's critical junctures, or intermediate measurements, it is possible to gain significant insight into the execution process. This can be very helpful during test program integration and field service.

Note: Do not call breakpoint or verify functions during critical times in a test (i.e. when an extended delay might introduce errors or create dangerous conditions). Store measurements and verify them after the critical point has passed. This protects the test system if breakpoints are enabled.

Note: A test object is also responsible for handling abort requests from the Test Executive. When an abort request is received, a test object should halt testing, release resources (e.g. instrument handles) and exit as soon as possible. Consequently, as a test object is being coded, it must track resource allocation and be prepared for an abort request at any time. Each type of test object executable has different requirements for abort handling. These are described in later sections: *Aborting a Test*, *Aborting a Standalone Test Object*, *DLL Abort* and *Aborting an ActiveX DLL*.

5.6 Agent Interface

In many circumstances it is useful for the Test Executive to respond to external control or queries from other processes. This includes applications that are conducting diagnostics (e.g. AI Test) or coordinating lengthy burn-in testing (e.g. an environmental chamber controller). To provide this capability, an "agent" interface has been designed to provide a client-server interface to the Test Executive. This allows multiple applications (i.e. clients) to access information from, and control over, the Test Executive.



The interface is provided by a single TexAgent DLL (TAgent32.DLL) and a set of eight functions. This DLL coordinates the flow of requests and commands between applications. The behavior of the interface is similar to controlling an instrument via SCPI commands. For example, a request for status would be issued by:

agSendMessage(agId, "TestExec:ETMan:Get:Status")

The second field in the command string, *ETMan*, identifies the source of the request. A reply from the Test Executive would look like:

"ETMan:TestExec:Reply:Status:Idle,Ready,Unlocked,c:\tpd\texdemo.tpd"

Every message sent through this interface requires a response from the server. The response may include requested information, operational results or simply an acknowledgment. This creates a closed-loop operation that has a direct cause and effect behavior and ensures uncomplicated message buffering. Consequently, each application using this scheme must be prepared to wait when a server is busy.

Note: This interface is not intended for use by a test object. Whenever a test object is active, the Test Executive returns a busy response to all messages except for some queries (e.g. "Get:Status").

Sample code and additional documentation is included with the Test Executive in a TexAgent subdirectory. This subdirectory only appears when a **custom** installation is performed.

5.7 Command Line Arguments

When the Test Executive is started, it supports several command line arguments. These are used to control various aspects of its startup behavior. This allows additional control by external applications, or adaptation to specific system configurations. By supporting a TPD file name on its command line, the Test Executive can be specified as the program to use when opening files with that extension (*.tpd).

Command	Description
/etm	Suppress display of the splash screen and disable the loading of a TPD file from the file history list.
/vc	If a system component fails version checking, allow the user to continue or cancel operation.
/tpl	Automatically activate the Test Program List when the Test Executive starts. Use with **/etm** to keep the previous TPD file from loading. The Test Program List is the best way for an inexperienced operator to select a TPD.
/sn *SerialNum*	Define a UUT S/N that is automatically set every time a TPD file is loaded. This is intended for use during test development only.
/op *OperId*	Define an Operator ID that is automatically set when the Test Executive starts. This is intended for use during test development only.
TpdFileName	Load the specified TPD file instead of one from the file history list. The file name **must** include a full directory path and drive letter.

The "**/sn**" and "**/op**" command line arguments should only be used during test development to eliminate the repetitious entering of this information when the Test Executive starts, or a TPD file is loaded. This also promotes the consistent use of a specific UUT S/N during development. This thus creates a database containing a test result history that reflects the various phases of test development and debug.

Note that defining a UUT S/N in this way is equivalent to manually entering one; therefore any **PreMSN** setup objects will be executed immediately following a TPD load. Also, these predefined values should match any formatting requirements specified for the Operator ID and UUT S/N.

5.8 Component Validation

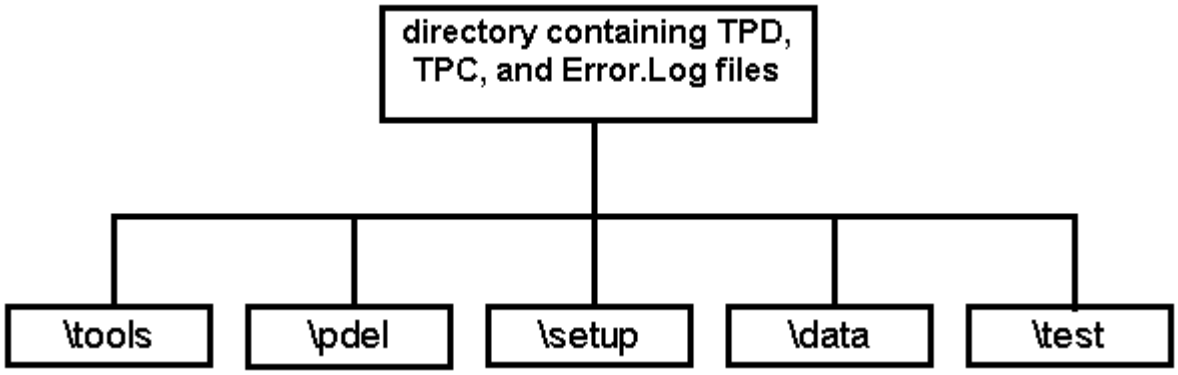
Computers and test systems are faced with ever increasing complexity. This results in a delicate balance of operating system, instrument drivers, test development tools and test-specific resources. These all need to be present, and compatible, in order for successful testing to occur. A single out-of-place file, or incompatible version, can cause days of frustration and lost productivity. The Test Executive has extensive support for tracking and validating the numerous files required for a modern test set.

When a TPD file is loaded by the Test Executive, the files referenced in it are validated. This is accomplished by first verifying that the files exist, followed by confirming that they are the correct version. The following sections describe the required TPD directory structure and the processes for existence and version checking.

The results of the file version checking are logged to the database, if one is active. These results include the expected and actual file dates and versions. This provides trace ability for possible changes to the test environment. For example, UUT failure trends could be tracked to a version change for a test object or instrument driver. This information is displayed in the Report Generator as a sequence with a single test entry (**Version Check**).

5.8.1 TPD Directory Structure

The following diagram shows the subdirectory structure required to support a TPD file. The "**\data**" and "**\pdel**" directories are automatically created if they don't exist. Multiple TPD files can use the same directory space. This is useful if there are numerous common files shared by several test programs.



The TPD parent directory contains the TPD, TPC and optional test program help file. This is also where a log file (error.log) is created if errors are encountered while loading the TPD. The "**\test**" subdirectory holds all of the test object files including executables, test object help and TOD files. The "**\tools**" and "**\setup**" subdirectories are for action and setup objects, respectively.

The "**\pdel**" subdirectory contains PDEL files (*.pdl) from Test Executive operations with the associated TPD. One file is created for each execution by the Test Executive. By default the Test Executive notifies the operator if more than 1000 PDEL files are in the directory.

Database files are found in the "**\data**" subdirectory. This subdirectory contains Microsoft Access database files created by the Test Executive. These databases, one per UUT S/N, contain UUT test results and are used by the Report Generator. By default, the test operator is notified if there are more than 50 files in this directory and is given the option to delete them.

These file management operations can be further controlled by settings in the Test Executive Configuration file. See the section on *Log File Management*.

5.8.2 File Existence Checking

The presence of files, referenced in a TPD, are confirmed when the TPD is loaded by the Test Executive. Certain types of files are only allowed in specific directories. This includes TOD and help files. Others, such as setup objects and action objects, can be elsewhere in the system.

- The test program help file, defined in the [Identification] section of the TPD, must be in the TPD directory.
- TOD, test object executables and test object help files must be in a "**\test**" subdirectory relative to the TPD directory.
- Action Objects (i.e. tools) are checked for in their relative subdirectory ("**\tools**") first, followed by an execution path search. If a full path is designated, the file must exist at that location.
- Setup Objects without a path are checked for in their relative subdirectory ("**\setup**") first. Then an execution path search is conducted for DLLs only. Non-DLL Setup Objects must be in the setup subdirectory or have a specified directory path. If a full path is designated, the file must exist at that location.
- Files listed in the [OtherFiles] section are checked based on their specified type and path. An executable file type (ExeFile) can optionally be located anywhere in the system's execution path. The table below defines the different possible path attributes and their corresponding search patterns.

File references in the [SetupObjects], [ActionObjects] and [OtherFiles] sections of the TPD can be defined with several different path attributes. The path attribute determines the type of checking that occurs when verifying the file's existence and version information.

Type	**Designator**	**Description**
Full Path	[Drive]: \Path\FileName.ext	Specifies a full path from the root directory. If no drive letter is included the default drive is assumed.
Relative Path	\Path\FileName.ext \FileName.ext	The file path is relative to the TPD directory.
Search Path	FileName.ext	The executable file (e.g. EXE or DLL) is somewhere in the execution path of the system (not valid for Setup executables).

5.8.3 Version Checking

Once the files referenced by the TPD are located, their versions are verified. All version checking can be enabled or disabled in the [Options] section of the TPD file. The results from version checking are logged to each UUT's database. Up to three attributes of a file can be validated with this process. These attributes are part number, version and last modification time/date.

The part number and version of a file are accessed differently depending on the type of file it is. For text files, this information is expected to be in a Windows INI format. This is the same format that is used for TPD and TOD files:

```
[Identification]
PartNumber = 12345678-1234
Revision = B
```

For executable files, the part number and version are read from an embedded version resource. The part number is read from the *Internal Name* field and the version is read from the *File Version* field.

Note: There are two *File Version* entries in a version resource; one is numeric and the other is a string. Version checking is always performed with the numeric entry.

Version information can be added to any Microsoft Windows file that can have Windows resources, such as a dynamic-link library (DLL), an executable file, or a font file. The version information can be manually examined by looking at a file's properties with a File Manager or the Windows Explorer.

If a part number is specified, it must match the one in the file or a TPD load error is generated. This behavior can only be disabled from the [Options] section. The version and time/date comparisons are controlled on an individual file basis by a *CheckType* field. This field specifies the type of comparison to perform (e.g. CHECK_EXACT, CHECK_ORLATER, CHECK_NONE). For more information on the format of version checking fields, see *Other Files Syntax*.

Part number and version validation is performed by case-insensitive string compares. This means that part number ABC equals aBc and revision E is newer than revision d.

Version checking is not performed on any of the help files specified in the TPD. If version checking for them is desired, they can be individually listed in the [OtherFiles] section.

Components required by the Test Executive (e.g. OCX's, DLL's, etc.) are validated when it starts. The operator is notified if they are not located or if they are not the correct version. Newer versions of components are accepted.

5.9 Operator Privileges

Test operators are assigned a set of privileges that control what capabilities of the Test Executive they may use. Privilege settings are stored in a database and are managed by an administrator with the Privilege Editor.

The initial operator is automatically assigned via the Window's API function **GetUserName**.

5.10 Power Monitoring

Some electronic assemblies have a limited amount of time that they can be powered up without damage. The Test Executive supports a monitoring mechanism that tracks when power is applied to a UUT. If the defined time limit is reached, current testing is aborted and no further testing is allowed until the UUT cools down. The power state and time remaining are shown on the Power Monitor display.

5.11 Independent Breakpoint Behavior

Starting in version 4.0 of the Test Executive (version 2.2 of the TEXDLL), embedded breakpoint, message, warning, and error dialog boxes are available outside the context of a test. This means that the functions: texEmbedBreakpoint(), texMessageBox(), texWarningBox(), and texErrorBox() can be used without calling texStart() or texFinish().

Programs that do not call texStart() rely on the Test Executive to control the enable/disable modes for warnings and embedded breakpoints. When such a program is run stand-alone, warnings and embedded breakpoints are disabled by default.

5.12 Prerequisite Tests

Each item in the Test Executive test list can have pretest requirements (i.e. prerequisites) associated with it. These requirements must be fulfilled before the test is executed. For example, test object B might depend on test object A first performing a safe-to-power-on test.

Prerequisites are defined for test groups and test objects in the Test Objects section of the TPD file.

5.13 Conditional Branching

Execution flow, within the Test Executive test list, is optionally controlled by conditional branching statements defined in the Test Objects section of the TPD file. These statements allow test objects to be repeated or skipped. The conditional operation (i.e. Pass or Fail) is based on the execution results of the preceding test object. A variety of conditional operations are defined with a test branch statement.

Conditional branching creates the possibility that a test object may be executed more than one time during a sequence. The test object may need to adjust data logging based on the number of times it is executed. The Test Executive passes execution counts to the test object via a command flag (e.g. **/rpt** 1:2:1).

5.14 Splash Screen

When the Test Executive is started, a splash screen is displayed with version and copyright information. This splash screen is optional and can be disabled by deleting or renaming the program (SPLASH32.EXE) in the Test Executive directory. For site specific requirements, the splash screen application can be replaced by a custom program displaying alternate information. A custom splash screen should have an automatic time-out of about 6 seconds and return focus to the Test Executive when the application ends.

6. Test Executive Files

This chapter describes the content, formatting and syntax of the files associated with the Test Executive. Some of the files are automatically created during the course of operation. Other files are externally created and define the behavior of the Test Executive.

The developer-created Test Program Definition (TPD) file controls much of Test Executive's appearance and behavior. It defines the test hierarchy, data requirements and operator interaction. A companion Test Program Configuration (TPC) file is generated automatically to track the most recent configuration data.

When the Test Executive loads a TPD file, extensive syntax checking is performed on its contents. Any errors or warnings are written to an error log file. During test execution, errors, warnings and logged information are written to the Debug Log window and debug log file.

The Test Executive uses a private initialization file to track information from one session to another. This information includes a file history list and the most recent settings for data logging and breakpoints.

If a test object is executed standalone, all of its data is written to the TestExec log file instead of being sent to the Test Executive. This gives a test developer insight into the internal behavior of the test object, which can be invaluable during debugging.

A Test Object Definition (TOD) file is used to separate test operation from measurement limits. This allows measurement limits to be modified without recompiling test objects.

The developer-created Test Executive Configuration file defines the test program list and operator privileges database. It also includes identification information for the current test station.

Parametric test data, and test configuration information, are optionally logged to a Microsoft Access database and/or Parametric Data Log (PDL) files. The database is used by the Report Generator to display and print reports on a UUT's test results. PDL files are typically gathered together and consumed by a data repository system for later analysis.

6.1 Error Log File

When a TPD file is loaded, the Test Executive creates ERROR.LOG to hold error and warning messages generated from the TPD. This file is placed in the directory containing the TPD file. It is overwritten when a new TPD file is loaded. This log file is particularly useful when correcting numerous syntax and file version errors during TPD development.

6.2 Debug Log File

When a TPD file is loaded, the Test Executive creates DEBUG.LOG to hold error and warning messages generated during test execution. This file is placed in the directory containing the Test Executive. It is overwritten when a new MSN is entered. This file receives all of the information that is displayed in the Debug Log window. It is useful for recording and viewing extensive debug information that might exceed the buffer of the Debug Log window.

6.3 TestExec Initialization File

The Test Executive uses an initialization file (TEXEC32.INI) to hold information about its window positions, file history and recent control options. This file is used, when the Test Executive starts, to resume operation where it was previously. This file can be deleted if the Test Executive encounters problems while starting. It is automatically rebuilt when the Test Executive runs.

This initialization file is **not** intended for viewing or manipulation by external processes. Its internal format is subject to change without notification.

6.4 TestExec Log File

When a test object is executed by itself, the TEXDLL generates a log file of all the function calls that it receives. This file (TESTEXEC.LOG) is created in the test object's home directory. The log file is useful during test development for examining verify results, warnings and breakpoints. Note that the logging behavior is controlled by command line arguments to the test object.

There is not necessarily a one-to-one mapping between the entries that appear in this log file and the lines that get written to the Test Results window when a test object is run by the Test Executive. The following is a list of entry formats and the TEXDLL interface functions that may produce them in the TestExec log file.

Entry Format	**Source**
TESTINIT:	texStart()
TOD: *file_name; part#; rev; format; err_count*	texStart()
TESTCLOSE: *test_status*	texFinish()
PRINT: *message_string*	texPrint()
PDEL: *pdel_message_string*	texLogPdel()

LOG entries are created for a breakpoint or a warning when the corresponding TEXDLL behavior has been set to "log-only". This mode allows breakpoints and warnings to be reported without interrupting the execution of a test object.

Entry Format	**Source**
LOG: Verify; *message_string*	texVerify(), texVerifyTOD()
LOG: Data; *message_string*	texEmbedBreakpoint()
LOG: Trace; *message_string*	texEmbedBreakpoint()
LOG: Warning; *message_string*	texWarningBox()
LOG: DLL_Warning; *message_string*	Internal TEXDLL warning.

BREAK entries are created any time a dialog box is opened and test object execution is suspended while waiting for user input. For breakpoints and warnings, this occurs when the corresponding TEXDLL behavior is fully enabled.

Entry Format	**Source**
BREAK: Verify; *message_string*	texVerify(), texVerifyTOD()
BREAK: Data; *message_string*	texEmbedBreakpoint()
BREAK: Trace; *message_string*	texEmbedBreakpoint()
BREAK: Warning; *message_string*	texWarningBox()
BREAK: Error; *message_string*	texErrorBox()
BREAK: MessageBox; *message_string*	texMessageBox()
BREAK: YesNoBox; *result*; *message_string*	texYesNoBox()
BREAK: DLL_Warning; *message_string*	Internal TEXDLL warning.
BREAK: DLL_Error; *message_string*	Internal TEXDLL error.

Each call to texVerify() or texVerifyTOD() produces one log entry of twelve fields according to the following format:

VERIFY: *ver#*; *TODver#*; *descr*; *id*; *type*; *limit1*; *limit2*; *prefix*; *unit*; *result*; *LB*; *UB*

LB and *UB* are the upper and lower bounds calculated for a verify of type RANGE, NOM_TOL, or NOM_PER

Note that a single log entry may require one or more lines. Generally, a log entry only requires multiple lines if it includes a *message_string* with multiple lines.

6.5 Test Program Definition File

The Test Program Definition (TPD) file contains the information required to load a set of tests for execution by the Test Executive. It is the single most important part of the Test Executive. TPD files are prepared in-house as part of the test object development process.

The TPD file is a flexible, standardized mechanism for configuring runtime requirements for test objects. Setup, testing, operator tools and configuration information is built into the TPD file and is used repeatedly for like units. The information contained within the TPD file can be minimal or extensive. A TPD file must include an Identification section and one or more test objects. All other sections are optional.

A TPD file contains up to nine sections of line-oriented information. See the following sections for more information on these parts of a TPD file.

Section	Description
Identification	Identifies test program by name, part number and revision. Also specifies the format version of the TPD file and an optional test program help file.
Options	Contains flags for enabling and disabling optional Test Executive behavior.
Setup Objects	Defines executable files to be run before or after a test. Setup objects ensure that the UUT is in a testable state.
Test Objects	Defines the items displayed in the test list hierarchy and the executable tests they are associated with.
Action Objects	Defines executable applications that are to be made available to the test operator. Action Objects are displayed for selection in the Tools Menu.
Power Monitor	Defines parameters for UUT power-on tracking.
Other Files	This section contains a list of files required by other elements of the test program.
Configuration	A set of parameters defining the data to enter or associate with a test.
Configuration Labels	Defines optional alternate labels for some configuration data fields.
User Defined	Optional sections for defining test-specific data.

A TPD file is specified on the command line or selected using the Test Executive's File Menu. When starting, the Test Executive automatically reloads the latest TPD file from its file history. If a problem is detected from the previous TPD file (not closed correctly due to a system crash, power failure, etc.), the user is prompted whether to restore interrupted test status from the database. Otherwise, when a TPD file is opened, the name of the TPD file is displayed on the title bar of the user interface and the tests defined therein are displayed in a hierarchical list for selection and execution by the test operator.

The Test Executive reads the TPD file and extracts needed information while performing error checking and file verification. For more information, see *Component Validation*. All detected errors are logged to the ERROR.LOG file and the Debug Log display accessible through the Window Menu. The operator is notified by a message box when certain errors, deemed "critical", prevent the TPD file from being loaded.

6.5.1 Test Program Definition Format

Each section of a TPD file begins with a header line that includes a section name enclosed in square brackets (e.g. [Identification]). The body of each section follows the header and consists of one or more data lines. One data item is defined per line, according to the following format:

keyword = *Field1*; *Field2*; ... ; *FieldN*

Where *keyword* is a legal keyword for the current section. A single data field or a semicolon delimited list of data fields is assigned to each keyword using an equal sign. Within a section, legal data items can be listed in any order. A TPD file can include blank lines and comments. Comments are preceded by a pair of forward slashes (//), and all text on a line following a pair of forward slashes is ignored.

Note: Tabs are considered illegal characters. The existence of a tab within the TPD file causes an error.

Some operating systems (e.g. Win95) allow semicolons and commas as part of file and directory names. To avoid parsing errors, files referenced in the TPD with embedded semicolons or commas should be enclosed in double quotes (e.g. “test1;a.tod”).

Several complete TPD files are shipped with the Test Executive as examples. These are a good place to start when learning about TPD design, behavior and syntax. They can be found in a subdirectory (**\tpd**) to the Test Executive’s home directory.

Note: TPD files should be stored separate from the Test Executive’s home directory. This makes them more portable and protects them from Test Executive upgrades or changes.

The following is a brief explanation of the contents of the sample TPD files:

File Name	Description
TXDEMO32.TPD	A minimal TPD file. Useful for exercising the basic functionality of the Test Executive.
TXTEST32.TPD	Extensive TPD file with more options, tests and setup objects. Exercises many aspects of the Test Executive including error handling, user interface options and prerequisites.
TXOPTN32.TPD	TPD file with a variety of Test Executive options, input data formats and a test program help file.
TXBRANCH.TPD	A TPD file with a selection of example conditional branching statements. An operator exercises branching by controlling test object status (i.e. pass or fail).

6.5.2 Identification

Each TPD file must have an Identification section. This section provides the Test Executive with critical information necessary to ensure compatibility between the unit under test and the test(s) to be performed. The TPD part number, revision, name and format version are included in this section. This is also where a Windows Help file is optionally defined for the test program. If a help file is specified, the Test Executive enables the "Test Program" option on its Help Menu. The following section describes the Identification section syntax.

6.5.2.1 Identification Syntax

This section identifies the TPD program name, part number and revision. It also indicates the TPD format version and optionally specifies a test program help file. If specified, the test program help file must reside in the same directory as the TPD file. Test program help files can also be defined in the Test Objects section of the TPD file.

Due to changes to the TPD file syntax for drop-down lists in the Test Configuration dialog, the format version has been advanced from 2.0 to 2.1. This new format value prevents earlier versions of the Test Executive from loading the TPD file. If drop-down lists are not specified in a TPD file, its format version can be left as 2.0 for greatest compatibility with existing systems.

Keyword	Description
PartNumber	Required part number for identifying TPD.
Revision	Required revision number of TPD.
ProgramName	Required descriptive name for the TPD.
FormatVersion	Required TPD format level (**2.1**).
ProgramHelp	Optional Windows Help file containing TPD information and links for related test objects and verifies.

All lines within the [**Identification**] section are formatted as:

keyword = *string*

Example

```
[Identification]
PartNumber = 123456-789
Revision = A
ProgramName = Test Program 59
FormatVersion = 2.1
ProgramHelp = helpfile.exe
```

6.5.3 Options

The Options section of a TPD file is used to define Test Executive operation and test behavior. Often when new capabilities are added to the Test Executive they are enabled, disabled or controlled from this section of the TPD file. This section controls the following behavior options:

- Enable or disable database and PDEL logging
- Enable or disable version checking
- Specify UUT MSN and Operator ID formatting requirements
- Define specific custom test types
- Enable or disable multiple test selection
- Specify application for custom report printing
- Specify test list expansion level when a TPD is loaded
- Define setup DLL unloading to be normal or reverse order
- Enable or disable automatic Test Configuration prompt
- Breakpoint settings when a TPD is loaded
- Define custom headers and footers for datasheets
- Control automatic test status reset behavior
- Control PDL logging format (PDEL or IEEE-1545)
- Control whether to retain logged data following an abort or halt
- Control display of TPD revision on Test List window

For more information on the formatting and syntax of these, see *Options Syntax*.

6.5.3.1 Options Syntax

This section defines behavior options for the Test Executive. These are used to customize or restrict the operation of the Test Executive for the corresponding TPD file.

Keyword	Description
VersionCheck	Version check referenced files (Y/N), default is Y.
Database	Controls database logging, default is User.
PDEL	Controls PDEL logging, default is User.
AutoStatusReset	Controls test status reset, default is Never.
UUTMSNFormat	Optional format mask, and minimum size, for UUT MSN.
OperatorIDFormat	Optional format mask, and minimum size, for Operator ID.
TestType	Optional list of up to 20 test types, delimited by a semicolon (;).
MultiSelect	Allow multiple tests to be selected (Y/N), default is N.
ReportPrinter	Specify custom application for printing a datasheet.
ExpandTestList	Set the number of levels to initially expand the test list (0-6), default is 6.
UnloadDLLs	Unload order for setup DLLs (Normal/Reverse), default is Normal.
Breakpoints	Define breakpoint settings when the TPD is loaded.
ReportHeader	Custom datasheet header.
ReportFooter	Custom datasheet footer.
PdelLimitType	Enables the logging of measurement limits to a PDEL file. The specified string is used as the <limit type> for the DEFINE_LIMIT statement in the PDEL file. For example: PdelLimitType = SPECIFICATION
DataLogType	Overrides system PDL log type (PDEL or IEEE-1545) for this TPD.
LogOnAbort	Retain logged data following an abort or halt (Y/N), default is N.
DisplayTpdRevision	Display TPD revision on Test List window (Y/N), default is N.
TestConfigurationPrompt	Prompt with TPC for every test sequence (Y/N), default is N.

All lines within the [**Options**] section are formatted as:

keyword = *string*

The **UUTMSNFormat** and **OperatorIDFormat** data strings consist of two fields delimited by a semicolon. The first field is the required format (see *Appendix E - Formatting Mask*) and the optional second field has the minimum number of characters for the associated keyword.

By default, database and PDL logged data is discarded when an abort occurs. PDL data is also discarded when a test sequence is automatically or manually interrupted (e.g. by using the Quit button). Use the **LogOnAbort** option to retain logged data following an abort or halt.

When version checking is disabled in this section (**VersionCheck = N**), this also disables the internal part number check that is performed on files. This is the only way to disable part number validation when a TPD is loaded. Disabling the individual version check for a file (i.e. CHECK_NONE), does not disable the file's part number validation.

> **Note: Version checking should always be enabled when a test system is released.**

Example

```
[Options]
VersionCheck = N
PDEL = Always                   // Always log test results in PDEL format files
Database = User; 1              // User controls logging, initial setting is on

UUTMSNFormat  =  ####-####      // Note, see Appendix E - Formatting Mask
OperatorIDFormat = ?###### ; 3

TestType = SAT; ACT; GRE; Grit Analysis
ReportPrinter = custom.exe; /p3; 1234-567;A

// Enable logging for Data and Trace breakpoints, enable a
// breakpoint on a unique verify ID (Vrfy34), enable
// Warning breakpoints and leave SOF unchanged.
Breakpoints = 1; 1; -Vrfy34; 1;

MultiSelect = Y                 // Allow multiple tests to be selected in test list
ExpandTestList = 2              // expand 2 levels of the test hierarchy on load
DisplayTpdRevision = Y          // show TPD revision on Test List window

TestConfigurationPrompt = Y     // display Test Configuration for every test run
UnloadDLLs = Reverse            // unload setup DLLs in reverse order

DataLogType = PDEL              // force logging to use original PDEL syntax

ReportFooter = \n\cSecure Document - Do Not Copy
```

6.5.3.2 User Control Options

Database logging, PDEL logging and automatic status reset options are controlled by the operator from the Test Executive Options Menu. The extent of the control allowed the operator is determined by the Options section of the TPD file. The possible settings for these options are:

Keyword	Description
Always	Operation always occurs. Option is shown checked and disabled (grayed) on menu.
Never	Operation never occurs. Option is disabled on menu.
User	Operation is controlled by operator. Initial condition is based on most recent setting.
User; 0	Operation is controlled by operator. Initial condition is off (unchecked).
User; 1	Operation is controlled by operator. Initial condition is on (checked).

The default for PDEL and database logging is **User**. If the **AutoStatusReset** keyword is not present, it is not included on the Options Menu and it defaults to **Never**.

The **User** setting, with an initial condition, is reverted to when an operator's privileges disallow direct control of the option.

6.5.3.3 Report Printer

A custom report printing application can be defined in the Options section of the TPD file. This is invoked when "Print Summary Datasheet" is selected from the File Menu. When selected, the application is executed with a command line containing the current UUT database name (enclosed in double quotes if its directory path contains spaces) followed by any arguments from the TPD entry. This custom application can be built with a variety of tools including Microsoft Access, Visual Basic or Crystal Reports. The internal structure of the UUT database is defined in Microsoft Access Database File.

The format for defining a custom report printer is:

ReportPrinter = *FileName; Arguments; PartNumber; Version*, *Date*, *Time*, *CheckType*

Field	**Description**
FileName	Name and path of application.
Arguments	Optional command line arguments for application.
PartNumber	Optional part number of file.
Version	Optional version of file (e.g. 2.1.3).
Date	Optional modification date of file.
Time	Optional modification time of file.
CheckType	Kind of version checking: CHECK_EXACT, CHECK_ORLATER (default), CHECK_NONE

The part number and version information is only necessary if version checking is to be performed on the application when the TPD file is loaded.

6.5.3.4 Test Type List

The Test Type list is a drop-down control on the Test Configuration dialog. It is used to identify the type of test being performed on the current UUT. This is quite valuable for categorizing the circumstances associated with logged test data. For example, data collected while developing or debugging a test can be identified as Engineering in order to isolate it from process data associated with manufacturing.

The Test Type drop-down list has a built-in list of four test types (Acceptance, Manufacturing, Calibration, Engineering). Add a custom list to the Options section of the TPD file to change these choices:

TestType = Manufacturing; Environmental; Military

A custom test type list has up to 20 entries. The four default test types are included within a custom list only if they are manually added.

When a TPD file is loaded the contents of the Test Type field are restored from the TPC file. In some situations there is a requirement that the test operator actively enter certain information before testing begins. For those circumstances, the first item in the test type list should be SELECT:

TestType = SELECT; Manufacturing; Environmental; Military

When a SELECT is present, the test operator is required to chose a test type from the list before the Test Executive runs its first test (following a TPD load). Subsequent tests run with whatever the operator has selected. So, a SELECT at the beginning of a test type list prevents the field's contents from being set to a default value, or restored from the TPC file, when the TPD file is loaded. Fields with the SELECT option are initially displayed as blank.

6.5.3.5 Breakpoint Settings

The **Breakpoints** keyword, in the Options section of a TPD file, defines the initial settings for breakpoints and logging. This forces the settings to a known state when the TPD file is initially loaded. Creating test procedure documentation is easier when the behavior of the Test Executive, following a TPD load, is predictable. These are also the settings that are reverted to when an operator's privileges disallow diagnostic operation. The format for the entry is:

Breakpoints = *Data*; *Trace*; *Verify*; *Warning*; *SOF*

Field	Description
Data	0 = None, 1 = Log Only, 2 = Log and Break
Trace	0 = None, 1 = Log Only, 2 = Log and Break
Verify	0 = None, 2 = Failing Verifies, 3 = All Verifies. A negative sign precedes a verify number or ID to break on.
Warning	0 = Off, 1 = On
SOF	0 = Off, 1 = On

Blank entries leave the breakpoint unchanged from its current setting.

6.5.3.6 Custom Datasheet Header/Footer

Sometimes it is necessary to add additional information to the printed reports created by the Test Executive and Report Generator. This information might include security warnings, tracking numbers or test constraints. The Options section of the TPD file allows the definition of a datasheet header (**ReportHeader**) and footer (**ReportFooter**). These entries can each be up to 255 characters long. Embedded formatting commands allow new lines (**\n**), tabs (**\t**), centered text (**\c**) and a date/time stamp (**\@**) for when it was printed. An "above the line" command (**\l**) forces the header to be the topmost printed text. The datasheet header also supports an embedded graphic (**\g**) that is always printed in the upper right corner of the datasheet. The graphic information is read from a file (**ReportHeader.bmp**) that can reside in the TPD directory, its parent directory or the application directory (typically C:\Program Files\SSI Test Executive). The graphic file is searched for in that order, so different graphics can be associated with different TPDs.

Note: Multiple new line commands (\n), separated by spaces, may be required when adding blank lines to a minimal header.

Example

```
ReportHeader = \cTP 456-89Q\n\cTestExec v6.0\n
ReportFooter = \n\cSECURE DOCUMENT\n\cDO NOT COPY
```

6.5.4 Setup Objects

Sometimes a UUT, or an instrument, must be in a specific state prior to, or after, testing. This section of the TPD file allows the test designer to specify executable programs as pre or post events for a unit. For example, a test may be designed so that a setup program turns on power and initializes the UUT prior to executing a test sequence. A post sequence program would then turn off power. Multiple programs can be associated with each of these pre or post conditions. The order that they are entered in this section is the order in which they are executed.

Setup objects can also be DLLs that are loaded for the duration of a test sequence, MSN or TPD. Keeping a DLL loaded facilitates information sharing between test objects and can avoid the reloading of instrument drivers. For help with DLL load problems, see *DLL Dependencies*.

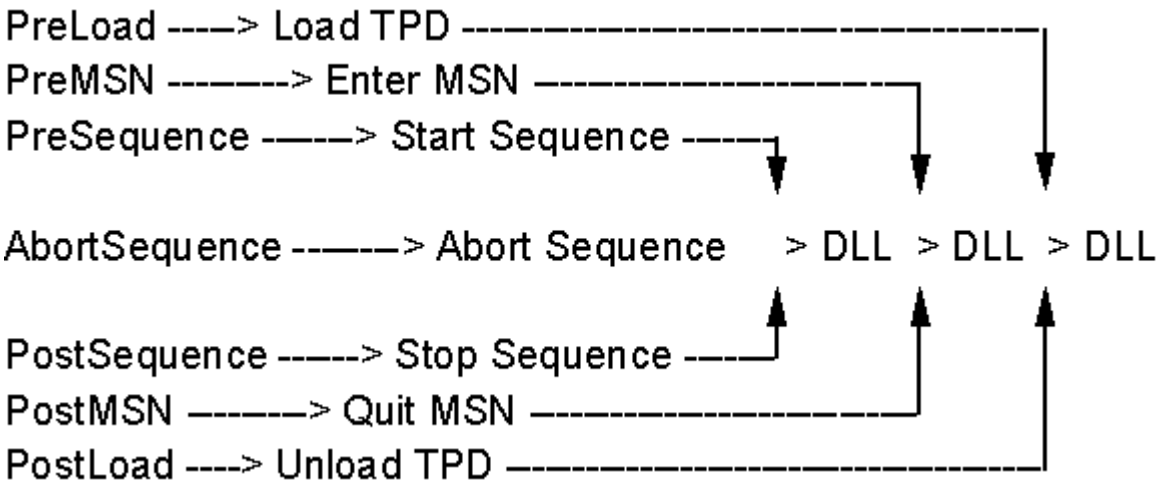
Setup objects have the same behavior requirements as test objects. It is necessary for them to signal their beginning and end with texStart() and texFinish(). They should also respond to abort requests from the Test Executive. Most of the TEXDLL functions are available for use by a setup object. TOD verifies and compares are not possible because there is no TOD file associated with a setup object. Direct verifies are supported, but the results are not logged or displayed by the Test Executive. See the *Test Executive Architecture* and *Test Program Design* sections for overviews on test and setup object operation.

PreSequence and PostSequence setup objects are run for each iteration of a loop (i.e. when the Test Executive is in Loop Mode). For more information on the formatting and syntax of this section, see *Setup Objects Syntax*.

Operational definitions:

PreLoad	executed when TPD file is loaded
PostLoad	executed when TPD file is unloaded
PreMSN	executed when a new UUT MSN is entered
PostMSN	executed before a new UUT MSN is accepted
PreSequence	executed prior to running a selected sequence of tests
PostSequence	executed after running a selected sequence of tests
AbortTest	executed when an abort occurs, from either operator or test object

The following diagram shows the context of setup object execution and DLL loading/unloading.



The Test Executive searches for setup object files in their relative directory first, followed by an execution path search (for DLLs only). If a full path is designated the file must exist at that location. For more information, see *File Existence Checking*.

Normally DLLs are unloaded in the same order that they were loaded (i.e. in the order listed in the TPD file). A TPD option (UnloadDLLs) is provided for reversing the unload order of DLLs. For more information, see the *Options* section.

To support post sequence processing, a command line argument (**/SeqPass**, **/SeqFail**, **/SeqError**) is added for all PostSequence setup objects. This allows the post sequence setup objects to conditionally initiate REQUEST FOR DEVIATION (RFD) and NO-TEST processing. PostSequence setup objects are executed before the operator is notified about reported, or detected, error conditions.

PreSequence setup objects receive a repeat count command line argument ("**/rpt** 0:1:1") which indicates the number of sequence loops that have occurred. For more information on this argument, see the *Test Object Command Line Arguments* section.

Note: Extreme care must be taken when developing a DLL that will serve as an abort object. Since the Test Executive does not allow abort objects to be aborted, it is difficult to deal with an abort object DLL that "hangs" (e.g. gets stuck in an endless loop). DLLs are not visible from the Windows Task Manager, so they are not easily detected or terminated. Terminating the Test Executive's DLL sharing process (texDLLControl) does force the abort object to shutdown.

An AbortTest setup object may want to execute a texFinish(RELOAD_FINISH) at the end of its operation. Reloading the current TPD can help to release resources that may remain after a test object is aborted. For more information, see the *TPD Reload* section.

6.5.4.1 Setup Objects Syntax

This section of the TPD file allows a developer to specify setup objects for pre or post manipulation of a UUT. Each entry specifies a part number, revision, command line argument and executable program to associate with a setup type. Multiple entries can be made for each setup type. Full and partial directory paths are allowed when specifying the location of a setup object. For more information, see *Setup Objects*.

Note: Setup objects must execute TEXDLL functions texStart() and texFinish() in order for the Test Executive to monitor their operation.

Keyword	Description
PreLoad	Executed when TPD file is loaded.
PostLoad	Executed when TPD file is unloaded.
PreMSN	Executed when a new UUT MSN is entered.
PostMSN	Executed before a new UUT MSN is accepted.
PreSequence	Executed prior to running a sequence of tests.
PostSequence	Executed after running a sequence of tests.
AbortTest	Executed when an abort occurs.

All lines within the [**SetupObjects**] section are formatted as:

keyword = *FileName; Arguments; PartNumber; Version*, *Date*, *Time*, *CheckType*

Field	Description
FileName	Name and path of setup object program.
Arguments	Optional command line arguments for setup object.
PartNumber	Optional part number of executable file.
Version	Optional version of file (e.g. 2.1.3).
Date	Optional modification date of file.
Time	Optional modification time of file.
CheckType	Kind of version checking: CHECK_EXACT, CHECK_ORLATER (default), CHECK_NONE

The part number and version information is only necessary if version checking is to be performed on the setup object when the TPD file is loaded.

Example

[SetupObjects]
// define a preload object with a full path to a common program repository
PreLoad = **:\common\setup\gasload.exe;/t5;Gasload;3.0**

// load a data sharing DLL when sequence initiated
PreSequence = Datashar.DLL;;DATASHAR;2.5,12/9/95,,CHECK_ORLATER;
PreSequence = Poweron.exe;/L +12;POWERON;1.5.04,,,CHECK_EXACT;
PostSequence = Poweroff.exe;;POWEROFF;1.4,6/14/57,,CHECK_ORLATER;

// run 2 programs when an abort occurs
AbortTest = Abort2.exe;;;3.4.1,,,CHECK_NONE
AbortTest = Abort1.exe;/a;;

6.5.5 Test Objects

This section of the TPD file allows the test designer to specify information about each test to be performed on the UUT. Individual tests and groups of tests are identified by their paragraph subset number and/or a unique test ID. Each test's TOD file, unique ID, part number, revision, executable file, command line arguments and prerequisites are specified on a single line in this section. An optional second line is used to define branch conditions for controlling execution flow.

If a test program help file is defined in the TPD, each test object can be linked to it. From within the Test Executive, the **F1** key causes a help topic to be displayed that corresponds to the selected test object. The optional test ID is used as the keyword for linking to the appropriate help topic. If a test ID is not defined, the index position of the selected test object is used as a context value. These values, which start from zero, are internally mapped to topics in the help file. The context mapping is defined in a project file when the help file is built.

The Test Executive searches for test object files in a subdirectory (**\test**) to the home directory of the TPD file. Each of the referenced files is validated and optionally version checked. For more information, see *Component Validation*.

For more information on formatting and syntax, see *Test Objects Syntax*.

6.5.5.1 Test Objects Syntax

This section of a TPD file determines the test hierarchy for the Test Executive as well as the characteristics of test object executions. Both test groups and test objects are defined in this section. Test objects represent executable elements, while test groups are collections of test objects (and other test groups).

A test group is defined by a line containing just a paragraph # and name. The members of the test group are indicated by their paragraph #'s. For example: Test Group 1.0 contains all test objects that are 1.1 to 1.n. This includes 1.1.1 to 1.n.n.

File names containing embedded commas or semicolons should be enclosed by double quotes (“). This protects them from being incorrectly parsed.

Note: The 32-bit version of the Test Executive can only execute 32-bit test objects. Test objects that are 16-bit are detected before execution and an error is generated.

All lines within the [**TestObjects**] section are formatted as:

Test = *Paragraph; Name; TOD File; TOD P/N; TOD Version; Test Executable; Arguments; Test P/N; Test Version; Prerequisites*

TestBranch = [*Conditional*] *Command*

Field	**Description**
Paragraph	Required, paragraph style of numbering (e.g. 1.1.4., 1.2.3., 3., etc.) controls test order.
Name	Up to 40 character test description.
TOD File	Name of optional TOD file.
TOD P/N	Part number for TOD file; always checked against TOD contents.
TOD Version	Comma delimited field with TOD version check information. For format, see *Other Files Section*.
Test Executable	Specifies an executable or DLL test object file.
Arguments	Command line arguments for test object executable.
Test P/N	Part number for test object executable; always checked against file.
Test Version	Comma delimited field with test object version check information. For format, see *Other Files Section*
Prerequisites	Optional list of prerequisites that must be met before test object runs.

Conditional	Pass or fail conditional for subsequent command.
Command	Post-execution command for repeating, stopping or branching.

Test object executables, TOD files and test object help files are all verified to exist when a TPD file is loaded by the Test Executive. TOD and test object executables are also checked for correct part numbers and versions. For more information on how file references are validated, see *Component Validation*. Test object executables can also be DLLs and ActiveX DLLs. In those cases the *Arguments* field also defines the name of the function being called (see *DLL Startup* and *ActiveX Test Object Syntax*). For more information on test object DLLs, see *DLL Loading and Execution*. For more information on ActiveX test objects, see *ActiveX DLL Test Objects*.

Example

```
//  Section Header:
[TestObjects]

// Test Group: prerequisites defined here are inherited by all subtests
Test = 1.;Power Tests

// Individual Test: with comma delimited unique ID in the name field
Test = 1.1.;FirstPowerTest, FPT1;pow1.tod;55647;A;pow1.exe;/J;;;;

// Individual Test: with branch condition (stop testing if this test object fails)
Test = 1.2.;Second Power Test;pow2.tod;55655;A;pow2.exe;/2;POW2;2.6;;
TestBranch = if Fail then Stop

// Individual Test:
Test = 1.3.;Third Power Test;pow3.tod;55668;A;pow3.exe;/W;;;1.1 P,1.2 P;
```

6.5.5.2 Test Description

The test description field of a test object definition can be further subdivided by commas to include a unique ID and a help file:

Name [, *UniqueID* [, *HelpFile*]]

The optional *UniqueID* for a test object allows links to a help file for context sensitive support. It also provides a controlled reference name for commands and queries from external applications such as client agents. The optional *HelpFile* entry allows a test object to reference its own help file rather than the Test Program help. Help files for individual test objects must reside in the same directory as the test object executable.

To link to a help file topic, the unique test ID should match a keyword defined for that topic. Tests without a unique ID are linked to a help file topic by their position in the test list (0 to N). The mapping of these values to context strings is defined in the help project file, which is used when compiling the help file. Context strings in a help file can only be composed of alphanumeric, period (.) and underscore (_) characters.

6.5.5.3 Test Object Command Line Arguments

The command line argument field, of a TPD test object entry, is where information is defined that is to be passed to the test object at the start of execution. This information might include behavior options such as diagnostics, error reporting or flow control. This is a very powerful mechanism that is used extensively by the sample test object (see *Appendix G - Sample Test Object*) to direct its operation. For a test object DLL, this field is also used to specify the DLL function that is to be executed (see *DLL Startup* in *DLL Test & Setup Objects*).

To the contents of this field, the Test Executive appends additional command flags that direct the operation of the TEXDLL. These include breakpoint settings and various file names (e.g. current TPD and TOD). These command flags are eventually passed to the TEXDLL via the texStart() function. In order to avoid a conflict between TEXDLL command flags and those used for a test object, it is strongly recommended that test object command flags be prefaced by "**/usr**" (e.g. /usrTest21). The Test Executive is guaranteed never to use command flags prefaced in this manner.

Breakpoint command flags, generated by the Test Executive, can be overridden by items specified in the test object's argument field (as of Test Executive version 6.2). This is useful if a test object has certain restrictions or requirements associated with breakpoints. For example, verify breakpoints could be disabled if they might cause timing problems for measurements within a test object. Or, it might be desirable to always enable warning breakpoints for certain test objects.

Command flags serve several purposes within the execution of a test object. Some command flags are generated by the Test Executive for the exclusive use of the TEXDLL. These include the file names and current test id. Other flags (i.e. breakpoints) are generated by the Test Executive but they can be overridden by the test object's argument field. The abort delay flag ("**/ad**") is only received by the TEXDLL if it is included in the test object's argument field. Finally, some command flags are generated by the Test Executive solely for the use of a test or setup object. These include the sequence status and repeat count.

To support post sequence processing, a command flag (**/SeqPass**, **/SeqFail** or **/SeqError**) is created by the Test Executive for all PostSequence setup objects. The command flag indicates the execution status of the recently completed sequence. This allows the post sequence setup objects to conditionally initiate REQUEST FOR DEVIATION (RFD) and NO-TEST processing.

A repeat count flag ("**/rpt** 1:2:1") is passed to test and setup objects in order to indicate how many times the test object has been executed during the current sequence. The addition of conditional branching has created the possibility that a test object may be executed more than one time during a sequence. Thus the test object may need to adjust data logging based on the number of times it is executed. Specifically for PDL logging, each time a verify is repeated it must be recorded with a different id (e.g. TestVCC_1, TestVCC_2, TestVCC_3, etc). The format of the argument is:

/rpt *RepeatCount*:*SequenceCount*:*LoopCount*

The *RepeatCount* identifies the current repeat loop (0 for first execution of test object). The *SequenceCount* is the number of times the test object has been executed during a sequence (starts at 1). This helps to identify when a test object is executed again based on a Goto operation or Repeat command on a Test Group. *LoopCount* is the number of times the Test Executive has looped on the selected test objects (always 1 for a simple Run).

6.5.5.4 Prerequisites

Prerequisites are used to control the execution order of test objects. This allows test objects to assume certain initial conditions. For example, a power supply calibration test might be set as a prerequisite for any test object that applies power to the UUT. An unfulfilled prerequisite causes the Test Executive to hold (not run) the dependent test object. The following is a list of key points concerning prerequisites:

- Prerequisite types are Pass, Run, Immediate Pass and Immediate Run.
- A child or subordinate test inherits prerequisites from its parents.
- A warning is displayed if prerequisites for any one item exceed 20.
- For a long list of test objects that must be run in order, make each a prerequisite of the next (i.e. like a linked list).

The *Prerequisites* field in a test object entry defines one or more prerequisite tests. Multiple prerequisites are specified using a comma-delimited list:

Paragraph PreReq, *Paragraph PreReq*, *Paragraph PreReq*, …

The *Paragraph* entry identifies the paragraph number of the test object, or test group, that must meet the prerequisite condition. The *PreReq* entry is the type of prerequisite to enforce:

Type	Description
R	Specified test must have been executed.
IR	Specified test must be executed immediately prior.
P	Specified test must have been executed and passed.
IP	Specified test must be executed and pass immediately prior.

The two "immediate" prerequisites are used to enforce a specific sequence of test execution. If Test B expects the UUT to be in a particular state following Test A, it can use an Immediate Pass prerequisite to guarantee that A is executed in the same test sequence.

Note: Prerequisites make tests conditional on other items in the test list. These conditions are not visible unless an operator specifically views them on the test list. Consequently, prerequisites should be used sparingly and they should be prototyped with a TPD file before interdependent test objects are created.

6.5.5.5 Test Branching

Test branching provides additional control over test execution flow. This entry is a way to repeat, branch or halt test execution based on a test object's results. A test branch entry is formatted as:

TestBranch = [*Conditional*] *Command*

The optional *Conditional* parameter tests the results of the immediately preceding test object (i.e. "**If Pass Then**" or "**If Fail Then**"). If the preceding entry is a test group (i.e. folder), the conditional is based on the test objects contained within the group. The *Command* is not executed unless the conditional is met.

The *Command* parameter is either **Stop**, **Goto** or **Repeat**. The Repeat command has a repetition count and can optionally be made dependent on the operation's result (e.g. Repeat(5) While Pass). The following describes the commands and their behavior:

Command	**Description**
Stop	Halt the test sequence execution.
Goto	Move sequence execution to the specified item in the test list. The destination is defined by a paragraph number or unique test id.
Repeat (X)	Repeat preceding entry 'X' times.
Repeat (X) While Pass	Repeat preceding entry 'X' times while it passes (or fails).
Repeat (X) While Fail < Y	Repeat preceding entry 'X' times as long as it fails (or passes) less than 'Y' times..

The following are examples of test branching syntax:

TestBranch = if Pass then Goto 1.2.3
TestBranch = if Fail then Goto TestDiag38
TestBranch = Goto 1.2.3
TestBranch = if Fail then Stop
TestBranch = Stop
TestBranch = if Fail then Repeat(15) While Pass
TestBranch = if Fail then Repeat(4) While Fail < 2
TestBranch = Repeat(8)
TestBranch = Repeat(6) While Pass

When a TPD file is loaded by the Test Executive, syntax errors for test branch entries are written to the Debug Log window. An example TPD (TxBranch.tpd), with a variety of branching entries, is included with the installation.

The final, displayed, test object status is typically based on the result of the test object's last execution. This is for both test branching and repeats. For example, a "Repeat(5) While Fail" quits after 5 failing executions or a single pass. The final test object status is thus fail or pass, respectively. Therefore, it is important to choose repeat criteria carefully. In the previous example, the test object is considered a pass if it only passes 1 out of 5 possible executions. Note that all test object executions are logged by the selected mechanisms (database and/or PDL).

For PDL logging, each verified measurement must be assigned a unique identifier. This means that during test branching or repeats, a test object must modify its verify ids during successive executions within the same sequence. Typically this means that a repetition count is appended to the original verify identifier. The Test Executive provides repeat and sequence counts to each test object via command line arguments (see previous section). Each test object is responsible for managing its own unique verify ids. When using TOD files, this is accomplished by retrieving the verify with texGetTODVerify(), modifying the verify id string and executing the verify with texVerify(). This common set of operations is easily built into a shared function within the test object code.

Note: Conditional test branching can quickly become complex and confusing to an end user. It is important to design it carefully, use it sparingly and test it extensively. Testing can be accomplished with the supplied sample test object and its "User Controlled Failure" (/ucf) command line argument.

Ideally, all possible permutations of branching should be exercised before a test program is released to production. This can be a daunting task if there is a large number of nested and conditional branches. Also consider that a test operator might select one or more individual test objects to execute (if **MultiSelect** is active). What affect might this have on your tightly crafted branching? Generally it is best to keep branching as simple as possible. Repeating on a single test object is a very contained and easily verified behavior. Using conditional branching to either run or bypass a diagnostic test object is also easy to understand and debug.

When trying to puzzle out the exact behavior of conditional branching, it is very helpful to use the often neglected Test Executive Step Mode. This single steps through the selected items on the Test List each time the Execution button is pressed. Single stepping is often the only way to observe a branch to a test group (folder). During normal operation, test groups are highlighted and restored too quickly to see.

6.5.6 Action Objects

The inclusion of action objects in the TPD file allows the Test Executive to list these items in the Tools Menu on the operator interface. These tools are programs, documents or web sites that can be of assistance to the operator during the testing process. A guided probe application or instrument soft front panels are good examples of action objects. A test procedure manual or interface adapter pictures are examples of documents that might be made available via the Tool Menu.

Action objects are optionally disabled while a test object is executing. This is used to prevent an operator from running a tool that might conflict with a test execution.

Non-executable action objects (e.g. documents or web sites) are opened by the application associated with their file type. For example, an Adobe Acrobat Document (*.pdf) is typically associated with a version of the Adobe Reader.

Note: File type associations can be changed by an operator, or when new software is installed. This is particularly true for graphic file formats (*.jpg, *.bmp, *.gif, etc) which many applications want to "own". These changes could affect the presentation of test documentation. Therefore it is important to monitor file associations and to try to use ones that are usually associated with a single application (e.g. *.pdf, *.hlp, *.chm, *.htm).

The Test Executive searches for action object files in their relative directory first (**\tools**), followed by an execution path search. This allows action objects to be placed in a common directory for sharing with other test programs. For more information, see *Component Validation*.

For more information on formatting and syntax, see *Action Objects Syntax*.

6.5.6.1 Action Objects Syntax

These objects (tools) are programs, documents or web sites that can be used by the operator during the testing process. They are added to the Tools Menu when the TPD file is loaded.

All lines within the [**ActionObjects**] section are formatted as:

Action = *Name*; *FileName; Arguments; PartNumber; Version*, *Date*, *Time*, *CheckType*; *Run*

Field	**Description**
Name	Description used for Tools Menu.
FileName	Name and path of action object program, document or web site.
Arguments	Optional command line arguments for action object.
PartNumber	Optional part number of executable file.
Version	Optional version of file (e.g. 2.1.3).
Date	Optional modification date of file.
Time	Optional modification time of file.
CheckType	Kind of version checking: CHECK_EXACT, CHECK_ORLATER (default), CHECK_NONE
Run	Enable menu entry: 0 = no active test, 1 = always (default)

The part number and version information is only necessary if version checking is to be performed on the action object when the TPD file is loaded.

Note that the name field can be used to control formatting in the Tools Menu. Using a hyphen (-) for a name creates a dividing line in the Tools Menu. An ampersand (&) preceding a letter makes that letter an access key for keyboard control. Access keys are identified by an underline (e.g. Calculator, Timing Set). They are activated by holding down the **Alt** key while typing the indicated letter. Note that some systems may hide the access key underline until the **Alt** key is pressed.

Web page references are indicated by the presence of a leading "**www.**" (e.g. www.cnn.com). Local HTML documents can be accessed by simply entering their file name in the tool list. When selected, the application associated with "**.htm**" files opens and displays the document.

Example

```
[ActionObjects]
Action = &Fault Dictionary;fltdict.exe;arg;23490;2.4; 0  // Note “&” for access key
Action = -;;;;;1                    // menu separator bar
Action = &Calculator;calc.exe;;Calc;3.10.0.103; 1     // Always enabled
Action = -;;;;;1
Action = &Timing Set;clock.exe;;clock;3.10,11/1/93, , CHECK_ORLATER; 0
Action = -;;;;;1
Action = Test &Procedure Document; TestProc.hlp; ; ; ,5/14/04, , CHECK_ORLATER
Action = Test Procedure &Web FAQ; www.XYZCorp.com\Test1234\Faq.htm
```

6.5.7 Power Monitor

Some electronic assemblies have a limited amount of time that they can be powered up without damage. The Test Executive supports a monitoring mechanism that tracks when power is applied to a UUT. If the defined time limit is reached, current testing is aborted and no further testing is allowed until the UUT cools down. The power state and time remaining are shown on the Power Monitor display.

If power monitoring is desired, each test or setup module must call **texPowerMonitor**, in the Test Executive DLL, to indicate when power is ON or OFF.

The Test Executive tracks power in one of two possible ways. The first uses an internal routine to monitor power cycles based on parameters specified in the TPD file. These parameters include a maximum power-on limit, a recovery time and an optional cooling rate. The second method of power tracking uses an external DLL. The external DLL is notified each time the UUT's power state changes. The DLL determines the power limits and recovery based on the specific requirements of the UUT.

If the power limit is reached, the Test Executive aborts the current test and runs the specified abort objects. Testing is then disabled until the necessary recover time has elapsed. If a new UUT is installed, the power limit is immediately reset.

Note: The recovery time period is not started until power is removed from the UUT. Thus it is important that an abort object remove power and notify the Test Executive with a texPowerMonitor call.

For more information on the format of this section, see *Power Monitor Syntax*.

6.5.7.1 Power Monitor Syntax

The [**PowerMonitor**] section of a TPD file contains control information for UUT power monitoring. This includes a sampling interval, and power limits or an external monitor DLL. The power limits are used by the Test Executive for internal monitoring. Alternately, the external DLL is called upon to decide power limits and recovery times.

Keyword	Description
TimeLimits	Defines power limits and recovery times for internal power monitoring.
MonitorDLL	Defines a DLL for monitoring power.
UpdateInterval	Optional sample rate for power monitoring (1-60s). Default rate is 5 seconds.

To activate internal power monitoring use the following:

TimeLimits = *MaxTime*; *RecoverTime*; [*CoolingRate*;]

Field	Description
MaxTime	Maximum accumulated time that power can safely be applied to a UUT (seconds).
RecoverTime	Amount of time required for a UUT to cool down once the *MaxTime* is reached (seconds).
CoolingRate	Optional cooling rate for accumulated power-off time.

The optional cooling rate decreases the accumulated power-on time based on the amount of time that power is off (e.g. 90/20 = 90 seconds of no power reduces the accumulated power-on time by 20 seconds). This accounts for extended times that the UUT is powered down or when testing is idle.

If internal monitoring is not defined by **TimeLimits**, an external DLL is defined by:

MonitorDLL = *FileName;* ; *PartNumber; Version*, *Date*, *Time*, *CheckType*

Field	Description
FileName	Name and path of power monitor DLL. Default path is "\setup"
PartNumber	Optional part number of executable file.
Version	Optional version of file (e.g. 2.1.3).
Date	Optional modification date of file.
Time	Optional modification time of file.
CheckType	Kind of version checking: CHECK_EXACT, CHECK_ORLATER (default), CHECK_NONE

The part number and version information is only necessary if version checking is to be performed on the DLL when the TPD file is loaded. Note that there is an empty field between the *FileName* and *PartNumber* fields. This is included in order to match the syntax of setup object declarations.

Example

```
[PowerMonitor]
TimeLimits = 200; 100; 50/20
UpdateInterval = 4
```

6.5.7.2 Power Monitor DLL

Since some UUTs have more complex power-on restrictions, an optional external power management module is supported by the Test Executive. This module is a user-created DLL that embodies the exact power restrictions for a particular assembly. The DLL is defined in the power monitor section of the TPD file.

As power is cycled on an assembly, the Test Executive notifies the DLL by calling **usrPowerMonitor**. Periodically the Test Executive also queries the DLL as to whether it is safe to continue or start testing. If the function returns a zero value in response, the Test Executive aborts the current test and runs the specified abort objects.

The power monitor DLL must also be specified as a PreLoad setup object. This then keeps the DLL loaded throughout the life of the TPD. It is left to the test developer to add it to the setup objects section because it may need to be loaded in a specific order in relationship to other setup objects. This also provides an opportunity to execute any necessary initialization routines that might be required. For more information on loading or executing a setup object, see *DLL Loading and Execution*.

Note: The power monitoring function has different input parameters and operational requirements than a function being called in a test object or setup object DLL.

The **usrPowerMonitor** function interface is the same as texPowerMonitor. One additional input value, TEX_POWER_RESET (= 4), is used to notify it when a new UUT has been installed. This function should not call texStart or any of the other TEXDLL functions. It should also not halt execution with operator prompts or other interface elements. The function is periodically called based on the update interval specified in the power monitor section of the TPD file. It may also be called sooner if a power state change occurs (i.e. power is turned on or off).

Example Power Monitor DLL

An example power monitor DLL (ExtPowerMonitor.dll) is included with the Test Executive installation. Its purpose is to allow prototyping with a functioning external power monitor DLL. It also supports an initialization function (usrPowerLimits) that allows the power limits to be set when it is called as a setup object. This mechanism is exercised by one of the sample TPD files (TxOptn32.tpd).

The source code for the power monitoring functions is part of the example test object DLL. This is included in a subdirectory to the Test Executive ("\Samples\Msvc4\TxObjDLL").

6.5.8 Other Files

To define additional files necessary for a test program, the TPD supports an [**OtherFiles**] section. This section contains a list of files that are verified and version checked by the Test Executive, when the TPD is loaded. This ensures that all the necessary components are present for running a test program. These might include documentation files, instrument driver DLLs, data files or configuration files.

This section also serves as a way to document all of the files related to the test program, which might be useful for an automatic installation utility. The format of the entries is similar to other file definitions in the TPD, see *Other Files Syntax*.

6.5.8.1 Other Files Syntax

The OtherFiles section of a TPD file contains a list of files required by the test program. Files listed here are verified as to their location and, optionally, their version. This is a useful way to validate a test configuration before testing begins

Keyword	Description
File	Defines a file that's not an executable or INI.
IniFile	Defines an INI formatted text file. This file type has an **[Identification]** section similar to a TOD or TPD file.
ExeFile	Defines an executable file (e.g. program, DLL, etc.)

All lines within the [**OtherFiles**] section are formatted as:

keyword = *FileName; PartNumber; Version*, *Date*, *Time*, *CheckType*

Field	Description
FileName	Name and path of file.
PartNumber	Optional part number of file.
Version	Optional version of file (e.g. 2.1.3).
Date	Optional modification date of file.
Time	Optional modification time of file.
CheckType	Kind of version checking: CHECK_EXACT, CHECK_ORLATER (default), CHECK_NONE

The part number and version information is only necessary if version checking is to be performed on the file when the TPD file is loaded (see *Version Checking* section).

The *FileName* follows standard Windows naming conventions, ending with an appropriate extension. The *PartNumber* describes an internal identifier for verifying the file. This can be an ASCII string in an INI file (IniFile) or the internal name assigned to an executable (ExeFile). The *Version* field can be the revision contained in an INI file (e.g. B) or the numeric version assigned to an executable (e.g. 3.0.20). *Date* and *Time* refer to when the file was created. The **keyword** for the line determines the file existence and version checking process (see also *Component Validation*).

Except for *FileName*, any of the fields can be left blank. The default *CheckType* is CHECK_ORLATER. Note that CHECK_NONE does not disable validation of the file's internal part number.

Example

```
[OtherFiles]
File = C:\tools\socket.exe; 1300T49; 3.10.0.103, , ,CHECK_ORLATER;
ExeFile = timeit.exe; TIMEIT; 1.1.2, , , CHECK_EXACT;
IniFile = C:\inidir\version.ini;323245; , , , CHECK_NONE;
IniFile = \test\tobj45.ini;
```

6.5.9 Configuration

This section of the TPD file allows a test developer to define the types of data associated with a test configuration. The data items are selected from the set of PDEL test event parameters (e.g. Work Order, Test Adapter, Test Procedure, etc.). These parameters can be disabled, assigned static values, selected from a list or edited by an operator.

This data can be defined for operator entry with default values, a selection list or assigned a specific, non-changeable value. Data input and output restrictions control formatting for the operator. Most of the configuration data fields can also be disabled.

Data Control	Application
Input restrictions	Fields can be designated to allow only: numeric characters, alpha characters, a combination of alpha-numeric characters, and a minimum number of characters. The use of a "minimum" number of characters requirement, coupled with an output format, gives the test designer the flexibility to specify an input range. For example, the Operator field may require at least three characters but no more than seven.
Output formatting	Specify fields requiring embedded literal characters, such as dashes or parenthesis. Indirectly sets a maximum number of characters limit.
Editable parameters	An input field editable by the operator. Displayed on the Test Configuration form as a "normal text box".
Static parameters	An input field not editable by the operator. Displayed on the Test Configuration form with a gray background.
Default values	Values automatically entered into the form for operator guidance or convenience.
N/A parameters	A parameter not applicable for the specified Test Program. Appears on the Test Configuration form as "N/A" on a gray background.

If editable, the operator is allowed to input specific information applying to test chambers, test sets, interface adapters and such. UUT part numbers, revisions and work order numbers are defined by the TPD file also. The configuration data is logged with the test results in the database and PDL files. A test is not run until all of the fields are populated.

The Test Executive creates an associated Test Program Configuration (TPC) file that contains configuration information entered while a TPD is loaded. The TPC file is used to reload the configuration data when the TPD file is again loaded. For more information, see *Test Program Configuration File*.

Note that an underscore is an invalid character for a data field with a format mask. For more information about the data control options mentioned, see *Configuration Syntax* and *Appendix E - Formatting Mask.*

6.5.9.1 Configuration Syntax

This section of the TPD file allows the test developer to define the data and optional formats associated with a test configuration. Each line consists of a keyword and up to 3 semicolon delimited fields.

Keyword	Description
UUTPartNumber	Can not be **N/A**
UUTRevision	Can not be **N/A**
WorkOrderNumber	
WorkOrderOperation	
TestChamberPartNumber	
TestChamberRevision	
TestChamberMSN	
TestChamberTemp	
InterfaceAdapterPartNumber	
InterfaceAdapterRevision	
InterfaceAdapterMSN	
TestSetPartNumber	(a.k.a. Test Station)
TestSetRevision	(a.k.a. Test Station)
TestSetMSN	(a.k.a. Test Station)
TestProcPartNumber	
TestProcRevision	
TestLocation	Defined by IEEE-1545
Custom1	Optional, user-defined entry

All lines within the [**Configuration**] section are formatted as:

keyword = *Value; Format; MinSize*

Field	Description
Value	Default, static or list for parameter.
Format	Formatting pattern, see *Appendix E - Formatting Mask.*
MinSize	Minimum number of required characters.

The *Value* field can be empty, or it can define a default value that is always placed in the field when the TPD file is loaded:

TestSetRevision = A

The data entry control is disabled if the *Value* field is set to non-applicable:

TestSetRevision = N/A

The entry becomes a non-editable static value when it is surrounded by angle brackets:

TestSetRevision = <A>

For a drop-down selection, the entry becomes a list of up to 30 values, separated by commas, enclosed in square brackets:

TestSetRevision = [A, B, C, D]

As a default, the first item in a list is automatically entered in the field when the TPD file is loaded. As an option, the field's contents can be restored to their most previous value (i.e. when the TPD was unloaded last) by making RESTORE the first item in the list:

TestSetRevision = [RESTORE, A, B, C, D]

The information to be restored is read from the TPC file. If the information in the TPC file does not match the current contents of the drop-down list (which is very common during TPD development) the field is left blank.

In some situations there is a requirement that the test operator actively enter certain information before testing begins. For those circumstances, the first item in the list should be SELECT:

TestSetRevision = [SELECT, A, B, C, D]

When a SELECT is present, the test operator is required to chose an item from the list before the Test Executive runs its first test (following a TPD load). Subsequent tests will run with whatever the operator has selected for that field. So, a SELECT at the beginning of a list prevents a field's contents from being set to a default value, or restored from the TPC file, when the TPD file is reloaded. Fields with the SELECT option are initially displayed as blank.

The **TestSet…**, **TestLocation** and **Custom1** keywords can alternately be defined in the Test Executive Configuration file. This allows their settings to be shared by all the TPD files that operate on that test station.

Configuration keywords missing from the TPD file are assumed to be editable and required. The Test Executive does not allow a test to be executed without all of the required configuration information being entered, and meeting the format requirements.

When working with masked edit formats, a default value must contain the same literals as the format mask. For example, if a keyword has a format mask of "##-##", a default value should be entered as "33-33". If a minimum of two characters is acceptable in this instance, the literal must still be entered as "33-". See *Appendix E - Formatting Mask* for more information on formatting masks and literals. Note that when a format mask is defined for a data field, underscores are used as placeholders. Therefore, underscores are not allowed as part of the entered data.

A format mask can include optional placeholders for characters (e.g. "a" = optional alphanumeric character). If optional placeholders are used, it is necessary to also specify a minimum number of characters for the data (e.g. TestProcRevision = ; ?a; 1).

If no minimum size is specified for a keyword, the data entered in the Test Configuration Window must exactly match the format specified. If no format is specified, a single character suffices. The format field also controls the maximum size of the data entered.

Example

```
[ConfigurationInformation]
TestChamberTemp = N/A              // disabled field
TestSetPartNumber = 13579B         // default value
UUTPartNumber = <123456789>        // angle brackets indicate static data
InterfaceAdapterMSN = n/a
TestSetMSN = <555-A4444>
TestSetRevision = A; ?? ; 1        // an alpha mask of 1 or 2 characters
TestSetPartNumber = ; ###-### ;    // field requires 6 numeric characters
```

6.5.10 Configuration Labels

The configuration information associated with a test is based on the required and optional header statements that are part of the IEEE-1545 specification. In some circumstances the labels for these data fields do not match commonly used terminology at a facility or test location. This section of the TPD file is used to rename configuration labels for display and printed reports. Note that the original field names are still used as keywords in the TPD, TPC and database files.

This section also provides a way to define new headers for storing configuration data in a PDL file. The IEEE-1545 specification defines a set of required and optional header statements. These cover "test event data" as opposed to "parametric test data". The required headers include such items as: UUT_IDENTIFICATION; and the optional headers have items such as: TEST_CHAMBER_IDENTIFICATION. IEEE-1545 also allows new header statements to be created via a DEFINE_HEADER statement. Within this section of the TPD file, configuration data can be assigned a new PDL header for storing information. This mechanism allows PDL headers to be defined that more closely match their associated data.

For more information about the configuration label options, see *Configuration Labels Syntax*.

6.5.10.1 Configuration Labels Syntax

This optional [ConfigurationLabels] section of a TPD file allows a test developer to define custom labels and custom PDL headers for displaying and logging test configuration data. Each line consists of a keyword and up to 2 semicolon delimited fields. Some of the keywords represent information for a piece of physical equipment, consequently there are three configuration items associated with the keyword (i.e. P/N, Rev and S/N). Other keywords are individual entries that can represent any aspect of the testing process.

Keyword	**Description**
WorkOrder	One field
WoOperation	One field
TestSet	P/N, Revision and S/N fields
InterfaceAdapter	P/N, Revision and S/N fields
TestChamber	P/N, Revision and S/N fields
TestProcedure	P/N and Revision fields
TestLocation	One field
Custom1	One field

All lines within the [**ConfigurationLabels**] section are formatted as:

keyword = *CustomLabel* [*; PdlHeader*]

Field	**Description**
CustomLabel	Label to use for data entry, display and printed datasheets.
PdlHeader	Header entry to use when logging this data to a PDL file (optional).

The optional *PdlHeader* field allows the associated configuration information to be stored as something besides the required or optional PDL headers. This is accomplished by placing DEFINE_HEADER statements in the PDL file. For example, suppose a test system uses a vibration table during testing and doesn't use a test chamber. The test chamber fields could be reused to enter/display vibration table information:

```
[ConfigurationLabels]
TestChamber = Vibration Table; VIBE_TABLE
```

The information entered for the vibration table would then be stored in a PDL file with these newly defined header statements:

```
DEFINE_HEADER VIBE_TABLE_IDENTIFICATION = tpd-defined, S10;
VIBE_TABLE_IDENTIFICATION         = 12345-6789;
DEFINE_HEADER VIBE_TABLE_REVISION = tpd-defined, S128;
VIBE_TABLE_REVISION               = D;
DEFINE_HEADER VIBE_TABLE_SERIAL_NUMBER = tpd-defined, I9;
VIBE_TABLE_SERIAL_NUMBER          = 09876543;
```

Notice that the defined PDL header, VIBE_TABLE, has been modified to include each of the three fields associated with that configuration item (i.e. IDENTIFICATION, SERIAL_NUMBER, and REVISION).

6.5.11 User Defined

Sometimes a test program requires additional information in order to operate. Often it would be convenient if the information could be stored in the test program's TPD file. This would then consolidate test information into one, easily managed file. The Test Executive permits user-defined sections in a TPD file if the section name begins with "**usr**" (e.g. [usrTestData]).

To access user-defined information, a test object can first retrieve the name of the TPD file with the TEXDLL function texGetTPD. Then it can read and parse the TPD file contents or use the Windows API function, **GetPrivateProfileString**, to read unique keyword entries from the user-defined section. The following example shows a user-defined section with unique keywords that would easily be read by **GetPrivateProfileString**:

```
[usrTestData]
RAM_Address = 8000
ROM_Address = 3000
Test_Bus = Y
```

Note that if you want to support the same comment mechanism as other parts of the TPD file, you need to strip trailing comments from a line before parsing its contents. For example, the following entry might confuse a parsing algorithm if the comment is not removed first:

```
ROM_CRC = 0x357E  // CRC = Address from 0 to 400
```

The "**usr**" prefix is also a quick way to hide a section of the TPD file from the Test Executive. This can be useful during development as a way to quickly enable or disable portions of a TPD file. For example, you may want to work without power monitoring enabled. You can simply add "**usr**" to the beginning of the section header (e.g. [usrPowerMonitor]) and the Test Executive ignores it.

6.5.12 Sample TPD Files

6.5.12.1 Simple TPD File

The purpose of this example is to show how simple a TPD file can be. The wealth of options supported by the Test Executive sometimes obscures the inherent simplicity of using defaults and minimal settings. While this might not be robust enough for production testing, it does serve as a simple starting point for developing a functional test program.

```
[Identification]
PartNumber = 2932634-0021
Revision = D
FormatVersion = 2.0
ProgramName = F-16 Subsystem 95-Q

[Options]
PDEL = ALWAYS      // always log test results to a PDL file
Database = ALWAYS  // always log test results to the database

// Run the following program if an abort occurs during testing

[SetupObjects]
AbortTest = tsetup32.exe;/a

// This test list has a simple hierarchy with no prerequisites.

[TestObjects]
Test = 1.;Full Run
Test = 1.1.;Subsystem A32-Q;;;;txdemo32.exe
Test = 1.2.;Up Link;;;;txdemo32.exe;/f
Test = 1.3.;Down Link A;;;;txdemo32.exe
Test = 1.4.;Down Link B;;;;txdemo32.exe
Test = 1.5.;Norden Sight Alignment;;;;txdemo32.exe
```

6.5.12.2 Complex TPD File

```
[Identification]
PartNumber = 2942234-0001
Revision = C
FormatVersion = 2.0
ProgramHelp = texdemo.hlp
ProgramName = Test for Test Executive

[Options]
PDEL = USER                    // Let operator control PDEL logging
DATABASE = Always              // Database logging is always active
VersionCheck = N                       // No version checking for any files
MultiSelect = Y                // Allow multiple tests to be selected in Test List
OperatorID = ?###### ; 3               // Op ID starts with a letter, at least 3 characters
ReportPrinter = custom.exe;;1234-567;A
TestConfigurationPrompt = Y    // Always prompt for TPC data when a test is run
UnloadDLLs = Normal
Breakpoints = 0; 0; 2; 1; 1            // Disable Data/Trace, break on failing Verifies and
                                       // Warnings, enable Stop-on-Fail behavior.

// Multiple setup objects of each type are supported in the next section.
// They are executed in the order that they are defined.

[SetupObjects]
PreSequence = preseq.exe; ;225388-0001;2.3
PostSequence = postseq1.exe; /p1 ;2232334-0002;4.5
PostSequence = postseq2.exe; /p2 ;2232334-0002;1.2
PreLoad = preload.exe; /pl ;1123445-0003;1.45
PreMsn = premsn.exe; ;3222111-0001;2.3
AbortTest = abort.exe; /a;1120990-0004;3.0,,,CHECK_EXACT

// The test objects in the next section can be defined in any order.  Their order
// in the Test Executive is based on their associated paragraph number.  An optional
// part of the test object name can be used to identify the test for external control
// (i.e. agent interface) or Help file indexing (e.g. Test = 1.2.1.;Up Link,TESTDEF;...).

 [TestObjects]
Test = 1.;Full Run,Introduction
Test = 1.1.;Subsystem A32-Q,TEST_A32Q;;;;txdemo32.exe;; ;2.0,,,CHECK_EXACT
Test = 1.2.;Navigation Satellite Link,TEST_LINK;;;;;;;;1.1. P
Test = 1.2.1.;Up Link,UpLink;;;;txdemo32.exe;/f;TXP;2.0,11/29/95, ,CHECK_EXACT
Test = 1.2.2.;Down Link,DownLink,dnlink.hlp;;;;;;;;
Test = 1.2.2.1.;Down Link A,DownLink;;;;txdemo32.exe;;;1.2, , ,CHECK_ORLATER
Test = 1.2.2.2.;Down Link B,DownLink;;;;txdemo32.exe;;;1.5, , , ;1.2.2.1. R
   TestBranch = If Fail then Repeat(5) While Pass
```

```
// The ampersand (&) in the following tool names indicate which letter is to be
// underlined in the Tool Menu.  The underlined letter then becomes the access key for
// that item.  An access key provides rapid keyboard selection of items in a menu.

[ActionObjects]
Action = &Calculator;calc.exe;;Calc;3.10.0.103;1
Action = -;;;;;1
Action = &Timing Set;clock.exe;;CLOCK;3.10,11/1/93,3:11:00am;0

// This next section defines files that are associated with this test program.  It is
// a way to document their existence as well as to verify their correct versions.

[OtherFiles]
ExeFile = calc.exe;Calc;3.10.0.103,,,CHECK_ORLATER
File = \test\texprog.c;;,11/28/95,2:48pm,CHECK_ORLATER
IniFile = \test\laser.tod;12345678-1234;B,,,CHECK_EXACT

// Items not specified in the next section are supported in the Test Configuration
// window as editable with no default value.

[ConfigurationInformation]
TestChamberPartNumber = <36556-2563>
TestChamberMSN = 22222
TestProcPartNumber = <7765432-0003>
TestProcRevision = ;??;1
InterfaceAdapterRevision = [SELECT, A, C, Q]
UUTRevision = <A>
UUTPartNumber = <2111221-2212>
Custom1 =                              // must be defined to be shown

[ConfigurationLabels]
Custom1 = Visual Inspection; VISUAL_INSPECT

// The following is a user-defined section
[usrTestLimits]
LowerLimit = 27
UpperLimit = 397
SkipTests = 12, 29, 41, 103
```

6.6 Test Object Definition File

A Test Object Definition (TOD) file is a repository for verify parameters. TOD files are used to isolate test limits from the program that performs the measurements. This allows the limits to be adjusted without rebuilding the executable. This is extremely valuable when adapting tests for engineering changes or when using different limits for production versus environmental test. Limits for several different test objects can be stored in one TOD file. This can make tracking, installing and updating simpler. Up to **1000** verify entries can be stored in a single TOD file.

Test objects access these parameters when they execute texVerifyTOD or texCompareTOD. These two functions compare the stored limits with the results of a measurement. Each verify parameter set is identified by a unique identifier. A test object can also retrieve verify parameters from a TOD with the texGetTODVerify function. This allows verify parameters to be examined, logged or scaled to meet specific measurement or system requirements.

For more information on TOD file formatting, see *Test Object Definition Format*.

6.6.1 Test Object Definition Format

A TOD file contains a list of verify parameters for automatic consumption by a test object. Each TOD file includes two sections: an Identification section and a Verifies section. The Identification section specifies a part number, revision and TOD format version. The Verifies section contains one or more verify definitions. Each verify definition consists of a line that specifies eight test criteria. For more information on these parameters, see *Test Criteria*.

Additional, user-defined sections are permitted in a TOD file if they are placed between the [Identification] and [Verifies] sections. User-defined sections in a TOD file should be prefaced by "**usr**" (e.g. [usrSection1]) and must not contain the following keywords: PartNumber, Revision, FormatVersion or Verify.

A test object consumes TOD verify data based on the unique identifier field. This allows the TOD file to contain verify data in any order. This protects the test developer from subtle problems due to changes in verify sequencing. A TOD file can hold verify data for several different test objects.

A TOD file is associated with a test object in the TPD file. The TOD file name is contained in the test object definition line.

6.6.2 TOD Identification

This section identifies the TOD part number and revision. It also indicates the TOD format version. All lines in the [**Identification**] section are formatted as:

keyword = *string*

Keyword	Description
PartNumber	Required part number for identifying TOD.
Revision	Required revision number of TOD.
FormatVersion	Required TOD format level (**1.0**).

The Part Number and Revision entries are used to identify the TOD file and to validate it against the TOD part number and version declared in a TPD file.

TOD files created by the TOD Builder are assigned a format version of 1.1. This lets the TOD Builder notify the user when a hand-edited TOD file is first loaded. Comments and spacing may be lost when the TOD Builder initially modifies a hand-created file.

Example

```
[Identification]
PartNumber = 54321
Revision = B
FormatVersion = 1.00
```

6.6.3 TOD Verifies

Each entry in this section of a TOD file corresponds to a single verify operation. All of the information necessary for performing and logging a verify is contained in an entry (except for the value of the actual measurement). A test object matches a measurement to its corresponding limits through the unique identifier field. Up to **1000** verify entries can be stored in a single TOD file.

All lines in the [**Verifies**] section are formatted as:

Verify = *VerifyName; UniqueID; LimitType; limit1; limit2; SigDigits; UnitPrefix; UnitType*

Field	**Description**
VerifyName	Descriptive name of verify (up to 40 characters).
UniqueID	Unique identifier for verify (only alpha-numeric & underscore accepted).
LimitType	Required, comparison type: Range, Nom_Tol, Nom_Per, Gr_Than, Gr_Than_Eq, Less_Than, Less_Than_Eq, Equal, Bin_Comp, Pass_Fail.
limit1	Numeric first limit value.
limit2	Numeric second limit value.
SigDigits	Number of significant digits for comparison (i.e. to the right of the decimal point). Range from 0 to 6.
UnitPrefix	Prefix to units (e.g. micro). Use NONE or empty field if no prefix is necessary.
UnitType	Measurement units (e.g. volts). Use NONE or empty field if no units are necessary.

Empty fields are allowed for limits, *SigDigits*, *UnitPrefix* and *UnitType* fields. Blank numeric fields are interpreted as zero (0).

Example

```
[Verifies]
Verify = Measure Power Supply; IDA; EQUAL; 1; 0; 0; MILLI; VOLTS
Verify = Check Noise Calibration; IDB; LESS_THAN; 2; 0; 0; MILLI; VOLTS
// Comments are allowed in the Verifies section
Verify = Adapter Test; IDC; GR_THAN; 0; 0; 0; NONE; NONE
```

6.7 Test Program Configuration File

A Test Program Configuration (TPC) file is created automatically by the Test Executive to store current TPD configuration information. The TPC file maintains a dynamic snapshot of configuration information, as entered by a test operator, for an associated TPD file. Data within the TPC file is used to populate the Test Executive's Test Configuration dialog, when a TPD is loaded, to avoid reentry of repetitive information. A TPC file has the same name as its corresponding TPD file, with a different extension (**.tpc**), and resides in the same directory.

The TPC file is immediately updated every time the configuration changes. No tests are executed until this information is complete. If a test object needs access to this data it can easily retrieve the information with the TEXDLL function, texGetTPC. To better support test object access, static configuration information is now written to the TPC file. The TPC file is always in the parent directory to the test object.

The [**Recover**] section of the TPC file contains the current UUT MSN and operator ID. This section is only present while the Test Executive is active. It is primarily used to determine if a power failure or system crash occurred during testing. If a crash is detected, the Test Executive prompts the user and offers to recover the test status for the most recent UUT. If a corresponding database is located, the database is automatically repaired before the test status is read and displayed. To manually compress/repair a database, use the **JetComp** utility available from Microsoft. The Recover section is also a convenient place for a test object to retrieve the UUT MSN.

Example

```
[CONFIGURATION]
TestType = Acceptance
UUTPartNumber = 2
UUTRevision = A
WorkOrderNumber = 33
WorkOrderOperation = 9876543
TestChamberTemp = 444
InterfaceAdapterPartNumber = 6578974-1492
InterfaceAdapterRevision = X
InterfaceAdapterMSN = 6663466-1776
TestSetRevision = YY
TestSetMSN = 4545456

[RECOVER]
OperatorID = Biff Lightsnack
UUTMSN = Sector 12
```

6.8 Test Executive Configuration File

The Test Executive Configuration File (TestExec32.cfg) defines operating characteristics for the Test Executive. This file is created or modified to match the requirements of a specific installation of the Test Executive. Within the file are defined test station information, security and an optional test program list.

The test station information section is used to identify the part number, revision and serial number of the physical test system. This provides a common place for this information to reside and allows it to be used by any test programs that are executed on the system. The information defined in this section has priority over information contained in a TPD file.

The security section specifies the operator privileges database that will be used by the Test Executive. If undefined, the Test Executive prompts to either create one or to disable the option. This typically occurs on the Test Executive's initial execution following installation.

> **Note: If operator privileges are required, the configuration file and privileges database should be write-protected to avoid unauthorized modifications.**

The optional test program list section has a list of test programs that are available for loading by the Test Executive. The list is presented to the operator, in the Test Program List window, as descriptive phrases or sentences. This allows the operator to load a test program by a meaningful name, or title, rather than by navigating a file dialog to a specific TPD file.

Within the configuration file, double slashes (***//***) are used to denote comments. This allows additional information to be included in the file for documentation and ease of maintenance. Commented lines are also useful during development for temporarily disabling portions of the file.

6.8.1 Test Station

The Test Station section, of the Test Executive Configuration file, defines information about the physical system that the Test Executive is operating on. The entries have the same format as the Configuration section of a TPD file. Static, default, list or formatted values can be defined for the items in this section. Values specified here take precedence over the corresponding **Test Set** values in a TPD file. When operating with an Environmental Test Manager, this section may conflict with the ETM's configuration file.

Keyword	Description
[TestStation]	Section header for test station information.
PartNumber =	Test Station part number
Revision =	Test Station revision
MSN =	Test Station serial number
TestLocation =	Test Station location
Custom1 =	Optional, user-defined entry

The format of an entry is:

keyword = *default* [; *format* [; *MinimumSize*]]

Field	Description
default	Default, static (<1234-5>), list ([A, B, C]) or disabled (N/A)
format	Masked edit definition
MinimumSize	Required minimum length of entered data

The information defined in this section is typically common for all tests executed on an individual test system. Consequently, it is advantageous to define the information in one Test Executive Configuration file, rather than in multiple TPD files or, worse, relying on an operator to enter it manually.

Example

```
[TestStation]
PartNumber = <110287-46>
Revision = C;??;1
MSN = 1101
TestLocation = <Santa Fe, NM>
```

6.8.2 Options

The Options section, of the Test Executive Configuration file, defines overall information about how the Test Executive should behave on the current test system. The entries have a format similar to the Options section of a TPD file. Included in this section are keywords to control the PDL logging format, set log file management and enable unique database naming.

Keyword	Description
[Options]	Section header for test station options.
DataLogType =	Sets system PDL log type (PDEL or IEEE-1545). The default is IEEE-1545. This can be overridden by individual TPD files.
PdelFileLimit =	Controls log file management of PDL files.
DataFileLimit =	Controls log file management of database files.
UniqueDatabaseName =	Create database files with unique names (Y/N), default is N.

The format of an entry is:

keyword = *string*

The management of PDL and database files is now externally controllable. The next section describes the capabilities of these entries.

By default, database file names match a UUT's serial number. This can present a problem if databases for different UUTs are gathered into a single directory space. Therefore, an option is provided to create a more unique name for each database file. When this option is active, database file names are created from a UUT's part number, revision and serial number.

Example

```
[Options]
DataLogType = PDEL

// Automatically recycle PDL files when 500, or more,
// are 30 days old (or older).
PdelFileLimit = 500; Recycle; No Prompt; 30

UniqueDatabaseName = Y   // Use PN_REV_SN for database names
```

6.8.3 Log File Management

The management of PDL and database files is now controllable through the [Options] section of the Test Executive Configuration file. By default, the Test Executive prompts the operator to delete files when there are 1000 PDL files or 50 database files. Now the file number, removal operation, prompt and file age are defined by the following statements:

PdelFileLimit = *Number of Files* [; *Removal Type* [; *Prompt Option* [; *File Age*]]]
DataFileLimit = *Number of Files* [; *Removal Type* [; *Prompt Option* [; *File Age*]]]

Field	Description
Number of Files	How many files must be present before an action occurs. Set to -1 to disable file management.
Removal Type	How are the files to be removed: Delete, Recycle, Never. Default is delete.
Prompt Option	Is the operator to be notified before removal (Prompt or No Prompt)? Default is prompt.
File Age	How many days old should the files be before being removed. Default is immediately.

Given these options, there are several different ways to implement log file management. One way is to allow the operator to automatically delete the files once they have exceeded their specified limit (similar to the default behavior):

```
DataFileLimit = 50
```

Another way is to prompt the operator that the files have exceeded a certain number and he informs a supervisor that the files need to be archived (i.e. the operator isn't given the option to remove them):

```
PdelFileLimit = 1275; Never
```

Alternately, the system could automatically recycle files that are older than 60 days, without informing the operator at all:

```
DataFileLimit = 40; Recycle; No Prompt; 60
```

An operator without **Log** privileges is never prompted to delete log files. He is prompted if the Test Executive is configured to notify without removal.

6.8.4 Security

The Security section, of the Test Executive Configuration file, defines the database file that contains operator names, passwords and privileges. Set the privilege file to "N/A" in order to disable security checking. If left undefined, the Test Executive offers to create a default one. The Privilege Editor defaults to loading the file defined here.

Keyword	Description
[Security]	Section header for security information.
PrivilegeFile =	Name and path of privilege database file
UsePassword =	Assign internal password when a new privilege database is created (Y/N), default is N.

When the Privilege Editor creates a new privilege database, it looks at the **UsePassword** entry to determine whether to lock the file with an internal password. A password prevents someone from reading, or editing, the database using conventional tools such as Microsoft Access. In addition, new privilege databases are encrypted to prevent viewing by low-level file tools.

Example

```
[Security]
PrivilegeFile = C:\SysSecure\Privileges.mdb
UsePassword = Y
```

6.8.5 Test Program List

The Test Program List section, of the Test Executive Configuration file, defines a list of test programs that are available for loading by the Test Executive. The list contains TPD file names matched to descriptive phrases or titles. The descriptions are presented to the operator in the Test Program List window. Note that some operators, with restricted privileges, may be able to only load TPD files from this list.

Keyword	Description
[TestProgramList]	Section header for test program list
HomeDirectory =	Common directory path for TPD files
Tpd =	Test program description and file name

The home directory entry is used to simplify the TPD entries and to make the configuration file more portable between test systems. It allows a common, or root, directory to be defined for the TPD files. The TPD files are then defined with relative paths to the common directory. If a TPD file is defined with a full path, the home directory is ignored for that file.

The TPD entries consist of a descriptive string and a file name, separated by a semicolon. Entries are used for labels and spacing by not including a file name. Indentation of the descriptions is possible by using a TAB designator (**\t**). TPD files are defined with full directory paths or with paths that are relative to a designated home directory. File paths and names should be enclosed in double quotes (**"**).

Note: TPD files in this list are not verified to exist until they are loaded.

Example

```
[TestProgramList]
// TPD paths are defined relative to this directory.
HomeDirectory = "C:\Program Files\SSI Test Executive"

// TPD files have a relative or full directory path.
Tpd=\t;
Tpd= ========== F-16 Navigation Unit ==========;
Tpd=\tPositioning System 456TW ;"\Tpd\Txdemo32.tpd"
Tpd=\tInertial Guidance 453E;"\Tpd\TxOptn32.tpd"
Tpd=\tPilot Display 5689-3B;"\Tpd\TxOptn32.tpd"
Tpd=\t;
Tpd= ========== F-16 Fire Control ==========;
Tpd=\tFull System Diagnostics;"C:\Tpd456\Txdemo32.tpd"
Tpd=\t\tAcquisition Subsystem 345-90;"\Tpd\Txdemo32.tpd"
Tpd=\t\tArming Interface 12123C;"\Tpd\Txdemo32.tpd"
Tpd=\t\t\tPhaser Lock 1723;"\Tpd\Txdemo32.tpd"
```

6.9 Microsoft Access Database File

When the database option is enabled in the Test Executive, a Microsoft Access database (Jet v3.51) is created to store the test information for each UUT MSN. The Report Generator uses these databases to view and print datasheets. The Test Executive creates a database for a new UUT by copying an empty database file (TEXDB351.MDB). The following is mainly provided for information only. The only reason for a developer to be concerned with the internal database structure is if she is creating a custom report printer. Note that an "Out of Memory" error has been corrected by using Jet v3.51. See *Appendix C - Database "Out of Memory" Error* for more information. A **JetComp** utility is available from Microsoft for compressing and repairing a database. A Microsoft Access database file can be up to 2GB in size. The database tables are designed so data from multiple UUTs can be merged together. The database has four tables:

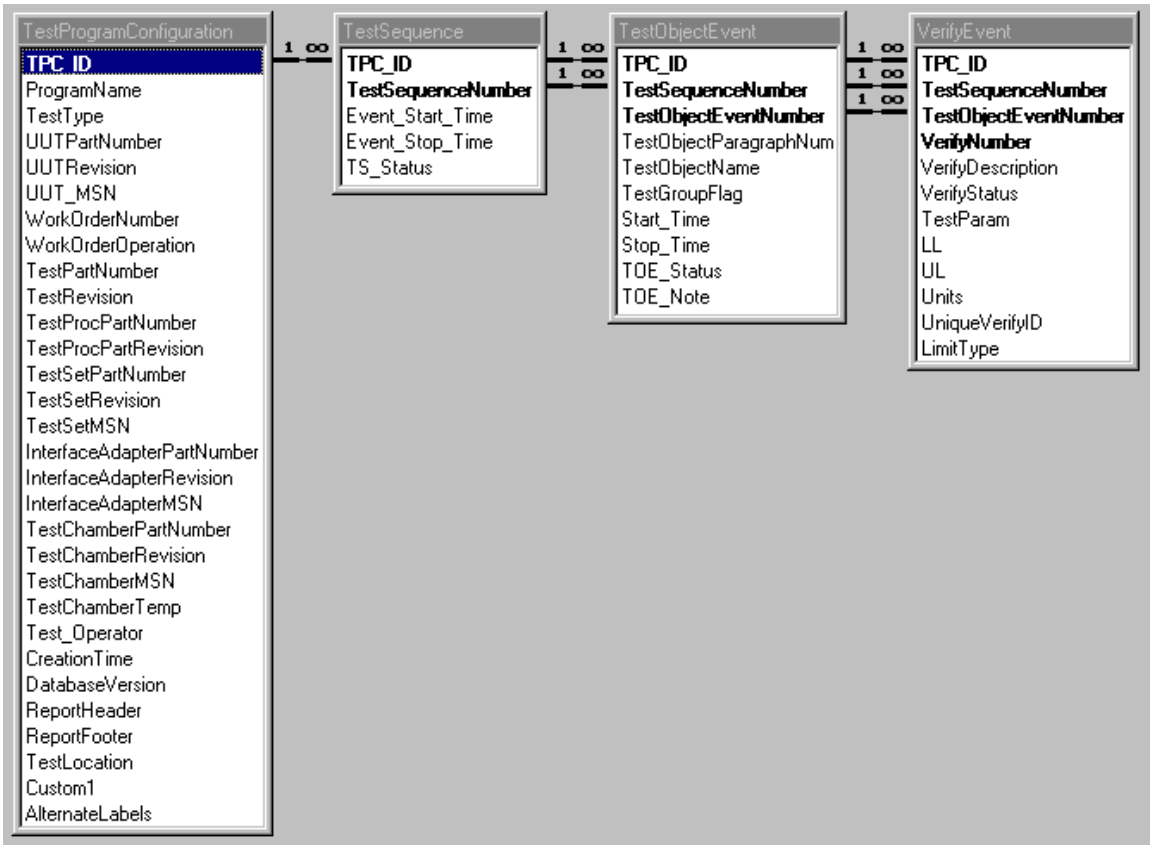
1. The **TestProgramConfiguration** table contains the information needed to create the header section of a datasheet.

2. The **TestSequence** table contains the information about the execution of a sequence of test object events.

3. The **TestObjectEvent** table contains the information describing a test object, the time the test object was executed, and the result of the execution.

4. The **VerifyEvent** table contains the information describing the verifies associated with a test object event.

The relationships between the tables are shown in the following diagram:



6.9.1 Test Program Configuration Table

The Test Program Configuration table holds the information entered in the Test Configuration dialog. This represents a snapshot of conditions when the associated tests are performed. If any of this information changes (e.g. a different operator), a new record is created in the database.

The **TestProgramConfiguration** table contains the following fields:

Field Name	Type	Size	
TPC_ID	Long		// Primary key
ProgramName	Text	40	
TestType	Text	16	
UUTPartNumber	Text	24	
UUTRevision	Text	12	
UUT_MSN	Text	24	
WorkOrderNumber	Text	24	
WorkOrderOperation	Text	24	
TestPartNumber	Text	24	
TestRevision	Text	12	
TestProcPartNumber	Text	24	
TestProcRevision	Text	12	
TestSetPartNumber	Text	24	
TestSetRevision	Text	12	
TestSetMSN	Text	24	
InterfaceAdapterPartNumber	Text	24	
InterfaceAdapterRevision	Text	12	
InterfaceAdapterMSN	Text	24	
TestChamberPartNumber	Text	24	
TestChamberRevision	Text	12	
TestChamberMSN	Text	24	
TestChamberTemp	Text	8	
Test_Operator	Text	24	
CreationTime	Date		
DatabaseVersion	Text	8	
ReportHeader	Text	255	
ReportFooter	Text	255	
TestLocation	Text	255	
Custom1	Text	255	Custom field for user-designated data.
AlternateLabels	Memo		List of alternate labels for configuration data.

6.9.2 Test Sequence Table

A Test Sequence table is created each time the Test Executive Run button is pressed. This represents a sequence of one or more test objects being executed. Consequently, this table is associated with one or more Test Object Event records.

The **TestSequence** table contains the following fields:

Field Name	Type	Size	
TPC_ID	Long		// Primary key
TestSequenceNumber	Long		// Primary key
Event_Start_Time	Date		
Event_Stop_Time	Date		
TS_Status	Text	5	Pass, Fail, Abort, Error

6.9.3 Test Object Event Table

A Test Object Event table is created each time the Test Executive executes a test object. Information about the test object is stored in this table as well as a link to its associated Test Sequence record. One or more Verify Event records are associated with each Test Object Event.

The **TestObjectEvent** table contains the following fields:

Field Name	Type	Size	
TPC_ID	Long		// Primary key
TestSequenceNumber	Long		// Primary key
TestObjectEventNumber	Long		// Primary key
TestObjectParagraphNum	Text	12	
TestGroupFlag	Integer		
TestObjectName	Text	40	
Start_Time	Date		
Stop_Time	Date		
TOE_Status	Text	5	Pass, Fail, Abort, Error, Group
TOE_Note	Text	128	

6.9.4 Verify Event Table

A Verify Event table is created each time a verify is executed by a test object. Information about the verify is stored in this table, as well as a link to its associated Test Object Event record.

The **VerifyEvent** table contains the following fields:

Field Name	Type	Size	
TPC_ID	Long		// Primary key
TestSequenceNumber	Long		// Primary key
TestObjectEventNumber	Long		// Primary key
VerifyNumber	Long		// Primary key
VerifyDescription	Text	40	
Units	Text	32	
TestParam	Text	10	
VerifyStatus	Text	5	Pass, Fail, Abort, Error
LL	Text	10	
UL	Text	10	
UniqueVerifyID	Text	40	
LimitType	Text	5	

6.10 Parametric Data Log File (IEEE-1545)

The Test Executive supports two versions of parametric data logging to ASCII files. The original format, Parametric Data Exchange Language (PDEL), has been adopted as an IEEE standard (IEEE-1545) and renamed to Parametric Data Log (PDL) format. During the standardization process, some changes were made to the syntax and content portions of the specification. PDEL support is being continued for backwards compatibility, but new test applications are strongly encouraged to use the IEEE-1545 format for data logging.

The default logging format for the Test Executive is IEEE-1545. The logging format can be defined in the Test Executive Configuration file for a test system, or in a TPD file for an individual test program. The active logging type is displayed on the Test Executive Options Menu and status bar (PDEL or PDL).

The other noticeable display difference is that IEEE-1545 uses “Serial Number” (S/N) rather than PDEL’s “Manufacturer’s Serial Number” (MSN). Corresponding display and print labels reflect this difference. Internal labels and keywords for database logging and TPD/TPC files are unchanged (i.e. they still use MSN). Also, IEEE-1545 introduced a new data field (Test Location) that is now supported on the Test Configuration dialog and in the TPD file.

> **Note: IEEE-1545 support includes more rigorous filtering of header names and verify identifiers. The IEEE-1545 specification excludes the use of identifiers that match any of the defined reserved words. Historically, PDEL generation has not been as rigorous about reserved words. The filtering process modifies an identifier until it no longer matches a reserved word or contains invalid symbols.**

For more information on the IEEE-1545 specification, access the IEEE's publication web site (http://standards.ieee.org/).

7. Programming Interface

7.1 Test Object Programming Interface

The TEXDLL is a Windows dynamic-link library (DLL) used to verify measurements, insert breakpoints, and provide a set of user interface functions for the test object. It supports two parallel modes of operation, depending upon whether a test object is invoked by the Test Executive or not. When a test object is invoked by the Test Executive, the TEXDLL automatically sends all test results to the Test Executive for processing. Otherwise, when run stand-alone, the TEXDLL automatically writes all test results to a log file (TestExec.log) for review by the user. This stand-alone mode allows test object development without requiring an active Test Executive.

The architecture of the Windows environment allows a DLL to be called from a wide range of languages and development environments. Sample projects, and application notes, are included in a SAMPLES subdirectory to demonstrate how to call the TEXDLL from the following environments:

- National Instruments LabWindows/CVI.
- Microsoft Visual Basic
- Microsoft Visual C++
- Borland C++
- National Instruments LabVIEW
- Geotest ATEasy

Several files are also included with the Test Executive to support these various development environments. Function definitions are available for C in TEXDLL32.H and for Visual Basic in TEXDEF32.BAS. A function panel file, TEXDLL32.FP, is provided for National Instruments LabWindows/CVI. A LabVIEW library (TexDll32.llb) is provided to simplify the use of TEXDLL functions in a LabVIEW block diagram. An import library, TEXDLL32.LIB, is also available to allow linking with C programs. Import libraries reside in several sample project directories because their format varies for different compilers. Additionally, most development environments can build an import library from an existing DLL.

Since the 32-bit version of National Instruments LabWindows/CVI supports several different C compilers, there are several import libraries included in subdirectories to its sample project directory. You can also easily build the necessary import library with CVI.

A secondary DLL (TEXINPUT.DLL) is used by TEXDLL for several user interface functions.

Note: See section *Test Objects and Microsoft Windows* for more information about operating system requirements.

Note that programs written in C++ (*.CPP) are compiled with “mangled” function names that include parameter count information. Test developers must keep DLL function names from being mangled. Do this by using the keyword **extern “C”** to bracket the TEXDLL include file:

```
extern “C” {
#include “texdll32.h”
}
```

7.2 TEXDLL Summary

The TEXDLL programming interface consists of five function groups. Access functions are used to delimit the beginning and end of a test; verify/compare functions evaluate values measured during a test; and user interface functions provide a consistent way to report messages, warnings, and errors. A collection of other functions handles various logging, scaling and program control options.

Access Functions

texStart	Required to signal the beginning of a test to the TEXDLL.
texFinish	Required to signal the end of a test to the TEXDLL.

Verify/Compare Functions

texCompare	Returns the pass/fail result of evaluating a parametric test value. Test criteria is provided by the test object.
texCompareTOD	Returns the pass/fail result of evaluating a parametric test value. Test criteria is provided from a TOD input file.
texVerify	Returns the pass/fail result of evaluating a parametric test value; and reports a verify event to the TestExec log file. Test criteria is provided by the test object.
texVerifyTOD	Returns the pass/fail result of evaluating a parametric test value; and reports a verify event to the TestExec log file. Test criteria is provided from a TOD input file.

User Interface Functions

texErrorBox	An error is shown in a dialog box and written to the TestExec log file.
texInputNumber	Prompts operator for numerical data entry.
texInputString	Prompts operator for text data entry.
texMessageBox	A message is shown in a dialog box and the TestExec log file.
texPrint	A print message is added to the TestExec log file.
texWarningBox	A warning is shown in a dialog box and the TestExec log file.
texYesNoBox	A message is shown in a YesNo box and the TestExec log file.

Information Functions (optional)

texGetTODVerify	Retrieves verify parameters from the current Test Object Definition file.
texGetTPC	Retrieves configuration data from the current Test Program Configuration file.
texGetTPD	Returns items extracted from the current Test Program Definition file.

Other Functions

texAbort	Used by an external application to abort a test object.
texEmbedBreakpoint	Creates a potential breakpoint in the test object code.
texLogPdel	A PDEL message is added to the PDEL file.
texPowerMonitor	Tracks UUT power state (e.g. on or off)
texScale	This function scales a given value from one prefix magnitude to another.
texSetAbortDelay	This function sets the time (in seconds) that a test object is given to close itself after an abort is requested. If control returns to the TEXDLL and this time has expired, the test object is terminated.
texStartActive	This function returns True if texStart() has already been called.

7.3 Access Functions

Each execution of a test or setup object should signal the beginning and end of a test to the TEXDLL by making one call to texStart() and texFinish(), respectively. These functions grant the calling application TEXDLL access for the duration of a test and release the TEXDLL when access is no longer required.

A test object must make one call to texStart() to signal the beginning of a test. This grants TEXDLL access to the calling application and initializes DLL resources for the start of a new test. When a new test is initialized, the default TEXDLL behavior is reinstated, the behavior specified in the texStart() command string is applied, the verify count is reset to zero, and a new TestExec log file is started. If this step is omitted, the DLL remains uninitialized and all other functions report an error.

While a test is running, the TEXDLL automatically records test object activity by making entries in a TestExec log file. If the test object is invoked by the Test Executive, log messages are sent to the Test Executive; otherwise they are redirected to a log file named (TESTEXEC.LOG).

A test object must make one call to texFinish() to signal the end of a test. The TEXDLL must be told explicitly when a test has completed legally with a valid pass/fail result. This is done by passing a status value (NORMAL_ FINISH) to the texFinish() function. Otherwise, no pass/fail result is recorded; since the TEXDLL must assume that some problem prevented the test object from reporting successful completion of the test. Another status value, RELOAD_FINISH, causes the Test Executive to reload the current TPD. This can be used to recover following an abort, or to load a customized TPD for the current UUT.

The TEXDLL grants access to one test object at a time. However, it always stops what it's doing and tries to grant access to the newest calling application. The following steps occur if a second test calls texStart() while a test is currently running:

1. The current test is aborted.
2. A warning is reported.
3. The second test is initialized and started.

This way, the TEXDLL cannot be locked up by a test object that fails to call texFinish().

In some circumstances it is valuable for an external tool or modular component to know whether texStart() has occurred. A function, texStartActive(), is available to determine this. If a test object is currently accessing the TEXDLL (i.e. has called texStart()), texStartActive() returns a nonzero value.

7.3.1 Behavior Modes

All TEXDLL behavior is controlled by passing a string of command flags to texStart() at the beginning of a test. This includes modal behavior regarding verify breakpoints, data breakpoints, trace breakpoints, and warnings. A TOD input file and a help file are also specified using command flags.

The intended source of the texStart() control string is the test object's command line. This allows TEXDLL behavior to be externally controlled by a test object's command line arguments. This permits the Test Executive to control breakpoints, warnings, TOD and help files when the test object is executed. It also permits control of TEXDLL behavior during program debug, since most development environments allow command line arguments to be set.

The standard for many Windows programs is to receive the command line string as an input parameter (e.g. via WinMain) or as an accessible variable (Command$ in Visual Basic). With these, all that is required is to reference the string in texStart(). Conventional C programs receive command line arguments as an array of strings. Under these circumstances, a command line string must be built for passing to texStart(). The following sample code shows how this can be easily accomplished:

```
#include <stdio.h>
#include <string.h>
extern "C" {  #include "texdll32.h"  };

#define CMDBUFSIZE 512

void main(int argc,char *argv[])
{
        int i;
        char CmdBuf[CMDBUFSIZE];

        /* build string containing space-delimited command line arguments */
        CmdBuf[0] = '\0';
        for ( i = 1; i < argc; i++ )
                {
                strcat(CmdBuf,argv[i]);
                strcat(CmdBuf," ");
                }
        texStart(CmdBuf);                   /* pass command line to DLL */
        texMessageBox ("Hello World!");
        texFinish (NORMAL_FINISH);
}
```

Note: This sample code may encounter a problem when running on Windows 95, or Windows NT, since they both allow spaces in file and directory names. If the path or name of a TOD file, or help file, contains two consecutive spaces, the above code will not be able to reassemble a valid file reference.

When the Test Executive invokes a test object, it also receives the test results produced by the TEXDLL. The test results are processed, and filtered, by the Test Executive and displayed in the Test Results window. A test object can use the TEXDLL without being invoked by the Test Executive. This stand-alone mode is typically used during test object development. When run stand-alone, the test object specifies TEXDLL behavior through the command line, and the test results are written to a file (TESTEXEC.LOG) that is created in the same directory as the test object.

7.3.2 Command Flags

The TEXDLL recognizes the following flags found in the command string passed to the texStart() function. If conflicting flags appear in the same command string, the **leftmost** flag overrides all conflicting flags (as of Test Executive v6.2). If no flags are specified, TEXDLL behavior is initialized using the default modes (marked below with an asterisk). Command flags intended for a test object should be preceded by "**/usr**" (e.g. /usrInitA) to avoid conflicting with flags intended for the TEXDLL. The Text Executive also generates command flags that are only intended for a test object to use such as repeat count ("**/rpt**") and sequence results (e.g. **/SeqFail**) for post sequence setup objects (see *Test Object Command Line Arguments*).

Flag	Behavior
Verify Breakpoints	
"/va"	Break at all verifies.
"/vf"	Break at failing verifies.
"/v" *number* / **'***id***'**	Break at verify *number* or *id* (e.g. "/v 25" or "/v **'**vfy5**'** ")
"/vn"	Disable verify breakpoints. *
Data Breakpoints	
"/bd"	Enable dialogs and log entries for all data breakpoints.
"/lbd"	Only enable log entries for all data breakpoints.
"/nbd"	Disable data breakpoints. *
Trace Breakpoints	
"/bt"	Enable dialogs & log entries for all trace breakpoints.
"/lbt"	Only enable log entries for all trace breakpoints.
"/nbt"	Disable trace breakpoints. *
Warnings	
"/w"	Enable dialogs and log entries for all warnings.
"/lw"	Only enable log entries for all warnings.
"/nw"	Disable warnings. *
File Names	
"/tod" *filename*	Specify TOD filename (e.g. "/tod uut52319.tod").
"/tpd" *filename*	Specify TPD filename (e.g. "/tpd texdemo.tpd").
"/hlp" *filename*	Specify Help filename (e.g. "/hlp sample.hlp").
Test Id	
"/tid" *paragraph*	Specify current test entry paragraph number.
Abort Delay	
"/ad" *delay*	Specify abort delay in seconds (default is 1s)

* Denotes a default behavior mode.

7.3.3 TPD Reload

One of the status values supported by texFinish() causes the current TPD to be reloaded (RELOAD_FINISH). This can be used as a way to maximize recovery from an abort, or as a way to adapt testing for a UUT-specific configuration. These options are both discussed in the following sections.

7.3.3.1 Abort Recovery

Ideally, each test object is carefully designed and built with reliable, integrated abort handling procedures. In reality, abort handling is sometimes a last minute consideration during test system integration, or, worse, after initial delivery to production. Under these circumstances, reloading the TPD, following an abort, may be a sufficient way to recover from an unhandled abort. **Note that this is not intended as a substitute for properly handling abort requests**.

One of the biggest challenges for abort handling in a test object is the release of allocated resources such as memory, open files and instrument handles. Unreleased resources can result in memory leaks, corrupt files and 'locked' instruments. The Windows operating system does automatically release many resources when an execution process is ended. This is not helpful for test object DLLs, because they execute within a shared process (TexDLLControl) that remains active (thus continuing to 'own' the resources). Therefore, halting the TexDLLControl process might release many of the resources previously allocated by the aborted test object.

Unfortunately, this is complicated by the fact that the TexDLLControl process is tasked with maintaining preloaded DLLs in memory during testing. These DLLs are preloaded, as defined by the TPD, in order to speed test execution or to hold state data (e.g. instrument handles). Therefore, the TexDLLControl process cannot simply be restarted, it must also be initialized as it was when the TPD was originally loaded. Thus, the best way to close, restart and initialize the TexDLLControl process is to completely reload the TPD.

> **Note: When the TexDLLController is restarted, any information stored by loaded DLLs is lost. Consequently, using this for abort recovery requires that preloaded DLLs either do not store intermediate test results/status, or that those results are placed in a file or other non-dynamic storage location. Alternately, judicious use of immediate prerequisites can guarantee that shared data is generated by one test object before being used by another.**

Reloading the TPD on abort is easily achieved with a simple AbortTest setup object. This is run after the test object's execution has been halted. As a minimum, all the abort object would need to do is issue a RELOAD_FINISH:

```
void main()
{
        texStart("");
        texFinish(RELOAD_FINISH);
}
```

Of course, the abort object is also an ideal place to perform abort-related cleanup and recovery tasks. These tasks might deal with non-resource items that the operating system is not going to handle. This could include deleting temporary files or resetting instruments.

Because of the complex nature of aborts, and TPD loading, it is highly recommended that you review the *Detailed Behavior* section for TPD reloading.

7.3.3.2 UUT-Adaptation

Another use for the TPD reload mechanism is to dynamically modify the test set based on a particular UUT's configuration. For example, a UUT at revision level A might require different tests, or limits, than one at revision level C. As a more extreme case, a UUT might have optional subsections, or components, that can be ignored if they are not populated. Given these situations, it is often useful if the TPD only displays and executes tests that match the UUT's exact capability.

To achieve this, a setup object (PreMSN or PreSequence) can review the UUT's configuration (i.e. serial number, revision, work order, etc.), using texGetTPC(), and modify, or swap, the TPD to match. A subsequent RELOAD_FINISH, by the setup object, loads the suitably modified TPD.

If there are only a few permutations of the TPD file (e.g. for revisions A, B and C), it is probably easiest to have three different TPD files and copy the applicable one over the currently loaded one. For example, a TPD is loaded (UUT_123.tpd) and the operator indicates that a revision B UUT is being tested. The PreMSN setup object copies the revision B TPD (UUT_123B.tpd) over the current TPD (UUT_123.tpd) and issues a RELOAD_FINISH. A file copy can be achieved, in most development languages, by a few lines of code or a single Windows API call (CopyFile).

> **Note, the setup object that swaps, or modifies, the TPD should first check to see whether it is already configured correctly. This will help to avoid unnecessary reloads. The currently loaded TPD can be quickly accessed with the texGetTPD() function.**

If there are many possible permutations for customizing a test set to a specific UUT, then it might be better to dynamically modify a single TPD. Some sections of the TPD have unique keywords, thus they can be easily changed using the WritePrivateProfile operations available from the Windows API. The other parts of a TPD file are easily read and written line-by-line. Ideally, the code that modifies the TPD should only be including or excluding entries that are already defined in the file. For example, the test hierarchy could be read line-by-line and unnecessary lines 'removed' by comments (see "Down Link B" below):

```
[TestObjects]
Test = 1.;Full Run
Test = 1.1.;Subsystem A32-Q;;;;txdemo32.exe
Test = 1.2.;Up Link;;;;txdemo32.exe;/f
Test = 1.3.;Down Link A;;;;txdemo32.exe
// Test = 1.4.;Down Link B;;;;txdemo32.exe
Test = 1.5.;Norden Sight Alignment;;;;txdemo32.exe
```

This approach keeps the test functionality completely defined within the TPD, not buried in setup object code. Thus, TPD modifications and tracking are easier. A slightly different approach would be to have a master TPD that incorporates all possible tests for a UUT. A customized TPD can then be built by just copying the applicable entries to another file.

Because of the complex nature of setup objects, and TPD loading, it is highly recommended that you review the *Detailed Behavior* section for TPD reloading.

7.3.3.3 Detailed Behavior

Reloading a TPD involves a complicated sequence of events. Since there can be interdependencies, or side effects, between operations, the exact reload process is detailed below.

1. A test or setup object issues RELOAD_FINISH via texFinish()
2. The test or setup object completes execution
3. If a sequence is active, the PostSequence setup objects are executed
4. The TPD is "unloaded"
   a) PostLoad setup objects are run
   b) TexDLLController is shutdown
5. The TPD is reloaded
   a) TexDLLController is restarted
   b) PreLoad setup objects are run
   c) Test Program Configuration (TPC) data is restored
   d) User interface settings are restored (breakpoints, logging, etc)
   e) PreMSN setup objects are run
   f) Reassert Power-on settings
   g) Displayed test status is restored (along with internal verify data)

Power-on tracking is restored to its previous state following a reload. Thus, if UUT power is on when a reload is requested, it is assumed to still be on after the reload occurs. This is the safest way to ensure a powered-on UUT is not overlooked during reload (PostLoad/PreLoad power monitor commands are ignored). Therefore, to guarantee that the Test Executive considers UUT power to be off, following a reload, issue a texPowerMonitor() command prior to the RELOAD_FINISH.

RELOAD_FINISH is **not** supported for PostLoad setup objects. Reloading a TPD, while it is being unloaded, has little value and could easily become an endless loop. Similarly, a reload request is ignored when the Test Executive is shutting down. The reload is also restricted to a **single** iteration. That is, if a PreLoad object initiates a reload, a second reload request, during the initial reload, is ignored. If this situation occurs, detailed information is written to the Test Result window and Debug Log window/file.

7.4 Verify/Compare Functions

There are four verify/compare functions in the TEXDLL programming interface. All four functions return a pass/fail status based on evaluating a measured test value with respect to a test criteria. All four use identical evaluation methods and test criteria parameters; so that each returns an identical status if given the same input.

One important distinction between these functions is that two perform a comparison; and two perform a verify. Another distinction is that two functions define test criteria using input arguments; and two use test criteria read from a TOD file.

Verify/Compare Functions

texCompare	Returns the pass/fail result of evaluating a parametric test value. Test criteria is provided by the test object.
texCompareTOD	Returns the pass/fail result of evaluating a parametric test value. Test criteria is provided from a TOD input file.
texVerify	Returns the pass/fail result of evaluating a parametric test value; and reports a verify event to the TestExec log file. Test criteria is provided by the test object.
texVerifyTOD	Returns the pass/fail result of evaluating a parametric test value; and reports a verify event to the TestExec log file. Test criteria is provided from a TOD input file.

The texCompare() and texVerify() functions accept a list of input arguments that define the test criteria. Fundamental data types are used, rather than structures, for all input arguments. This allows the functions to be called by a wide range of languages and tools.

The texCompareTOD() and texVerifyTOD() functions use test criteria parameters read from an input TOD file specified in the command string, (e.g. "/tod sample.tod"). Verify data in the TOD file is 'looked up' using the unique identifier field. A unique identifier is specified as an input parameter to both of these functions; this value is matched with a corresponding identifier field from one verify line in the TOD file. This way, TOD file entries can be listed in any order; and multiple test objects may reference the same TOD file. If a matching identifier field cannot be found in the TOD file, an error is raised and the function call fails.

Each time a verify function is called, the verify count is incremented in the TEXDLL and the verify and its result are reported to the TestExec log file. Each verify has the potential to invoke a verify breakpoint, based on the current TEXDLL behavior; and verify results determine the outcome of a test. After the normal conclusion of a test, all verifies must pass for the test to pass. Each test object can perform up to 1000 verifies.

Compare functions behave like a passive, off-line form of verify. Any number of compares can be made at any point during a test without affecting the verify behavior. Compares do not increment the verify count or invoke verify breakpoints. Compares do not influence the outcome of a test; and they are not recorded in the TestExec log file.

Verify parameters in a TOD file are retrieved by texGetTODVerify(). This function reads and returns verify parameters associated with a unique verify id. This information can be used for additional logging, scaling, or to modify before using with texVerify(). For example, a test object might want to append a timestamp to a description parameter before it is verified (i.e. logged). This is easily accomplished by adding a small utility function within the test object code.

Note that verify commands can also be used to display, and log, additional information about test conditions. For example, if a string compare fails, a follow-up verify could include the failing string. Or, a series of verifies could be used to log instrument serial numbers and calibration dates (if available).

7.4.1 Verify Breakpoints

Four modes of verify-based breakpoint behavior are available. The TEXDLL verify breakpoint behavior is set by including one of the following command flags in the command string passed to texStart(). By default, verify breakpoints are disabled:

Flag	**Behavior**
Verify Breakpoints	
"/va"	Break at all verifies.
"/vf"	Break at failing verifies.
"/v" *number* / '*id*'	Break at verify *number* or *id* (e.g. "/v 25" or "/v 'vfy5' ")
"/vn"	Disable verify breakpoints (default).

A breakpoint on a specific verify operation is achieved by specifying the verify number, incremented sequentially from the beginning of the program, or the verifies' unique identifier (ID) string. Using the unique ID makes it easier to isolate an individual verify during the execution of several test objects. Note that when specifying a verifies' unique ID in a command string, the ID must be enclosed in single quote characters (').

When a verify breakpoint box appears, it has an OK button and optionally a Help button. The Help button only appears if a help file has been specified in the TPD file. If the Help button is pressed, the help topic displayed corresponds to the current verify. This is achieved by using the unique verify id as one of the keywords for the help topic.

Note: Verify breakpoints halt execution until an operator responds. Consequently, a verify function should not be called during time-critical measurements or when hazardous conditions are active. It is better that the measurement be stored and verified when time permits. Verify breakpoints can be disabled for a test object through command line arguments, but this disallows them for debugging and it is a weak protection mechanism.

7.4.2 Test Criteria

Verify/compare test criteria is defined by a set of eight parameters:

Parameter	Type	Description
description	string	A descriptive, user-friendly string. Up to 40 characters. Empty strings or NULL pointers raise a warning.
id	string	A unique string identifier. Up to 40 characters. This string can only contain alphanumeric characters and underscores. Illegal characters, empty strings or NULL pointers cause the verify to fail and are reported as errors.
limitType	short	Determines what test method is used. Legal constant values for this parameter are defined in the file (TEXDLL32.H).
limit1	double	The usage depends upon *limitType* (see *Test Methods*).
limit2	double	The usage depends upon *limitType* (see *Test Methods*).
sigDigits	short	Number of significant digits applied to the test criteria parameters. Legal values for this parameter are zero through six.
prefix	string	The order of magnitude for the units of measurement (e.g. “milli”). Up to eight characters. Empty strings and NULL pointers are allowed (see *Appendix B - Units and Prefixes).*
units	string	The units of measurement for the compare (e.g. “volts”). A value of “hex” displays the compare using hexadecimal format. Up to 20 characters. Empty strings and NULL pointers are allowed (see *Appendix B - Units and Prefixes)*.

7.4.3 Invalid Test Criteria

To help identify problems during test object development, the TEXDLL identifies and warns the user about the following inappropriate test criteria whenever a compare or a verify is made.

Condition	Recovery Behavior
No description string.	A default string is substituted.
Description too long.	Description is truncated at the maximum length.
Id too long.	Id is truncated at the maximum length.
Too many verifies.	Verify function returns immediately.
Negative significant digits.	Minimum value (0) is used.
Too many significant digits.	Maximum value (6) is used.
Empty units or prefix.	Default string “None” is substituted.
Specified prefix with no units	Ignore prefix. Default string “None” is substituted.

The following conditions raise an error because there is no possible recovery behavior.

Condition	Error Behavior
No id string.	Reports an error, and the function always fails.
Illegal characters in id.	Reports an error, and the function always fails.
Id not found in TOD file.	Reports an error, and the function always fails.
Unknown limitType.	Reports an error; and the function always fails.

Generally, unused test criteria parameters should be given a value of zero. The user is warned whenever an unused parameter has a non-zero value. Unused parameters do not influence the result of a verify or compare.

Test Method	Condition	Recovery Behavior
EQUAL	Non-zero *limit2.*	None needed.
LESS_THAN	Non-zero *limit2.*	None needed.
GR_THAN	Non-zero *limit2.*	None needed.
PASS_FAIL	Non-zero *limit1.*	None needed.
PASS_FAIL	Non-zero *limit2.*	None needed.
PASS_FAIL	Non-zero *sigDigits.*	None needed.
BIN_COMP	Non-zero *sigDigits.*	None needed.
NOM_TOL	Negative *limit2.*	Absolute value is used
NOM_PER	Negative *limit2.*	Absolute value is used
RANGE	*limit1* > *limit2.*	Values for *limit1* and *limit2* are swapped.

7.4.4 Significant Digits

The significant digits field from a verify/compare determines the number of significant digits applied to the measured value and the limit(s). Legal values for this parameter are zero through six.

The header file (TEXDLL.H) defines the following set of predefined constants for specifying significant digits. These string values are accepted in the TOD file as well.

Significant Digits Value	Constant
SIG_INT	0
SIG_1	1
SIG_2	2
SIG_3	3
SIG_4	4
SIG_5	5
SIG_6	6

7.4.5 Test Methods

Ten different test methods are available for verifies and compares. The following constant values specify how the test criteria are used to evaluate a given test parameter. They are defined in the file (TEXDLL.H).

Value	Description
EQUAL	Passes only if *testParam* is equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
GR_THAN	Passes only if *testParam* is greater than *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
GR_THAN_EQ	Passes only if *testParam* is greater than or equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
LESS_THAN	Passes only if *testParam* is less than *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
LESS_THAN_EQ	Passes only if *testParam* is less than or equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
RANGE	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are *limit1* and *limit2* , respectively. Significant digits are applied to *testParam* and both bounds.
NOM_TOL	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are based on a nominal tolerance: The lower bound is *limit1* - *limit2*. The upper bound is *limit1* + *limit2*. Significant digits are applied to *testParam* and both bounds.
NOM_PER	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are based on a nominal percentage: The lower bound is *limit1* - (| *limit1* | * *limit2*). The upper bound is *limit1* + (| *limit1* | * *limit2*). Significant digits are applied to *testParam* and both bounds.
BIN_COMP	Passes only if *testParam* logically ORed with *limit2* is equal to *limit1* logically ORed with *limit2* (significant digits are not used).
PASS_FAIL	Passes only if *testParam* is non-zero (*limit1*, *limit2*, and significant digits are not used).

The BIN_COMP test method implements a “don’t care” mask, such that all bits set in the mask parameter (*limit2*) are ignored by the evaluation. Note that you reverse the mask behavior if you logically invert the mask parameter. This way, you can achieve a “select” mask; such that only the bits set in the mask parameter are tested by the evaluation.

7.5 User Interface Functions

The user interface functions provide a consistent way for a test object to report warnings, report errors, get operator input, print a message, display a message, or get a yes-or-no answer from the user. When a simple dialog box is required, all user interface functions use a standard Windows message box. The *message* parameter for these functions can include embedded carriage return/linefeed pairs for multiple line formatting and an optional custom title. The *message* length can be up to 1024 characters. Messages greater than that length are ignored. Note that the dialog box's appearance is dependent upon system display parameters (e.g. font size, screen resolution). The input functions allow a prompt message of up to five lines with a total of 160 characters.

User Interface Functions

texErrorBox	An error is shown in a dialog box and written to the TestExec log file.
texInputNumber	Prompts operator for numerical data entry.
texInputString	Prompts operator for text data entry.
texMessageBox	A message is shown in a dialog box and the TestExec log file.
texPrint	A print message is added to the TestExec log file.
texWarningBox	A warning is shown in a dialog box and the TestExec log file.
texYesNoBox	A message is shown in a YesNo box and the TestExec log file.

Calling texErrorBox() always opens a dialog box with an error message and makes an entry in the TestExec log file. This behavior cannot be disabled.

The texInputNumber() and texInputString() functions open a dialog box with a message prompting the user for data entry. This behavior cannot be disabled. Calling texMessageBox() always opens a dialog box with a message and makes an entry in the TestExec log file. This behavior cannot be disabled.

Calling texPrint() always makes a "Print" entry in the TestExec log file. No dialog box is opened. This behavior cannot be disabled.

Calling texYesNoBox() always opens a dialog box with a message, a YES button, and a NO button. If YES is selected, the function returns TRUE; and if NO is selected, the function returns FALSE. This behavior cannot be disabled.

The simple message box style functions can be used independent of texStart(). The behavior of texWarningBox() depends upon the current warning behavior specified for the TEXDLL. Warnings have an independent behavior mode that can be set to one of three values using the following command flags. By default, warnings are disabled at the start of a test.

Flag	Behavior
Warnings	
"/w"	Enable dialog boxes and log entries for all warnings.
"/lw"	Only enable log entries for all warnings.
"/nw"	Disable warnings (default).

7.5.1 Custom Dialog Titles

A custom title can be displayed in the dialog box created by any user interface function. Custom titles are specified by appending a title string to the message string according to the following format:

Format	Example
"*message* | *title*"	"This is a message|This is a title"

Each user interface function accepts a message string as an input parameter. In the example shown above, the message string is "This is a message" and the title string is "This is a title".

The vertical line character (|) serves as the delimiter separating the message and title. This character is not displayed as part of the message or the title. If no title is specified, a different default title is displayed for each type of user interface dialog box. The default dialog titles are:

User Interface Function	Default Dialog Title
texErrorBox	"TEX ERROR"
texInputNumber	"Input Number"
texInputString	"Input String"
texMessageBox	"TEX DIALOG"
texPrint	This function does not create a dialog box.
texWarningBox	"TEX WARNING"
texYesNoBox	"TEX DIALOG"

The functions texInputString() and texInputNumber() allow titles of approximately 40 characters in length. The other functions can display titles of approximately 80 characters in length. Note that the specific length of the title is dependent upon the operating system, screen resolution and font selection (i.e. large or small). Titles greater than 128 characters are ignored by most functions.

7.6 Information Functions

This set of functions is used to optionally extract information from some of the files that are active during execution. This provides additional information to a test object that might be necessary for data logging or to modify its operation. Many test objects will not need to call these functions.

Information Functions

texGetTODVerify	Retrieves verify parameters from the current Test Object Definition file.
texGetTPC	Retrieves configuration data from the current Test Program Configuration file.
texGetTPD	Returns items extracted from the current Test Program Definition file.

The texGetTODVerify() function can be used if some modification is required for verify parameters stored in a TOD file. The modification might be for measurement scaling or to add additional information to one of the parameters. For example, the verify description parameter might be modified to include a timestamp for the measurement. Or, the prefix and units parameters might be set to the failing string following a string comparison. After the parameters are modified, the verify is performed by passing them to texVerify().

The texGetTPC() function provides information about the current circumstances for the current test. This includes part numbers and revisions of test equipment and processes. This information could be used to control test sequencing, or criteria, based on a UUT's revision or the type of test interface adapter that is present.

The texGetTPD() function returns information from the current TPD file. This can be used to retrieve additional information about a test object's description, help file, paragraph number or power monitor settings. This function can also return the TPD file name, which is necessary when accessing any user-defined portions of a TPD file.

7.7 Other Functions

7.7.1 Embedded Breakpoints

Embedded breakpoints are created in a test object's code by calling the following TEXDLL function:

texEmbedBreakpoint ((LPSTR) *message*, (short) *type*);

Where *message* is a breakpoint message provided by the test object, and *type* is a predefined constant value that distinguishes data breakpoints from trace breakpoints.

There are two types of embedded breakpoints available:

Type	**Purpose**
TEX_BRK_DATA	Data breakpoints provide an opportunity to display intermediate measurements during a test.
TEX_BRK_TRACE	Trace breakpoints are used to indicate the path of execution during a test.

Each type of breakpoint has an independent behavior mode that can be set to one of three values using the following command flags. By default, both breakpoint types are disabled at the start of a test. Note that breakpoints are useable independent of texStart().

Flag	**Behavior**
Data Breakpoints	
"/bd"	Enable dialogs and log entries for all data breakpoints.
"/lbd"	Only enable log entries for all data breakpoints.
"/nbd"	Disable data breakpoints (default).
Trace Breakpoints	
"/bt"	Enable dialogs & log entries for trace breakpoints.
"/lbt"	Only enable log entries for all trace breakpoints.
"/nbt"	Disable trace breakpoints (default).

7.8 TOD Files

One TOD input file can be specified in the command string passed to the texStart() function. (e.g. "/tod sample.tod"). For a detailed description of the TOD file format, see *Test Object Definition Format*. The TEXDLL requires a TOD file to contain the following information:

- An Identification block that specifies a Part Number, a Revision, and a Format Version for the TOD file. The Format Version must be 1.xx to be compatible with the current version of the TEXDLL. For example: 1.00, 1.01, 1.50, 1.99 are all considered compatible; but version 2.0 is considered incompatible.
- A Verifies block containing zero or more verify definitions. Each verify definition consists of a line that specifies eight test criteria.

The TEXDLL automatically reads the specified TOD file when texStart() is called. An error is reported if the TOD file cannot be opened, if an incompatible Format Version is read, if there are more than 1000 verify definitions, or if one or more warnings were raised about the file.

Warnings are reported for the following problems:

- Any part of the required information listed above is missing.
- A verify definition contains an unknown verify type.
- A test criteria parameter is not of the correct data type.
- A verify definition lists fewer than eight test criteria parameters.

If one or more problems are detected in a TOD file, the texStart() function returns FALSE. This denotes that TEXDLL initialization has failed and the Developer API functions are not available. The entire TOD file is read by the TEXDLL so that many problems can be detected in one pass. Note that warnings must be enabled to create a record of specific TOD problems in the TestExec log file. Inappropriate test criteria values (e.g. "No description string.", "Too many significant digits.", ...) are only detected at runtime, when a verify or compare uses the criteria.

7.9 Test Object Help Files

One help file can be specified in the command string passed to the texStart() function (e.g. "/hlp sample.hlp"). This help file is used to provide context-sensitive help support for the verify operations contained in a test object. An error is reported if TEXDLL cannot open the help file when texStart() is called. If a help file is available, an OK button and a HELP button appear on the verify breakpoint dialogs. Selecting the HELP button invokes the Windows' function, WinHelp(), with the verify's *id* parameter specified as a partial key.

An example test object help file is included with the Test Executive's demo TPD files. The text source for the help file is included in the SAMPLES subdirectory.

7.10 Test Objects and Microsoft Windows

Test and setup objects are executable modules that run in Windows 95, 98, NT or Windows 2000. Consequently, test objects must adhere to certain requirements imposed by the operating system. Most modern software development environments support writing Windows-compatible code. The following sections, *Windows Compliance for Test Objects* and *Windows GUI and Multitasking*, quickly cover important concepts for developing test objects that will operate smoothly within the Windows environment.

In addition, the Test Executive requires that test and setup objects must signal their activity (see *Access Functions*) and respond to abort requests. Of these, abort handling is by far the more difficult subject to master. As described in the *Test Executive Architecture* and *Test Program Design* sections, test objects operate in three different ways: Executable, DLL and ActiveX DLL. Each of these has a specific behavior and certain requirements for responding to abort requests from the Test Executive. See the following section, *Aborting a Test*, for an introduction to abort processing for test objects.

7.10.1 Windows Compliance for Test Objects

Test and setup objects are programs or executable modules (e.g. DLLs) that run under a Windows operating system. All Windows programs, including test objects, must implement certain behavior that keeps them "compliant" with the operating system and with other programs. The following sections describe how to achieve Windows compliance with little or no work.

Note that no Windows programming is required to use the TEXDLL's test object programming interface. This interface supports interaction between a test object and the Test Executive. By using the TEXDLL, test objects can verify test parameters, insert breakpoints, and control a variety of user interface dialog boxes.

There are two important Windows compliance issues that every test/ setup object must satisfy:

1. Compliance with the Windows graphical user interface (GUI) and multitasking. This includes message handling and releasing time back to the operating system. Even though Windows 95, 98, NT and Windows 2000 all have preemptive multitasking, it is still important to know when to release time in order to better enable other processes. For more information see section *Windows GUI and Multitasking*.

2. Compliance with an abort request from the Test Executive. How an abort request is implemented is dependent upon the type of test object being interrupted (i.e. executable, DLL or ActiveX DLL). For an overview see the section, *Aborting a Test*. More specific information is available in sections *Aborting a Standalone Test Object*, *DLL Abort* and *Aborting an ActiveX DLL*.

Many program development environments provide an easy way to create programs that are Windows compliant. For example, little work is required to achieve Windows compliance using Visual Basic. Item one is addressed by occasionally calling the DoEvents() function, from your test object, if long sequences of operations are being performed. The second item is handled automatically, and additional control can be implemented in the main form's QueryUnload() or Unload() functions. Alternately, most development environments provide a sizable collection of example applications that can be modified to suit particular test object requirements.

7.10.2 Windows GUI and Multitasking

Applications in Windows have queues that accumulate messages from user events (e.g. mouse clicks), the system and other applications. These messages are processed by each application and responded to in an appropriate manner. If an application doesn't process its messages, it ceases to respond to external conditions and events. Therefore, message processing is an important fundamental behavior of a compliant Windows program.

It is very important for all test objects to process their message queue. If messages are not processed, the test object can't respond to user and system requests (e.g. resize, repaint, move, etc.). This can cause the test object to appear 'frozen' and makes it unresponsive to the operating system. An application often releases control to other applications when it services its message queue.

Windows 95, 98, NT and Windows 2000 are preemptive multitasking operating systems. This means that an application is often interrupted by the operating system so that another program can execute. Typically each application is allotted a 'slice' of CPU time for execution. These time-slices are normally around 20 milliseconds each.

While it is not strictly necessary for a 32-bit test object to explicitly release time to Windows, it is generally considered a good programming practice. It is very important that control loops waiting for external conditions (e.g. time-outs or instrument responses) should release time whenever possible.

Different languages and development environments support various ways to process message queues and release time to Windows. From a Visual Basic program, you process the message queue, and release time, by calling the DoEvents() function. From a standard C program, you can service the message queue, and release time, by using the winRelease() function shown below.

```
#include <windows.h>

void winRelease(void)
{
        MSG msg;

        while (PeekMessage((LPMSG)&msg, NULL, 0, 0, PM_REMOVE))
        {
                if (msg.message == WM_QUIT) break;
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
}
```

More detailed information on Windows multitasking, message handling, or the Windows API, can be found in numerous introductory books on Windows programming. Additional information is also available from Microsoft's Web site.

7.10.3 Aborting a Test

Aborting a test object is one of the more complicated and misunderstood aspects of the Test Executive. The concept is simple, press the Abort button to stop the test object's execution. Unfortunately the interaction between the Test Executive, the test object and the operating system can become quite complex. So the first question that should be asked is: "What do you need to accomplish?"

Halting a Test Object

It is actually quite easy for the Test Executive to halt a test object's execution. The Test Executive takes the same steps that the Windows' Task Manager does when a user presses the **End Task** button. This requires no support whatsoever on the part of the test object. While this can work well for standalone executable test objects, it is less successful for DLLs and ActiveX test objects. The reason for this is that a standalone test object is a separate process and the operating system releases resources when a process is terminated. DLL and ActiveX test objects are executed as a thread in a process. Terminating an execution thread does not automatically release its resources. A test object's resources can include allocated memory, opened files and instrument handles. Aborting without releasing resources can cause memory leaks, corrupt files and 'locked' instruments. Reloading the TPD, with an abort setup object, will help to release a test object's resources (see *TPD Reload* and *Abort Recovery* sections).

Detecting an Abort Request

In order to avoid being arbitrarily interrupted, a test object must detect an abort request from the Test Executive and terminate itself. This gives the test object an opportunity to free memory, close files and release instrument handles before quitting. A test object is allowed one second to respond to an abort request before the Test Executive terminates its execution. A test object may need additional time to handle an abort request so there is a mechanism provided for that. See the *Critical Processes* section immediately below as well as the section on *Abort Delay*.

Unfortunately each type of test object has its own requirements and behavior around handling abort requests. These are described in the following sections: *Aborting a Standalone Test Object*, *DLL Abort* and *Aborting an ActiveX DLL*. Source code examples are included with the Test Executive to illustrate abort processing for a standalone executable (\Samples\Msvc4\TxProg32) a DLL (\Samples\Msvc4\TxObjDLL) and an ActiveX DLL (\Samples\ActiveX\ssTestObjX.vbp).

Critical Processes

Sometimes a test object has tasks that could be hazardous to a UUT, or operator, if arbitrarily interrupted. Typically these conditions only exist for a short period of time during testing. These circumstances can be easily handled by having the test object request additional time to complete such a task before being aborted. The **texSetAbortDelay** function can be called by a test object to request from 1 to 120 seconds "grace period" before being interrupted by an abort. Thus this requires that a test object identify and protect critical areas of execution.

Recovering from an Abort

When a test object is aborted there may be "cleanup" tasks that need to be performed. These can include deleting temporary files or turning power supplies off. The Setup Objects section of a TPD file supports multiple "Abort Objects" to handle exactly this situation. The test developer simply needs to create an executable that restores or reinitializes conditions and instruments to a known state. This will probably be similar to the code that a test object uses when it completes. This does not require any special code or handling by the test object.

7.10.4 Aborting a Standalone Test Object

The Test Executive has an “Abort” button that allows the user to interrupt execution of a test object and return control to the Test Executive. When “Abort” is pressed, the Test Executive goes through the following series of operations to shut down a standalone executable test object:

1. Try to abort the test object by sending a WM_CLOSE message to all enabled top-level windows owned by the test object. The WM_CLOSE message represents the request for a test object to unload its resources and close. This message is sufficient to close a Windows-compliant application. If the test object is a DLL, it also attempts to call a user-created abort function (**usrAbortTest**).

2. Wait for the test object to close. The TEXDLL releases control to Windows and waits until the test object closes or a designated time elapses. If the test object closes within the specified time period, the abort operation concludes. This designated time is called the abort delay. Test objects can dynamically specify the abort delay using the function, texSetAbortDelay(). The default delay is one second; and legal values range from 1 to 120 seconds.

3. If the test object is still active following the abort delay, the TEXDLL uses a Windows API function, TerminateProcess(), to terminate execution of the test object. When TerminateProcess() is used, the test object exits as if a general protection fault occurred. There is no opportunity for the test object to perform cleanup and DLLs are not notified that the application is closing.

For an abort operation to execute cleanly, the test object should recognize and respond to a WM_CLOSE Windows message sent by the TEXDLL. Many program development environments (e.g. Visual Basic, LabWindows/CVI, Visual C++) have options for automatically supporting Windows messages such as WM_CLOSE. Under these circumstances no additional support is required from the test object. Abort setup objects are supported by the Test Executive to handle any necessary cleanup or reinitialization.

In some rare situations a test object may need to more actively respond to an abort condition. This might occur if a specific power down sequence is required that cannot be handled by a general Abort setup object. In this case, the abort delay could be increased to ensure the test object has sufficient time to close. The test object must be written to detect an abort by checking for the receipt of a WM_CLOSE message. This requires a detailed understanding of Windows message handling and is only recommended for experienced Windows programmers.

Note: The message processing described in a previous section is very important for the proper receipt of the WM_CLOSE and subsequent messages.

When a Windows program is being closed, a series of messages are dispatched and received. The default response to a WM_CLOSE message is for an application to dispatch a WM_DESTROY message to itself. When WM_DESTROY is received, a WM_QUIT is dispatched. The reception of a WM_QUIT message indicates the application is to exit (typically detected in its message processing loop).

Many factors influence whether a test object will exit gracefully upon receiving a WM_QUIT message. These include what resources are loaded, active threads and the specific message handling routines. If a controlled abort shutdown is desired, it might be advisable to terminate the application after responding to WM_DESTROY. 32-bit test objects can terminate themselves by calling the Windows API function ExitProcess().

It is quite simple to add abort handling procedures to Visual Basic test objects. The WM_CLOSE message causes a QueryUnload() event for each of the loaded forms in the application. An input parameter indicates why the form is closing (e.g. from code, user selection or another application). Abort specific code can be executed if the form is being closed by another application. The QueryUnload() event occurs in all forms before any are unloaded, and the Unload() event occurs as each form is unloaded.

To handle abort processing, some developers have created 32-bit test objects with multiple threads. One of the execution threads is dedicated to running tests, while the other deals with message processing. If the message processing thread receives a WM_CLOSE, it can shut the testing thread down and perform any necessary cleanup. This approach does require a thorough understanding of 32-bit programming and multithreaded behavior.

7.10.5 Abort Delay

The abort delay is how long the TEXDLL waits for a test object to terminate itself following an abort request. The default delay is one second; and legal delay values range from 1 to 120 seconds. The abort delay can be controlled from within a test object by the texSetAbortDelay() function, or externally with a command flag ("/ad *delay*").

The abort delay can be changed at any time during a test, so an appropriate delay can be in effect at all times. The abort delay automatically returns to the default value at the end of each test. This means that the abort delay is reset to one second by texStart() and texFinish().

When initially implementing abort handling, some developers have difficulties with the sequence of events that surround an abort request. This usually leads them to using the abort delay and breakpoints in an attempt to observe the steps that occur. Often this does not behave the way that they expect and the abort delay is blamed. The following are a few additional observations about a test object abort:

1. Once an abort is instigated, all TEXDLL display functions (e.g. texMessageBox) are disabled. This means that these functions won't display during abort processing and can't be used to "freeze" execution when testing the abort delay.

2. Once an abort is instigated, verify information is not transferred to the Test Executive. This can give the impression that the test object has been prematurely terminated. This is not the case. Also, by not passing verify information to the Test Executive, the verifies are executed more quickly. Consequently, you cannot rely on verify execution time to test the abort delay. The best way to test the abort delay is to have your test object execute a Windows API Sleep command. A single input parameter defines the number of milliseconds for execution to be suspended. A delay loop using CVI functions may not operate the same (CVI has automatic message handling in some of its functions).

3. The abort delay can be defined as a command line argument (e.g. "/ad 20") to your test object (see the section on *Command Flags*). This is particularly handy when testing abort behavior.

4. A test object DLL with a visual interface (e.g. window or form) receives a WM_CLOSE message when an abort is requested. Unless handled, this could supercede the processing that occurs in the **usrAbortTest** function. For more information see *DLL Abort* in *DLL Test & Setup Objects*.

7.11 DLL Test & Setup Objects

7.11.1 DLL Loading and Execution

Most Test objects require several support DLLs in order to operate. These include instrument drivers, class libraries and custom DLLs. Loading several large DLLs is time consuming, even on a fast test system. This was minimized in the 16-bit Test Executive by allowing DLLs to be preloaded as a setup Object. This greatly improved the test-to-test speed because the DLLs were not being reloaded during each Test object execution.

Windows NT enforces a strict isolation between processes. Consequently, preloading DLLs with the Test Executive doesn't help a test object that runs in a separate process. The 32-bit Test Executive now supports a second process that handles DLL preloading and allows Test objects to be defined as DLLs. This then places the test object DLLs and preloaded DLLs in the same process, eliminating the redundant DLL reloading. It also allows test and setup object DLLs to share a set of common instrument handles, as has been successfully done in the 16-bit version. Additionally, this allows Test object DLLs to be executed more quickly because a new process isn't created each time.

Test object DLLs are handled by a separate process in order to insulate the Test Executive from programming errors. Otherwise, the Test Executive would be vulnerable to Test object crashes or mistakes (e.g. a simple **ExitProcess** in a Test object DLL would shut the Test Executive down).

To use this enhancement, there are several application issues that must be handled: These include DLL dependencies, starting a DLL function, aborting a DLL object, error recovery and debugging. The following sections cover these topics.

Example Test Object DLL

An example test object DLL (TobjDLL.dll) is included with the Test Executive installation. Its behavior is similar to that of the example test object described in Appendix G - Sample Test Object. The complete source code for the example test object DLL is included in a subdirectory to the Test Executive (“\Samples\Msvc4\TxObjDLL”).

7.11.2 DLL Dependencies

Preloading DLLs can significantly improve the speed at which test object DLLs load and execute. Preloaded DLLs also maintain their state, so acquiring instrument handles and allocating resources only needs to occur once. There can be subtle complexities that occur when multiple DLLs are required by a test object. These typically are related to the dependencies that exist between the various DLLs.

When the Windows operating system loads an object module (i.e. program or DLL) for execution, it also loads the DLLs that the module requires. Loading these DLLs can cause a cascading chain of loading other DLLs as the requirements (i.e. dependencies) for each are resolved. This process encounters difficulties if Windows is unable to locate a DLL for loading. Windows has a defined sequence of locations that are searched in order to locate a DLL. This is known as the Windows "search path" and it is described below:

The Search Path Used by Windows to Locate a DLL

With both implicit and explicit linking, Windows first searches the set of pre-installed DLLs such as the performance library (KERNEL32.DLL) and the security library (USER32.DLL). Windows then searches for the DLLs in the following sequence:

1. The directory where the executable module for the current process is located. For a test object DLL, this is the Test Executive's home directory because the process executable (texDLLControl) is located there.

2. The current directory. For a test object, this is usually the location of the TPD file.

3. The Windows system directory. The **GetSystemDirectory** function retrieves the path of this directory.

4. The Windows directory. The **GetWindowsDirectory** function retrieves the path of this directory.

5. The directories listed in the PATH environment variable. Note that the LIBPATH environment variable is not used.

As can be seen from the above, DLLs in the Windows directories are easily located. DLLs in other directories are dependent on the PATH environment variable, the location of the process executable or the current directory. These limitations thus require careful planning when preloading multiple, dependent DLLs.

If any of the preload DLLs are commonly used by the test system (e.g. instrument drivers), it might be simplest to install them in the Windows system directory. Certainly this is a viable option for interface drivers such as VISA or GPIB.

Otherwise, the dependencies between the DLLs must be determined so they can be loaded in reverse order. For example, three DLLs in the TPD's setup directory are to be preloaded. DLL 1 is dependent on DLL 2; and DLL 2 is dependent on DLL 3. Trying to load DLL 1 first fails because Windows can't locate DLL 2. Therefore, these DLLs should be preloaded in the following order: 3-2-1.

Unfortunately, many versions of Windows do not clearly state the reason that a DLL could not be loaded. In the above example, trying to load DLL 1 first would typically cause an error message something like: "Could not find or load DLL 1". This usually results in frustrated attempts to move DLL 1 to somewhere that Windows can "find" it. Meanwhile, the problem is actually that Windows cannot find DLL 2.

7.11.3 DLL Startup

Standalone test objects receive command line arguments when they are started. These arguments include information from the Test Executive (i.e. breakpoint settings, help files, etc.) and from a user-defined field in the TPD file. This behavior is handled by passing a single string parameter to the specified function in a test object DLL. The arguments field in a TPD test specification defines the DLL function to call. Additional operating parameters are included within trailing parenthesis.

```
Test = 2.3.;Test Object DLL; ; ; ; tobjdll.dll; usrExecuteTest(/loop 3); ; ;
```

Different functions can be called within the same test object DLL. A test object DLL is always loaded, executed and unloaded. To keep it loaded during testing, preload it as a setup object DLL.

Test object DLLs are responsible for releasing any DLLs or resources that they load during their operation. Since the process that they operate in is not ended, resources (e.g. allocated memory) could accumulate for each execution.

Previously, a DLL setup object was simply loaded for a specified duration (e.g. during a sequence execution). Now DLL setup objects are also able to execute a specified function. Leave the arguments field empty to have the setup DLL loaded only. To keep the DLL loaded, and execute one of its functions, use two setup object entries.

```
PreLoad = \test\tobjdll.dll; ; ;
PreLoad = \test\tobjdll.dll; usrExecuteTest(/pn Preload); ;
```

Prior to execution, functions defined for DLLs are verified to exist.

7.11.4 DLL Abort

A standalone executable test object operates as an independent process and handles its own messages. When such a test object is aborted, a carefully orchestrated sequence of events occurs that allows the test object to shut itself down. If it doesn't, the Test Executive forces it to halt with Windows API functions.

If an abort occurs during execution of a test object DLL, a default function (**usrAbortTest**) is called via a new thread. Its input parameter is a handle to the test execution thread. This function's task is to safely halt the test execution and perform any recovery or cleanup operations. Both threads are allowed the abort delay before they are terminated by the TexDLL (via TerminateThread). Calling a user-created function to handle an abort request is supported because a DLL may not have a conventional message-processing interface (i.e. window or form). The sample test object DLL that is included with the Test Executive is an example of this.

After **usrAbortTest** is called, WM_CLOSE messages are dispatched to all enabled top-level windows owned by the test object. The abort delay is then measured while the test object is monitored to see if it terminates. This means that a test object DLL with a visual interface must receive and respond to a WM_CLOSE message as described in the section: *Aborting a Standalone Test Object*.

Note: Extreme care must be taken when developing a DLL that will serve as an abort object. Since the Test Executive does not allow abort objects to be aborted, it is difficult to deal with an abort object DLL that "hangs" (e.g. gets stuck in an endless loop). DLLs are not visible from the Windows Task Manager, so they are not easily detected or terminated. Terminating the Test Executive's DLL sharing process (texDLLControl) does force the abort object to shutdown.

There are several approaches to handling abort requests with a test object DLL. The simplest might be to have the user abort function (**usrAbortTest**) set a flag, or semaphore, that the execution thread is monitoring. If the execution thread detects an abort request, it can then release allocated resources (e.g. memory, files, instruments, etc) and shut itself down (via **ExitThread**). Fortunately, many test development languages provide an error/exception dispatch mechanism that can be used to bypass the normal flow of control.

For example, the **try**, **throw**, and **catch** statements have been added to the C++ language to implement exception handling. When an exception is "**thrown**", the execution flow returns to the most recent **try** block and executes the code in its companion **catch** block. A detected abort request could thus **throw** an exception that is "**caught**" by code that is aware of what resources need to be released. Most exception handling can daisy-chain itself back through one or more **catch** blocks until it returns to the original execution entry point.

Another tool available, for releasing resources following an abort, is the RELOAD_FINISH option for the texFinish() function. Reloading the TPD, with an abort setup object, helps to release a test object's resources. For more information, see the *TPD Reload* and *Abort Recovery* sections).

7.11.5 DLL Debugging

Development environments that are capable of creating DLLs also have many tools for debugging them. Ideally, a test object DLL should be debugged standalone, within the development environment, before attempting to integrate it with the Test Executive operation. In many cases the additional complexity of running from the Test Executive can hinder the early stages of DLL debugging. Sometimes a DLL being debugged requires other DLLs to operate (e.g. a preloaded setup object DLL). These additional DLLs can be added to the project for static loading, or dynamically loaded with a few lines of code (e.g. the Windows API function LoadLibrary). Some development environments, such as Microsoft Visual C++, specifically support loading additional DLLs during debug. Again, the goal is to debug the DLL as completely as possible before attempting to integrate it within the Test Executive operation.

Occasionally additional debugging is required once the DLL is being executed by the Test Executive. Often this can be accomplished by the selective use of texEmbedBreakpoint calls. In some extreme cases a lower level of interactive debug may be required.

Program development environments typically support DLL debugging by having the user identify the executable that will call the DLL. When debugging begins, that executable is started and its' calls to the DLL are monitored for breakpoints. As described earlier in this section, the Test Executive uses a separate process to load and execute DLLs. This separate program (TexDLLControl) operates as an out-of-process server. Therefore **TexDLLControl.exe** is the executable that the development environment should launch for debugging. Because it functions as a server, it requires a command line argument (**/embedding**) in order for it to remain in memory.

Note: The development environment must start executing TexDLLControl before the Test Executive is run. Otherwise the development environment will be unable to attach to the correct process for debugging.

The DLL to be debugged is then loaded and executed by the out-of-process server when instructed by the Test Executive. Note that the DLL function is executed by a secondary thread of the TexDLLControl process. It is possible that you will need to close the Test Executive and restart debugging after one execution of the DLL. For example, Microsoft Visual C++ v4.1 only monitors the first thread that executes the DLL. Subsequent executions are ignored by Visual C++ v4.1 until the Test Executive is closed and debugging is restarted.

It is strongly recommended that you use the example test object DLL code to check the interactive debugging behavior of your development environment. It is far better to experiment with a small test case than to attempt it with a large quantity of unknown code. Note that the debugging behavior may be affected by the operating system, development environment type and its version.

Note: Prior to version 5.5, National Instruments LabWindows/CVI had certain restrictions for DLL debugging. The following describes a method for debugging test object DLLs with earlier versions of LabWindows/CVI.

Additional steps are required to use the above approach with earlier versions (prior to v5.5) of National Instruments LabWindows/CVI. Consequently, a debugging support DLL is included with the sample test objects. This DLL, **CVIDebug.dll**, loads the target DLL in a manner that enables CVI debugging. To use it, specify its name as the test object executable and place its' function, **usrDllDebug**, in the arguments field:

```
Test = 2.1.;Debug_DLL;;;;CVIDebug.dll;usrDllDebug(≥);
```

Then, within the trailing parenthesis, include the target DLL name, target function name and any other arguments. Separate these by a vertical pipe character (|). The CVIDebug DLL must be in a subdirectory (Test) to the TPD file's location. If the target DLL is in the same directory it requires a partial path (see below), or a full path if it resides elsewhere.

```
≥; usrDllDebug(test\tObjDLL.dll|usrExecuteTest| /f );
```

To work properly for debugging, the CVIDebug DLL should not be preloaded by the Test Executive. It is important that it be released (i.e. unloaded) after each execution so that it always reloads the CVI DLL. This is because earlier versions of the LabWindows/CVI development environment have the following restriction:

> **Multithreaded Executables**
> LabWindows/CVI can debug only one thread. If a multithreaded external process calls functions in debuggable DLLs in more than one thread, LabWindows/CVI can debug only the thread that first loaded a debuggable DLL. LabWindows/CVI does not honor breakpoints and watch expressions in the other threads. If the external process unloads all the debuggable DLLs and then loads another, LabWindows/CVI can debug only in the thread that loaded the new DLL.

If the CVIDebug DLL is used with Microsoft Visual C++ it can allow multiple debug runs without restarting the Test Executive. To achieve this behavior the DLL being debugged must execute the Windows API function **ExitThread** upon its completion.

7.11.6 DLL Error Recovery

It is very common for a test object to crash during development. If the test object is a DLL, a crash will most likely kill the process (texDLLControl) that is created by the Test Executive for DLL sharing. Consequently, the Test Executive detects such a crash and recreates the process with its loaded DLLs. The operator is notified that such a recovery has occurred.

7.12 ActiveX DLL Test Objects

In order to allow closer integration of test objects written in Visual Basic, version 6.3 of the SSI Test Executive supports test objects that are ActiveX DLLs. ActiveX DLLs are COM objects that are created with Microsoft Visual Basic 6.0 (Service pack 3 or later). These ActiveX test objects are executed in a common process that allows them to share system resources, such as instrument handles, with "regular" DLLs.

The common process that executes both ActiveX and regular DLLs is called texDLLControl. It operates independently from the Test Executive in order to provide protection from a test object crash. When executing ActiveX DLLs, texDLLControl uses an additional support library module (texActiveX.dll). For an overview of this behavior, see the *Test Executive Architecture* section.

There are many subtle differences between Visual Basic ActiveX DLLs and "regular" DLLs that are created with a C compiler. The differences include coding, debugging, registration, threaded operation, shared resources, naming conventions and abort handling. The following sections discuss these and related topics.

7.12.1 Building an ActiveX Test Object

It is easy to start a Visual Basic ActiveX test object. Tell the Visual Basic development environment to create a new project and specify that it is to be an ActiveX DLL. This results in a new project (Project1) with a single class module (Class1). Immediately rename the project and class in order to make them unique (e.g. xxProject1 and xxClass1). See the following section for more information on project and class naming. Use the *Project* menu to *Add Module* and add the TexDLL definitions (TexDef32.bas) to the project.

From the *Tools* menu select *Add Procedure…* and enter a name for the Public Sub procedure that you are creating (e.g. MainTest). For a test object, the procedure must take a single string input parameter. Add **texStart** and **texFinish** and you have the beginnings of a test object:

```
Public Sub MainTest(ByVal CmdBuf As String)

    texStart CmdBuf

    texFinish NORMAL_FINISH

End Sub
```

At this point you can compile the project and call it from the Test Executive. Note that to access the test procedure, the TPD *Arguments* field must include the project, class and function names (e.g. xxProject1.xxClass1::MainTest).

As easy as it is to get started, an ActiveX test object has very specific requirements for naming, resource sharing, abort handling, debugging and system registration. All of these are discussed in the following sections.

7.12.2 Naming an ActiveX Object

ActiveX DLLs contain one or more class modules. Each class has a unique name and defines its own variables and procedures. Procedures designated as *Public* are callable from outside the class. At runtime an executable object is created (instantiated), by the Test Executive, from the class module. In the operating system, objects are classified by their project name and class (e.g. xxProject1.xxClass1). Consequently it is a good idea to plan meaningful project and class names from the very beginning of test object development. Otherwise you might end up with obscure or generic names that are uninformative or not unique. Unique names are necessary because all ActiveX classes are placed in the system registry (see *Registering ActiveX DLLs*). Multiple versions of *Project1.Class1* would cause insidiously subtle conflicts because the only version that would be executed is whichever was last registered.

7.12.3 Object Creation and Shared Memory

In the Test Executive, an ActiveX object is created, run and closed each time it is encountered on the test list. This means that there is no shared memory from one execution to another. Preloading an ActiveX DLL only reduces the time required to load it into memory, it does not provide any persistent or shared memory during testing. The lack of shared memory is a result of the "object creation" required for ActiveX DLLs and the compartmentalization of individual threads (see the thread discussion in the following abort section). There are also several significant restrictions on multithreading in Visual Basic (see Microsoft article Q241896).

One way to share information between object instances is to use the *Persistable* property in the class module. When active (True), this property adds methods and events to a class that aid in tracking changes to variable values. This allows them to be stored when the object is terminated and read back when a new object is created. There is some variable management code that must be written in order to use this mechanism. A possibly simpler approach would be to write a user-defined type structure to a binary file and read it back in order to save and restore object properties.

Another way to share resources and information between test objects is to create a DLL setup object that acts as a repository for intermediate data. This would need to be a "regular" DLL that is preloaded so it stays in memory during testing. ActiveX DLL test objects can then request or set data via function calls to the preloaded setup object. Alternately, the setup object could allocate a block of named shared memory by defining a file-mapping object. Other test and setup objects can then access the memory by requesting a pointer from the operating system via **MapViewOfFile**. For more information consult Microsoft's web site and Developer Network (MSDN) database.

When a Visual Basic object is created its Class_Initialize function is executed. This is used to set initial conditions for the object. For an ActiveX test object it is recommended that most initial conditions, particularly involving instruments, be set at the beginning of the test procedure. The reason for this is that once the test procedure executes **texStart**, the TexDLL Error/Warning/Breakpoint functions are available for reporting and diagnosing setup or instrument problems. **texStart** cannot be executed in Class_Initialize because the command flags, required by **texStart**, are only passed to the designated test procedure.

When a Visual Basic object is closed, its Class_Terminate function is executed. This is used to release resources for the object. It is recommended that all shutdown procedures be executed within the test procedure. This again gives access to TexDLL functions for reporting shutdown problems prior to issuing a **texFinish**. Also, depending on abort handling, it is possible that the Class_Terminate function might not be called.

7.12.4 Aborting an ActiveX DLL

On the surface, aborting an ActiveX DLL seems very similar to aborting a regular DLL. A user-created function (usrAbortTest) is called on a second thread when an abort is requested. A single input parameter defines the handle for the thread that is executing the test procedure. The function's job is to coordinate an orderly shutdown of the test object. Unfortunately the underlying mechanism of an ActiveX DLL makes the abort handling trickier than for a regular DLL. The reason for this has to do with how threads are handled in an ActiveX DLL. The following is extracted from *Visual Basic Concepts* in the Microsoft MSDN:

> In Visual Basic, *apartment-model* threading is used to provide thread safety. In apartment-model threading, each thread is like an apartment — all objects created on the thread live in this apartment, and are unaware of objects in other apartments.
>
> Visual Basic's implementation of apartment-model threading eliminates conflicts in accessing global data from multiple threads by giving each apartment its own copy of global data… This means that you cannot use global data to communicate between objects on different threads.

Since the abort function needs to communicate with the test procedure, the Test Executive uses *cross-thread marshaling* to execute usrAbortTest in the same apartment, and object, as the test procedure. This synchronizes (serializes) the execution of the test procedure and the abort function. This means that the test procedure is suspended while the abort function is active.

The best way to handle an abort request is for the abort function to notify the test procedure that an abort is pending. This can be easily handled by a global flag variable. The test procedure should then shut itself down when it detects that an abort is pending. This allows the test procedure to control when a shutdown occurs so that instruments and resources are left in a known state. Though this is the best approach, it does require that the fundamental design of a test procedure must include abort request handling and awareness.

A simpler but riskier approach is for the abort function to execute the Windows API command ExitThread. This effectively shuts down the test procedure and releases system resources. It is definitely preferable to the Test Executive executing a TerminateThread which doesn't release system resources. The risky part is that the abort function might be interrupting a critical or hazardous portion of the test procedure. Note that ExitThread also immediately halts the abort function because it has been marshaled to the same thread as the test procedure.

7.12.5 Debugging an ActiveX DLL

The best way to debug an ActiveX test object is from within the Visual Basic interactive development environment (IDE). The code to exercise an ActiveX test object is very simple:

```
Dim testObj As xxProject1.xxClass1

Set testObj = New xxProject1.xxClass1
testObj.MainTest "/bt /w /tod xxSample.tod"

Set testObj = Nothing              ' Terminate object
```

This code can be run from a Standard EXE project that coexists with the ActiveX DLL project in the Visual Basic IDE. This makes iterative testing and corrections a very dynamic process. It also avoids many integration issues that occur when debugging with the Test Executive active. Note that the string parameter passed to MainTest supports all of the command flags that the Test Executive uses to control test object behavior (e.g. breakpoints, warnings, support files, etc.).

Occasionally additional debugging is required once the ActiveX DLL is being executed by the Test Executive. Sometimes this can be accomplished by the selective use of **texEmbedBreakpoint** calls. In some cases interactive debug may be required.

The Visual Basic IDE supports ActiveX DLL debugging by having the user identify the executable that will call the ActiveX DLL (*Start Program* on the *Debugging* tab of the *Project Properties* dialog). When debugging begins, that executable is started and its' calls to the ActiveX DLL are monitored for breakpoints.

The Test Executive uses a separate process to load and execute ActiveX and regular DLLs. This separate program (TexDLLControl) operates as an out-of-process server. Therefore **TexDLLControl.exe** is the executable that the Visual Basic IDE should launch for debugging. Because it functions as a server, it requires a command line argument (**/embedding**) in order for it to remain in memory.

> **Note: The IDE must start executing TexDLLControl before the Test Executive is run. Otherwise the IDE is unable to attach to the correct process for debugging.**

The DLL to be debugged is then loaded and executed by the out-of-process server when instructed by the Test Executive. If the ActiveX debug session is halted, the Test Executive may need to be restarted.

7.12.6 Registering ActiveX DLLs

Contrary to regular DLLs, ActiveX DLLs must be registered with the operating system in order to be executed by the Test Executive. The Visual Basic development environment automatically registers an ActiveX DLL when it is compiled. When you move an ActiveX DLL to another computer it needs to be registered. This is easily accomplished by running *Regsvr32.exe* from the Run dialog:

```
Regsvr32 c:\TPD\UUT_123\test\ssTestObjx.dll
```

There are also Windows API functions that aid automated registration from code (see Microsoft article Q207132). Note that once an ActiveX DLL is registered, the operating system expects to always find it in the same location (i.e. directory). Thus ActiveX DLLs cannot be arbitrarily moved elsewhere in the execution path as regular DLLs can. Simply renaming a registered ActiveX DLL does not hide it from the operating system. Therefore it is important to carefully manage the versions and registration of ActiveX DLLs in order to avoid confusion during development and test system integration.

When a TPD is loaded, the referenced ActiveX classes (e.g. xxProject1.xxClass1) are checked to see if they are registered. If an unregistered class is detected, the operator is offered the opportunity to automatically register the associated ActiveX DLL. If operator privileges are active, automatic registration is only offered to operators with TPD privileges.

7.12.7 ActiveX Test Object Syntax

The TPD syntax for an ActiveX DLL test object is similar to that for referencing a regular DLL. The file name is placed in the *Test Executable* field and the *Arguments* field contains the function name and any input parameters. A TPD entry for an ActiveX function must include the associated project and class. For example, if you want to reference "MyFunction" in "MyClass" that was built in "MyProject", the entry would be:

```
Test = 2.5.; My ActiveX; ; ; ; MyX.dll; MyProject.MyClass::MyFunction(/ad 15);
```

Note that the double colons (::) between the class name and function name are very important for identifying this as an ActiveX DLL.

8. TEXDLL Function Definitions

This section lists the TEXDLL functions in alphabetical order. Each function entry includes a description, Visual Basic and C syntax of the function, a description of each parameter, and possible error codes.

Constants and function declarations are provided in TEXDLL32.H and TEXDEF32.BAS. The function declarations in these files have compile options for selecting between 16 and 32-bit development. Also provided are an object library for linking (TEXDLL32.LIB) and a CVI function panel (TEXDLL.FP).

The easiest way to begin using these functions is to start with one of the projects in the SAMPLES subdirectory. Several different programming environments have representative projects there. First you should compile and execute a sample project without changes. This verifies the operation of your development environment and the installation of the Test Executive. Then you can modify the sample project to meet your initial needs. Small incremental steps are highly recommended for this process.

Note: The key to modern development environments is the project, or "make", file. These define the several hundred options required to successfully compile, link and execute a program. While it is possible to correctly select each of these options yourself, it is much better to start with a working project file and modify it as necessary.

8.1 texAbort

VB Function texAbort (ByVal *abortDelay*%, ByVal *appName*$, ByVal *appReason*$**)**

C short texAbort (short *abortDelay*, const char **appName*, const char **appReason***);**

Parameter	I/O	Description
abortDelay	in	Amount of time to wait for test object (seconds).
appName	in	Name of the application requesting the abort.
appReason	in	Reason that the application is requesting an abort.

Return Value

The function returns these constants to indicate how the abort request was handled:

Value	Description
TEX_ABORT_TE	Handled completely by Test Executive.
TEX_ABORT_DLL	Handled completely by TEXDLL.
TEX_ABORT_DLL_TE	Handled by TEXDLL & Test Executive.
TEX_ABORT_NONE	No active test object or Test Executive.

Remarks

This function causes the Test Executive to abort the current test object and execute the specified abort objects. This function is intended for external applications only, it should **not** be called by a test object. Test objects can initiate an abort by issuing a texFinish(ABORT_FINISH) and terminating themselves.

The first input parameter controls how much time is permitted for the test object to shut itself down (following abort notification). If it is negative, it is ignored and the abort delay is the default, one second, or whatever has been set by texSetAbortDelay(). The other input parameters allow the external application to identify itself and the reason for the abort. This information is added to the Test Results window when the abort occurs.

The return values from texAbort identify how the abort request was handled. If the Test Executive is active, the abort objects are executed once the active test object (if any) has been aborted. If the Test Executive is not present, the test object is terminated by the TEXDLL

8.2 texCompare

VB Function **texCompare (**ByVal *testParam*#, ByVal *description*$, ByVal *id*$,
ByVal *limitType*%, ByVal *limit1*#, ByVal *limit2*#,
ByVal *sigDigits*%, ByVal *prefix*$, ByVal *units*$**)**

C short **texCompare (**double *testParam*, const char **description*,
const char *id, short *limitType*, double *limit1*, double *limit2*,
short *sigDigits*, const char **prefix*, const char **units***);**

Parameter	I/O	Description
testParam	in	This test value is evaluated by the compare.
description	in	A descriptive, user-friendly name for the compare. Empty strings or NULL pointers raise a warning.
id	in	A unique identifier for the compare. This string can only contain alphanumeric characters and underscores. Illegal characters, empty strings or NULL pointers cause the verify to fail and are reported as errors.
limitType	in	Determines what test method is used. Legal constant values for this parameter are defined in the file (TEXDLL.H).
limit1	in	The usage of *limit1* depends upon *limitType* (see remarks).
limit2	in	The usage of *limit2* depends upon *limitType* (see remarks).
sigDigits	in	Number of significant digits applied to the test criteria. Legal values for this parameter are zero through six.
prefix	in	The order of magnitude of the units of measurement (e.g. “milli”). Empty strings and NULL pointers are allowed.
units	in	The units of measurement for the compare (e.g. “volts”). A value of “hex” displays the compare using hexadecimal format. Empty strings and NULL pointers are allowed.

Return Value

The function returns these constants:

Value	Description
TRUE	The compare passed.
FALSE	The compare failed.

Remarks

This function compares a parametric test value to a criteria supplied by the test object. The return value is the pass/fail result of this evaluation. Both texVerify() and texCompare() accept identical parameters. Given the same input, they both return identical evaluations. The differences are that:

1. Compares do not increment the verify count for the test,
2. Compares are not reported to the TestExec log file,
3. Compare results do not influence the outcome of a test.

Eight different test methods are available. The *limitType* parameter determines which method is used. The following constant values are defined in the file (TEXDLL.H).

Value	**Description**
EQUAL	Passes only if *testParam* is equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
GR_THAN	Passes only if *testParam* is greater than *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
GR_THAN_EQ	Passes only if *testParam* is greater than or equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
LESS_THAN	Passes only if *testParam* is less than *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
LESS_THAN_EQ	Passes only if *testParam* is less than or equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
RANGE	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are *limit1* and *limit2* , respectively. Significant digits are applied to *testParam* and both bounds.
NOM_TOL	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are based on a nominal tolerance: The lower bound is *limit1* - *limit2*. The upper bound is *limit1* + *limit2*. Significant digits are applied to *testParam* and both bounds.
NOM_PER	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are based on a nominal percentage: The lower bound is *limit1* - (| *limit1* | * *limit2*). The upper bound is *limit1* + (| *limit1* | * *limit2*). Significant digits are applied to *testParam* and both bounds.
BIN_COMP	Passes only if *testParam* logically ORed with *limit2* is equal to *limit1* logically ORed with *limit2* (significant digits are not used).
PASS_FAIL	Passes only if *testParam* is non-zero (*limit1*, *limit2*, and significant digits are not used).

8.3 texCompareTOD

VB Function texCompareTOD (ByVal *testParam*#, ByVal *id*$**)**

C short texCompareTOD (double *testParam*, const char **id***);*

Parameter	I/O	Description
testParam	in	This test value is evaluated by the compare.
id	in	This value uniquely identifies one verify from a TOD file. This string can only contain alphanumeric characters and underscores. Illegal characters, empty strings or NULL pointers cause the verify to fail and are reported as errors.

Return Value

The function returns these constants:

Value	Description
TRUE	The compare passed.
FALSE	The compare failed.

Remarks

Just like texCompare(), this function applies a comparison criteria to a parametric test value. The return value is the pass/fail result of this evaluation. The difference for texCompareTOD() is that criteria data is not provided by the function parameters.

All TOD-based functions use criteria data read from a TOD input file. The texCompareTOD() function always uses criteria data for the TOD verify specified by the given *id* parameter. If no TOD data matches the specified *id*, an error box is raised and the function fails.

8.4 texEmbedBreakpoint

VB Sub texEmbedBreakpoint (ByVal *displayMessage*$, ByVal *breakType%***)**

C void texEmbedBreakpoint(const char **displayMessage*, short *breakType***);**

Parameter	I/O	Description
displayMessage	in	This breakpoint message is reported to the TestExec log file and displayed in a breakpoint dialog box. An empty string or a NULL pointer are legal values for this parameter. The string can have up to 1024 characters, though the dialog box contents may be limited by screen resolution.
breakType	in	Specifies the breakpoint type. Constants for legal *breakType* values are defined in TEXDLL.H. They include:

Break Type	Purpose
TEX_BRK_DATA	Data breakpoints provide an opportunity to view intermediate measurements during a test.
TEX_BRK_TRACE	Trace breakpoints are used to indicate the path of execution during the test.

No Return Value

Remarks

This function is used to define breakpoints within the test object code. The behavior of each breakpoint type is controlled independently using the following command flags (also see *Behavior Modes*) or the Test Executive's Breakpoint Control window. By default, both breakpoint types are disabled at the start of a test.

Flag	Behavior
Data Breakpoints	
"/bd"	Enable dialogs and log entries for all data breakpoints.
"/lbd"	Only enable log entries for all data breakpoints.
"/nbd"	Disable data breakpoints (default).
Trace Breakpoints	
"/bt"	Enable dialogs and log entries for all trace breakpoints.
"/lbt"	Only enable log entries for all trace breakpoints.
"/nbt"	Disable trace breakpoints (default).

8.5 texErrorBox

VB Sub texErrorBox (ByVal *displayMessage*$**)**

C void texErrorBox (const char **displayMessage***);*

Parameter	I/O	Description
displayMessage	in	This error message is reported to the TestExec log file and displayed in an error dialog box. An empty string or a NULL pointer are legal values for this parameter. Custom dialog titles are specified by appending a title string to the message string according to the following format:

Format	Example
"*message* | *title*"	"This is a message|This is a title"

No Return Value

Remarks

This function makes an entry in the TestExec log file using the keywords (BREAK: ERROR;). An error dialog box is also raised showing the *displayMessage*, an exclamation mark, an OK button and either the default caption: (TEX ERROR) or a custom title. This error box behavior of the TEXDLL cannot be disabled.

Errors generated by the TEXDLL are distinguished from test-generated errors by the dialog caption (TEXDLL ERROR); and by the keywords (BREAK: DLL_ERROR;) in the TestExec log file.

This function can display a title of approximately 80 characters in length. Note that the specific length of the title is dependent upon the operating system, screen resolution and font selection (i.e. large or small). The maximum size for *displayMessage* is 1024 characters, though how much of this is displayed is also dependent on system parameters. Some formatting of the message is possible by embedding carriage returns and linefeeds in the text.

Note: This suspends code execution, in the test object, until acknowledged by an operator. If a hazardous condition is detected, the test object code should always attempt to correct the situation (e.g. turn off power) before notifying the operator.

8.6 texFinish

VB Sub texFinish (ByVal *testStatus%***)**

C void texFinish (short *testStatus***);**

Parameter	I/O	Description
testStatus	in	Specifies the test object status at the time of termination. Constants for legal *testStatus* values are defined in TEXDLL.H. They include:

Value	Description
NORMAL_FINISH	Test closed normally.
HALT_FINISH	Test quitting early, test sequence stops.
ERROR_FINISH	Test object detected an error condition and is ending the test early.
ABORT_FINISH	Test object detected a problem and asks Test Executive to execute abort setup objects.
RELOAD_FINISH	Requests that the Test Executive reload the TPD file and restore the test status.
RESET_FINISH	Requests that the Test Executive reload the TPD file and reset the test status.

No Return Value

Remarks

This function signals the end of a test or setup object to theTEXDLL. To record test completion and signal the Test Executive to display/record a pass/fail result, a test must call texFinish() with a *testStatus* of NORMAL_FINISH. If a test exits without calling texFinish(), it is assumed to have ended with a status of ERROR_FINISH and the operator is notified.

Negative *testStatus* values are used to indicate specific error numbers. Test objects should not use positive *testStatus* values other than the constants shown here. Either type of ERROR_FINISH causes an error dialog box to be presented to the user. This dialog is displayed as a "topmost" window so it can't be hidden by other applications. Prior to presenting the error dialog, all PostSequence setup objects are executed.

A *testStatus* value of ABORT_FINISH requests that the Test Executive perform its normal abort processing. This includes halting the test sequence and executing each of the abort setup objects defined in the TPD file. A *testStatus* value of RELOAD_FINISH requests that the Test Executive reload the current TPD. This can be used to recover from abort processing, or to reconfigure the test set to match the attributes of a specific UUT. RESET_FINISH requests the same sequence of events except it resets test object status rather than restoring it. For more information, see the *TPD Reload* section.

Note: Test and setup objects should always begin and end their use of the TEXDLL with one call to texStart() and one call to texFinish(). If this function is called before texStart(), an error is reported and the function does nothing.

8.7 texGetTODVerify

VB Function **texGetTODVerify(**ByVal *id*$, ByVal *description*$,
ByRef *limitType*%, ByRef *limit1*#, ByRef *limit2*#,
ByRef *sigDigits*%, ByVal *prefix*$, ByVal *units*$**)**

C short **texGetTODVerify(**char **id*, char **description*, short **limitType*,
double **limit1*, double **limit2*, short **sigDigits*,
char **prefix*, char **units***);**

Parameter	I/O	Description
id	in/out	A unique identifier for retrieving verify parameters from the current TOD file. Sequential access is provided by two special ids ("**Get-First**", "**Get-Next**"). This parameter is **always** overwritten so it must be a string pointer, not a literal string (i.e. string constant or macro). The identifier string must be at least 40 characters in length.
description	out	Returned description of the specified verify. Receiving string must be at least 40 characters in length.
limitType	out	Returned verify test method (see remarks for texVerify).
limit1	out	Returned *limit1* value (see remarks for texVerify).
limit2	out	Returned *limit2* value (see remarks for texVerify).
sigDigits	out	Returned number of significant digits (0-6) to be applied to the test criteria.
prefix	out	Returned order of magnitude (e.g. “milli”) for the verify. Receiving string must be at least 8 characters in length.
units	out	Returned units of measurement for the verify (e.g. “volts”). Receiving string must be at least 20 characters in length.

Return Value

Returns TRUE if the verify is located and read from the TOD file. Returns FALSE if the verify doesn't exist, a TOD file cannot be located or a test is not active (i.e. texStart() has not been previously executed).

Remarks

This optional function provides access to the verify parameters defined in a Test Object Definition (TOD) file. A TOD file is used to isolate test limits from the test object that performs the measurements. A test object might use this function to retrieve verify parameters for measurement scaling or modification. For example, a retrieved verify description could be modified to include a timestamp that enables correlation to a telemetry stream. A subsequent measurement is then verified by passing it, along with the retrieved parameters, to the texVerify() function. Alternately, the verify id might be modified to accommodate PDEL logging for multiple test object executions caused by conditional branching.

The **Get-First** id returns the first verify in the TOD file, while **Get-Next** retrieves the verify that sequentially follows the last one read by texGetTODVerify(). This provides a way to consecutively access verify entries without knowing their individual unique identifiers.

The TOD file name is passed to the TEXDLL, by texStart(), via a command flag (/tod). During normal operation this command flag is supplied by the Test Executive (see *Behavior Modes*). For test object creation and debug, the TOD name can be included in the command line arguments defined by the program development environment (e.g. CVI, Visual C++, etc.).

Note: When operating from within a program development environment, the TOD file name must be specified with a full directory path. Without a full directory path this function is unable to locate the TOD file.

When calling this function from Visual Basic it is recommended to use a fixed length string for the string parameters. Otherwise, fill a string to the appropriate size with String$ or Space$.

Example

This example function receives a measured value and verify ID as inputs. It uses the ID to retrieve the verify parameters from the TOD file. The verify description is then altered to include a timestamp before the verify is executed. Note, TimeStr() is a function supported by National Instruments LabWindows/CVI.

```
int VerifyTODWithTime(double value, char *vid)
{
  unsigned short vType, sigDigits;
  double limit1, limit2;

  char id[TEX_MAX_ID_LEN+5], desc[TEX_MAX_DESCR_LEN+5];
  char prefix[TEX_MAX_PREFIX_LEN+5];
  char units[TEX_MAX_UNIT_LEN+5], buf[100];

  strcpy(id, vid);          // Make local copy for overwriting
  texGetTODVerify(id, desc, &vType, &limit1, &limit2, &sigDigits, prefix, units);

  // Add the time to the verify description
  sprintf(buf, "%s @%s", desc, TimeStr());
  return(texVerify(value, buf, id, vType, limit1, limit2, sigDigits, prefix, units));
}
```

8.8 texGetTPC

VB Function texGetTPC (ByVal *field*$, ByVal *buffer*$, ByVal *bufsize*%**)**

C short texGetTPC (const char **field*, char **buffer*, short *bufsize***);**

Parameter	I/O	Description
field	in	Specifies the Test Program Configuration entry to read and return in the buffer. The field identifiers match the keywords used in a TPC file.
buffer	out	During the function, this is filled with the entry referenced by the *field* parameter (everything to the right of the equal sign).
bufsize	in	Specifies the size of the output buffer. Up to (bufsize - 1) characters are returned in the output buffer, not counting the NULL terminator.

Return Value

Returns TRUE if the field is located and read from the TPC file. Returns FALSE if the field doesn't exist, the TPC file cannot be located or a test is not active (i.e. texStart() has not been previously executed).

Remarks

This optional function provides access to the contents of the Test Program Configuration (TPC) file. A TPC file stores current test configuration information that is viewed and edited via the Test Configuration dialog. Some test objects might use this information to modify test criteria. For example, revision B of a UUT could be tested differently from revision F. Other uses for this data include advanced or customized data logging.

The name and directory path of the TPC file is derived from the current TPD file name. The TPD file name is passed to the TEXDLL, by texStart(), via a command flag (/tpd). During normal operation this command flag is supplied by the Test Executive (see *Behavior Modes*). For test object creation and debug, the TPD name can be included in the command line arguments defined by the program development environment (e.g. CVI, Visual C++, etc.).

Note: When operating from within a program development environment, the TPD file name must be specified with a full directory path. Without a full directory path this function is unable to locate the TPC file.

When calling this function from Visual Basic it is recommended to use a fixed length string for *buffer$.* Otherwise, fill a string to the appropriate size with String$ or Space$.

8.9 texGetTPD

VB Function texGetTPD (ByVal *id*$, ByVal *field*%, ByVal *buffer*$, ByVal *bufsize*%**)**

C short texGetTPD (const char **id*, short *field*, char **buffer*, short *bufsize***);**

Parameter	I/O	Description
id	in	Specifies the Test Program Definition item to access. This is either a TPD section (e.g. "Identification") or test object identifier (e.g. "1.2.1" or "**Get-Current**").
field	in	Selects the field to return from the above specified TPD item. Some fields may contain sub-fields (e.g. TPD_TEST_NAME). Consult the TPD file format section for more information on the syntax and contents of TPD entries.
buffer	out	During the function, this is filled with the entry referenced by the *id* and *field* parameters.
bufsize	in	Specifies the size of the output buffer. Up to (bufsize - 1) characters are returned in the output buffer, not counting the NULL terminator.

Return Value

Returns TRUE if the section and field is located and read from the TPD file. Returns FALSE if the section or field doesn't exist, the TPD file cannot be located or a test is not active (i.e. texStart() has not been previously executed).

Remarks

This optional function provides access to the contents of the current Test Program Definition (TPD) file. A TPD file defines the overall behavior of the Test Executive and the operating characteristics of each test object. Some test objects might use this information for user prompts or customized data logging. The following numeric field constants are defined in the header file (TEXDLL32.H).

Id	Field
"Identification"	TPD_FILE_NAME TPD_PROGRAM_NAME TPD_PART_NUMBER TPD_REVISION
"PowerMonitor"	TPD_POWER_TIME
Test Id or Paragraph (e.g. "RamTest25c", "1.2", "Get-Current")	TPD_TEST_PARAGRAPH TPD_TEST_NAME TPD_TEST_TOD_FILE TPD_TEST_TOD_PN TPD_TEST_TOD_VERSION TPD_TEST_OBJECT_FILE TPD_TEST_OBJECT_ARG TPD_TEST_OBJECT_PN TPD_TEST_OBJECT_VERSION TPD_TEST_PREREQUISITES TPD_TEST_BRANCH

The TPD file name and a current test identifier (i.e. paragraph number) are passed to the TEXDLL, by texStart(), via command flags (/tpd and /tid). During normal operation these command flags are supplied by the Test Executive (see *Behavior Modes*). For test object creation and debug, these flags can be included in the command line arguments defined by the program development environment (e.g. CVI, Visual C++, etc.). The test identifier flag is only necessary if this function is to retrieve parameters for the current test (i.e. id = "Get-Current").

Note: When operating from within a program development environment, the TPD file name must be specified with a full directory path. Without a full directory path this function is unable to locate the TPD file.

When calling this function from Visual Basic it is recommended to use a fixed length string for *buffer$.* Otherwise, fill a string to the appropriate size with String$ or Space$.

8.10 texInputNumber

VB Function texInputNumber (ByVal *message*$, ByVal *buffer*$, ByVal *bufsize*%**)**

C short texInputNumber(const char **message*, char **buffer*, short *bufsize***);**

Parameter	I/O	Description
message	in	Specifies the prompt displayed in the Input Number dialog box. Custom dialog titles are specified by appending a title string to the message string according to the following format:

Format	Example
"*message* | *title*"	"This is a message|This is a title"

buffer	in/out	Input contents specify the default entry value. Output contents return the data entry string. Input and output strings in this buffer are always NULL terminated.
bufsize	in	Specifies the size of the output buffer. Up to (*bufsize* - 1) characters are returned in the output buffer, not counting the NULL terminator. The maximum input buffer size is 41 characters.

Return Value

Returns TRUE if the operator selected OK from the Input Number dialog box. In this case, the given *buffer* returns the string entered by the operator. Returns FALSE if the operator selected CANCEL, or closed the Input Number dialog box by some other means. In this case, the given *buffer* remains unchanged.

Remarks

This function is used to prompt the operator for numerical data entry. The data entered by the user is returned in the given *buffer*. The input is filtered to specify only decimal and hexadecimal values. Decimal values start with an optional plus or minus sign, followed by a series of one or more digits (0-9), optionally including one period. Hexadecimal values must start with an "0x", followed by one or more hexadecimal digits, (0-9, A-F).

The operator is prompted repeatedly until a legal input is provided; illegal or empty strings are not accepted. Note that a legal default value can be specified in *buffer* as an input parameter.

This function can display a title of approximately 40 characters in length. Note that the specific length of the title is dependent upon the operating system, screen resolution and font selection (i.e. large or small). The maximum size of *message* is 160 characters in length. Some formatting of the display is possible by embedding carriage returns and linefeeds in the text (up to 5 lines). Long lines are automatically wrapped at spaces in the text.

Note: When using a custom title with this function, it is important to indicate to the user that a numeric value is expected. Otherwise, some entries could be rejected without the user understanding why.

When calling this function from Visual Basic it is recommended to use a fixed length string for *buffer$.* Otherwise, fill a string to the appropriate size with String$ or Space$.

8.11 texInputString

VB Function texInputString (ByVal *message*$, ByVal *buffer*$, ByVal *bufsize*%**)**

C short texInputString (const char **message*, char **buffer*, short *bufsize***);**

Parameter	I/O	Description
message	in	Specifies the prompt displayed in the Input String dialog box. Custom dialog titles are specified by appending a title string to the message string according to the following format: **Format** "*message* | *title*" **Example** "This is a message|This is a title"
buffer	in/out	Input contents specify the default entry value. Output contents return the data entry string. Input and output strings in this buffer are always NULL terminated.
bufsize	in	Specifies the size of the output buffer. Up to (bufsize - 1) characters are returned in the output buffer, not counting the NULL terminator. The maximum input buffer size is 41 characters.

Return Value

Returns TRUE if the operator selected OK from the Input String dialog box. In this case, the given *buffer* returns the string entered by the operator. Returns FALSE if the operator selected CANCEL, or closed the Input String dialog box by some other means. In this case, the given *buffer* remains unchanged.

Remarks

This function is used to prompt the operator for text data entry. The data entered by the user is returned in the given *buffer*. An empty string is acceptable input. Note that a default value can be specified in *buffer* as an input parameter.

This function can display a title of approximately 40 characters in length. Note that the specific length of the title is dependent upon the operating system, screen resolution and font selection (i.e. large or small). The maximum size of *message* is 160 characters. Some formatting of the display is possible by embedding carriage returns and linefeeds in the text (up to 5 lines). Long lines are automatically wrapped at spaces in the text.

When calling this function from Visual Basic it is recommended to use a fixed length string for *buffer$.* Otherwise, fill a string to the appropriate size with String$ or Space$.

8.12 texLogPdel

VB Sub texLogPdel (ByVal *PdelMessage*$**)**

C void texLogPdel(const char **PdelMessage***e***);**

Parameter	I/O	Description
PdelMessage	in	A request to add this message to the PDEL file is reported to the TestExec log file. A PDEL file may or may not be active, based upon the state of the Test Executive. An empty string or a NULL pointer are legal values for this parameter.

No Return Value

Remarks

This function makes a PDEL entry in the TestExec log file using the keyword (PDEL:). This behavior of the TEXDLL cannot be disabled; however the Test Executive determines if a PDEL file is active or not.

Note: *PdelMessage* must be in a form that is valid for a PDEL file. This function does not add PDEL formatting to the parameter.

8.13 texMessageBox

VB Sub texMessageBox (ByVal *displayMessage*$**)**

C void texMessageBox (const char **displayMessage***e***);**

Parameter	I/O	Description
displayMessage	in	This message is reported to the TestExec log file and displayed in a dialog box. An empty string or a NULL pointer are legal values for this parameter. Custom dialog titles are specified by appending a title string to the message string according to the following format:

Format	Example
"*message* | *title*"	"This is a message|This is a title"

No Return Value

Remarks

This function makes an entry in the TestExec log file using the keywords (BREAK: MSGBOX;). A dialog box is also raised showing the *displayMessage*, an OK button and either the caption: (TEX DIALOG) or a custom title. This message box behavior of the TEXDLL cannot be disabled.

This function can display a title of approximately 80 characters in length. Note that the specific length of the title is dependent upon the operating system, screen resolution and font selection (i.e. large or small). The maximum size for *displayMessage* is 1024 characters, though how much of this is displayed is also dependent on system parameters. Some formatting of the message is possible by embedding carriage returns and linefeeds in the text.

8.14 texPowerMonitor

VB Function texPowerMonitor(ByVal *powerCmd* As Integer**) As Long**

C long texPowerMonitor(short *powerCmd***);**

Parameter	I/O	Description
powerCmd	in	Specify the state of UUT power (On/Off) or request information about the power state or time remaining. Constants for legal *powerCmd* values are defined in TEXDLL.H. They include:

Value	Description
TEX_POWER_OFF	Indicate UUT power is off.
TEX_POWER_ON	Indicate UUT power is on.
TEX_POWER_TIME	Request remaining time for UUT power.
TEX_POWER_STATE	Request current power state (on/off).

Return Value

The function returns TEX_POWER_OFF or TEX_POWER_ON when the input parameter is TEX_POWER_STATE. The other input parameters have a return value of the time remaining for UUT power (in seconds). If the return value is negative, that indicates the time remaining that the UUT needs to be powered off in order to recover from reaching its power limit.

Remarks

This function does **not** control power to the UUT. It simply provides a mechanism for test and setup objects to communicate the state of UUT power. This optional function is necessary when a UUT has strict limits on how long it can be powered without damage. The Test Executive manages the display of power status and handles overall tracking of UUT power limits and recovery (as defined in the TPD file). A test object can use this function to determine if it is safe to apply UUT power or if there is sufficient time available to complete the designated test. If a UUT's power limit is reached, an abort sequence is performed by the Test Executive. Note that one of the abort objects should always turn UUT power off and inform the Test Executive by executing a **texPowerMonitor**(TEX_POWER_OFF).

For more complex power-on restrictions, an optional external power management DLL is supported by the Test Executive.

8.15 texPrint

VB Sub texPrint (ByVal *displayMessage*$**)**

C void texPrint (const char **displayMessage***e***);**

Parameter	I/O	Description
*displayMessage*e	in	This message is displayed in the Test Results window of the Test Executive. An empty string or a NULL pointer are legal values for this parameter.

No Return Value

Remarks

This function make a print entry in the TestExec log file using the keyword (PRINT:). This print behavior of the TEXDLL cannot be disabled; however the Test Executive determines if print messages are displayed or not.

8.16 texScale

VB Function texScale(ByVal *inVal#*, *outVal#*, ByVal *prefix$*, ByVal *mode%*, ByVal *targetPrefix$***)**

C short texScale(double *inVal*, double far **outVal*, const char **prefix*, short *mode*, const char **targetPrefix***);**

Parameter	I/O	Description
inVal	in	The value to be scaled.
outVal	out	The result of the scale operation.
prefix	in	The source magnitude prefix.
mode	in	The following two scale modes are supported:

Scale Mode Value	Description
SCALE_DIRECT	The *targetPrefix* parameter directly supplies the target magnitude prefix string.
SCALE_TOD	The *targetPrefix* parameter supplies a unique verify ID string. The target magnitude prefix string is then read from the current TOD file.

targetPrefix	in	The target magnitude prefix or a verify ID string (see above).

Return Value

Returns TRUE if successful, otherwise returns FALSE.

Remarks

This function scales a given value from a source prefix magnitude to a target prefix magnitude. The scaled value is written to the given *outVal* address. The scale mode parameter specifies how the *targetPrefix* parameter is interpreted.

8.17 texSetAbortDelay

VB Sub texSetAbortDelay(ByVal *seconds%**)***

C void texSetAbortDelay(short *seconds***);**

Parameter	I/O	Description
seconds	in	The number of seconds assigned to the TEXDLL abort delay.

No Return Value

Remarks

This function sets the TEXDLL abort delay to a specified number of seconds. The abort delay is how long the TEXDLL waits for a test object to terminate itself following an abort request. The default delay is one second; and legal delay values range from 1 to 1800 seconds (30 minutes).

If the abort delay is >15 seconds a message indicating the delay is displayed in the Test Results window. When aborting a test object DLL, the Test Executive GUI is disabled and a message box with the delay value is displayed in response to a mouse click. When aborting a test object executable, the Test Executive GUI displays a countdown value of the remaining time until the abort completes.

The abort delay can be changed at any time during a test, so an appropriate delay can be in effect at all times. The abort delay automatically returns to the default value at the end of each test. This means that the abort delay is reset to one second by texStart() and texFinish().

A complete discussion of abort behavior is provided in sections *Aborting a Test*, *Abort Delay* and *DLL Abort*. The following command flag can also set the abort delay:

Flag	Behavior
Abort Delay "/ad" *delay*	Specify abort delay in seconds (default is 1s)

8.18 texStart

VB Function texStart (ByVal *commandFlags$*)**

C short texStart (const char **commandFlags*);**

Parameter	I/O	Description
commandFlags	in	A single string containing command flags that initialize TEXDLL behavior at the beginning of a test. See section *Behavior Modes* for more information about the source of this parameter.

Return Value

Returns TRUE if TEXDLL initialization was successful, otherwise returns FALSE. If texStart() fails, the test object should exit immediately. If it doesn't, the Test Executive will be unable to abort it and most of the TEXDLL functions will be inaccessible to it.

TEXDLL initialization fails if one or more problems are found in a TOD file or if a specified help file is missing. If warnings are enabled, a detailed list of problems are written to a file (TESTEXEC.LOG) or the Debug Log window.

Remarks

This function signals the beginning of a test or setup object to the TEXDLL and Test Executive. A test object must call texStart() before calling most other TEXDLL functions. Some user interface functions are exempt from this requirement (see below). If a TEXDLL function is called before texStart(), an error is reported and the function does nothing. Test or action objects can use texStartActive() to determine if a test has begun. This is useful when creating reusable code modules that may be used separately or with others.

The TEXDLL services one test at a time. It always stops what it's doing in order to service the newest client. If a second test calls texStart() while a previous test is still running:

1. The previous test is halted.
2. A warning is reported.
3. The second test is started.

Note: A test object should always begin and end it's use of the TEXDLL with one call to texStart() and one call to texFinish(). If this function is called more than once, the subsequent calls report an error and do nothing.

The Test Executive uses command line arguments to control a test object's behavior. The command line is passed from the test object to the TEXDLL via the *commandFlags* parameter in texStart(). The contents of the command line include the following list of flags for defining breakpoints, file names and a test id.

If conflicting flags are specified, the **leftmost** flag overrides all previous conflicting flags (as of Test Executive v6.2). This allows individual flags to be overridden by the arguments specified in a test object's TPD entry. See the section *Test Object Command Line Arguments* for more information.

If no flags are specified, the TEXDLL behavior is initialized using the default modes. Section *Behavior Modes* has additional information and sample code that demonstrates how to pass command line arguments to the TEXDLL.

Flag	**Behavior**
Verify Breakpoints	
"/va"	Break at all verifies.
"/vf"	Break at failing verifies.
"/v" *number* / **'***id***'**	Break at verify *number* or *id* (e.g. "/v 25" or "/v **'**vfy5**'** ")
"/vn"	Disable verify breakpoints. *
Data Breakpoints	
"/bd"	Enable dialogs and log entries for all data breakpoints.
"/lbd"	Only enable log entries for all data breakpoints.
"/nbd"	Disable data breakpoints. *
Trace Breakpoints	
"/bt"	Enable dialogs & log entries for trace breakpoints.
"/lbt"	Only enable log entries for all trace breakpoints.
"/nbt"	Disable trace breakpoints. *
Warnings	
"/w"	Enable dialogs and log entries for all warnings.
"/lw"	Only enable log entries for all warnings.
"/nw"	Disable warnings. *
File Names	
"/tod" *filename*	Specify TOD filename. (e.g. "/tod uut34581.tod")
"/tpd" *filename*	Specify Help filename. (e.g. "/tpd texdemo.tpd")
"/hlp" *filename*	Specify Help filename (e.g. "/hlp sample.hlp").
Test Id	
"/tid" *paragraph*	Specify current test entry paragraph number.
Abort Delay	
"/ad" *delay*	Specify abort delay in seconds (default is 1s)

* Denotes a default behavior mode.

In order to avoid a conflict between TEXDLL command flags and those used for a test object, it is strongly recommended that test object command flags be prefaced by "**/usr**" (e.g. /usrTest21). The Test Executive is guaranteed never to use command flags prefaced in this manner. Some command flags are solely intended for the test object. This includes an execution repeat count value. For more information, see the section on *Test Object Command Line Arguments*.

Note: Starting in version 2.2 of the TEXDLL, the functions: texEmbedBreakpoint(), texMessageBox(), texYesNoBox(), texWarningBox() and texErrorBox() can be used without calling texStart() or texFinish().

Programs that use the above functions, without calling texStart(), rely on the Test Executive to control the enable/disable modes for warnings and embedded breakpoints. When such a program is run stand-alone, warnings and embedded breakpoints are disabled by default.

8.19 texStartActive

VB Function texStartActive( )

C short texStartActive(void**);**

Parameter	I/O	Description
None		

Return Value

Returns TRUE if a test is active (i.e. texStart() has already been executed), otherwise returns FALSE. If the return value is FALSE, it is safe to execute texStart() without causing an error.

Remarks

This function determines if texStart() has been executed. This facilitates the creation of reusable code modules that may be used standalone or in combination with others. It also permits action objects (tools) to monitor whether a test is active or not. In some cases, it may be necessary for the action object to limit its behavior while tests are running.

8.20 texVerify

VB Function **texVerify(**ByVal *testParam*#, ByVal *description*$, ByVal *id*$,
ByVal *limitType*%, ByVal *limit1*#, ByVal *limit2*#,
ByVal *sigDigits*%, ByVal *prefix*$, ByVal *units*$**)**

C short **texVerify(**double *testParam*, const char **description*,
const char *id, short *limitType*, double *limit1*, double *limit2*, short
sigDigits, const char **prefix*, const char **units***);**

Parameter	I/O	Description
testParam	in	This test value is evaluated by the verify.
description	in	A descriptive, user-friendly name for the verify. Empty strings or NULL pointers raise a warning.
id	in	A unique identifier for the verify. This string can only contain alphanumeric characters and underscores. Illegal characters, empty strings or NULL pointers cause the verify to fail and are reported as errors.
limitType	in	Determines what test method is used. Legal constant values for this parameter are defined in the file (TEXDLL.H).
limit1	in	The usage of *limit1* depends upon *limitType* (see remarks).
limit2	in	The usage of *limit2* depends upon *limitType* (see remarks).
sigDigits	in	Number of significant digits applied to the test criteria. Legal values for this parameter are zero through six.
prefix	in	The order of magnitude of the units of measurement (e.g. “milli”). Empty strings and NULL pointers are allowed.
units	in	The units of measurement for the verify (e.g. “volts”). A value of “hex” displays the verify using hexadecimal format. Empty strings and NULL pointers are allowed.

Return Value

The function returns these constants:

Value	Description
TRUE	The verify passed.
FALSE	The verify failed.

Remarks

This function verifies a parametric test value with respect to a criteria supplied by the test object. The return value is the pass/fail result of this evaluation. Both texVerify() and texCompare() accept identical parameters. Given the same input, they both return identical evaluations. The differences are that:

1. Each verify increments the verify count for the test.
2. Verifies are reported to the TestExec log file.
3. Verify results determine the outcome of a test. One or more failing verifies means the test has failed.

Ten different test methods are available. The *limitType* parameter determines which method is used. The following constant values are defined in the file (TEXDLL32.H).

Value	**Description**
EQUAL	Passes only if *testParam* is equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
GR_THAN	Passes only if *testParam* is greater than *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
GR_THAN_EQ	Passes only if *testParam* is greater than or equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
LESS_THAN	Passes only if *testParam* is less than *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
LESS_THAN_EQ	Passes only if *testParam* is less than or equal to *limit1*. Significant digits are applied to *testParam* and *limit1* (*limit2* not used).
RANGE	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are *limit1* and *limit2* , respectively. Significant digits are applied to *testParam* and both bounds.
NOM_TOL	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are based on a nominal tolerance: The lower bound is *limit1* - *limit2*. The upper bound is *limit1* + *limit2*. Significant digits are applied to *testParam* and both bounds.
NOM_PER	Passes only if *testParam* is within an inclusive range whose lower and upper bounds are based on a nominal percentage: The lower bound is *limit1* - (| *limit1* | * *limit2*). The upper bound is *limit1* + (| *limit1* | * *limit2*). Significant digits are applied to *testParam* and both bounds.
BIN_COMP	Passes only if *testParam* logically ORed with *limit2* is equal to *limit1* logically ORed with *limit2* (significant digits are not used).
PASS_FAIL	Passes only if *testParam* is non-zero (*limit1*, *limit2*, and significant digits are not used).

8.21 texVerifyTOD

VB Function texVerifyTOD (ByVal *testParam*#, ByVal *id*$**)**

C short texVerifyTOD(double *testParam*, const char **id***);**

Parameter	I/O	Description
testParam	in	This test value is evaluated by the verify.
id	in	This value uniquely identifies one verify from a TOD file. This string can only contain alphanumeric characters and underscores. Illegal characters, empty strings or NULL pointers cause the verify to fail and are reported as errors.

Return Value

The function returns these constants:

Value	Description
TRUE	The verify passed.
FALSE	The verify failed.

Remarks

Just like texVerify(), this function applies a test criteria to a parametric test value. The return value is the pass/fail result of this evaluation. The difference for texVerifyTOD() is that criteria data is not provided by the function parameters.

All TOD-based functions automatically use criteria data read from a TOD input file. The texVerifyTOD() function always uses criteria data for the TOD verify specified by the given *id* parameter. If no TOD data matches the specified *id*, an error box is raised and the function fails. See also *Test Object Definition (TOD) Files*.

Each test object has an upper limit of 1000 verifies.

As an alternate to the texVerifyTOD() function, verify parameters can be retrieved with the texGetTODVerify() function and processed by texVerify(). This allows the verify parameters to be logged or modified before the verify is performed. For example, a timestamp embedded in a verify description makes it easier to correlate a measurement to a telemetry stream. Or, the prefix and units parameters could be set to the failing string following a string comparison.

8.22 texWarningBox

VB Sub texWarningBox (ByVal *displayMessage*$**)**

C void texWarningBox (const char **displayMessage***e***);**

Parameter	I/O	Description
displayMessage	in	This message is reported to the TestExec log file and displayed in a warning dialog box. An empty string or a NULL pointer are legal values for this parameter. Custom dialog titles are specified by appending a title string to the message string according to the following format:

Format	Example
"*message* | *title*"	"This is a message|This is a title"

No Return Value

Remarks

The TEXDLL provides three modes of warning behavior:

1. When fully enabled (using command flag "/w"), this function makes a warning entry in the TestExec log file using the keywords (BREAK: WARNING;). A dialog box is also raised showing the *displayMessage*, an exclamation mark, an OK button and a default caption (TEX WARNING) or custom title.

2. When enabled for logging only (using the command flag "/lw"), this function only makes a warning entry in the TestExec log file using the keywords (LOG: WARNING;). No dialog box is raised, but warnings are recorded in the TestExec log file. In this mode, test execution is continuous and the user is not required to acknowledge warnings.

3. When disabled (using the command flag "/nw"), this function does nothing.

Warnings generated by the TEXDLL are distinguished from test-generated warnings by the dialog caption (TEXDLL WARNING); and by the log entry keywords (BREAK: DLL_WARNING;). The modal behavior of TEXDLL warnings is always the same as the behavior specified for test-generated warnings.

This function has the same size and formatting capacity as texMessageBox(). For more information see texMessageBox.

8.23 texYesNoBox

VB Function texYesNoBox (ByVal *displayMessage*$**)**

C short texYesNoBox (const char **displayMessage***);*

Parameter	I/O	Description
displayMessage	in	This message is reported to the TestExec log file and displayed in a YesNo dialog box. An empty string or a NULL pointer are legal values for this parameter. Custom dialog titles are specified by appending a title string to the message string according to the following format:

Format	Example
"*message* | *title*"	"This is a message|This is a title"

Return Value

A True is returned if the YES button is clicked; and a False is returned if the NO button is clicked.

Remarks

This function makes an entry in the TestExec log file using the keywords (BREAK: YESNOBOX;). A dialog box is also raised showing *displayMessage*, a question mark, a YES button, a NO button, and a default caption (TEX DIALOG) or custom title. This YesNo behavior of the TEXDLL cannot be disabled.

This function can display a title of approximately 80 characters in length. Note that the specific length of the title is dependent upon the operating system, screen resolution and font selection (i.e. large or small). The maximum size for *displayMessage* is 1024 characters, though how much of this is displayed is also dependent on system parameters. Some formatting of the message is possible by embedding carriage returns and linefeeds in the text.

9. Appendix A - Frequently Asked Questions

1. What language should I build my test, setup and action objects with?

   Use whatever language is appropriate for the job. Test and setup objects must be able to interface to a DLL. You may even want to use several different languages. For example, an action object with a user interface might be written in Visual Basic while a test object might be written in C for faster execution.

2. How do I keep the user from breaking at a critical point in the test?

   Do not call breakpoint or verify functions during critical times in the test. Store your measurements and verify them after the critical point has passed.

3. Can I have more than one TPD file in the same directory?

   Yes, this is an appropriate way to share common test objects, setup objects, action objects and TOD files between similar TPDs. Action and setup objects can also be placed in other, common directories that are specified in the system execution path.

4. I have too many test objects and TOD files, how can I reduce the number I need?

   Maintenance and tracking are greatly enhanced by having fewer files. One way to accomplish this is to combine the functionality of several test objects into one executable. The required test object behavior can then be specified as a command line argument to the executable. This approach preserves the granularity of numerous test objects without the confusion of numerous executables. Also, verify data for several test objects can be concatenated into one TOD file.

5. Why aren't Warning breakpoints displayed from my IDE?

   When executing from an Interactive Development Environment (IDE), Warning breakpoints can be enabled by a command line switch (/w). Most IDEs have a way to specify command line arguments for the program being executed. This is a particularly useful mechanism during debug because breakpoint behavior can be modified from the command line without recompiling the program. When running stand-alone (i.e. without the Test Executive) warnings are logged to a file (TESTEXEC.LOG) if enabled (/w or /lw).

6. How do I get a log of trace and data points without running the Test Executive?

   Breakpoints and logging are controlled through the test object's command line (see related question above). If logging is enabled, and no Test Executive is active, the information is written to a file (TESTEXEC.LOG) in the test object's home directory.

7. How do I share data between test objects?

   There are several ways to share data between test objects. The simplest is to write the data to a file and access it from other test objects. This can be easily accomplished for small amounts of data by using the Windows API functions **WritePrivateProfileString** and **GetPrivateProfileString**. These functions work with files that are formatted similarly to Windows INI files.

   Another way to share data is to create a DLL to serve as a data repository during the testing sequence. When a DLL is specified as a presequence setup object it is kept loaded throughout the test sequence. This allows the DLL to store data from one test object and share it with the next.

8. What's the difference between Warnings and Errors?

   The only difference between Warnings and Errors is one of usage. Warnings typically notify the operator of unusual or unexpected conditions that still permit the program to continue (e.g. Rebuilding missing file). Errors are used for more serious conditions that require attention from the operator and that indicate the program cannot continue (e.g. The power supply is shorted to ground). Warnings and Errors are always displayed in the Debug Log. Warning breakpoints (i.e. message boxes) can be disabled through the Breakpoint Control window. Error breakpoints are never disabled.

9. What is the difference between Data and Trace breakpoints?

   Typically, Trace breakpoints are used to track the execution flow through various functions and modules in a program. Data breakpoints are used to view intermediate values of measurements or calculations. This distinction is entirely at the discretion of the individual program developer.

10. What's the difference between Abort and Quit?

    When a test sequence is running, or looping, the Quit button is used to halt the sequence following the completion of the currently executing test object. The Abort button is used to interrupt and cancel the current test object; followed by the sequenced execution of all the abort setup objects defined in the TPD file.

11. How do I implement a "Greater Than or Equal To" (>=) verify?

    Version 6.0 of the Test Executive now supports "Greater Than or Equal To" and "Less Than or Equal To" verify types. Use the constants GR_THAN_EQ and LESS_THAN_EQ that are defined in the supplied header files.

12. How do I verify strings?

    First, compare the strings using whatever mechanism is available for your language of choice. Then execute a PASS_FAIL **texVerify** with the results of the comparison. Note that the verified string can be logged as part of the verify description, or as the prefix and units parameters. For a verify from a TOD file, use the description field to store the expected string and retrieve it with the **texGetTODVerify** function.

13. How can I find out a test object's name and paragraph number for display?

    One way to accomplish this is to pass the information in on the test object's command line. Another way is to use the **texGetTPD** function to retrieve the necessary information from the TPD file. Note that some of the data returned by this function may contain multiple fields delimited by commas.

14. What do I do if I don't need all of the data defined in the TPC window?

    Almost all of the fields in the Test Configuration window can be made static, or disabled, by entries in the TPD file. The [ConfigurationInformation] section of the TPD file controls the behavior of the TPC data. Data encapsulated by angle brackets (e.g. UUTRevision = <A>) is made static and uneditable in the window. Fields are disabled by setting them to N/A (e.g. TestSetMSN = N/A).

15. Can action objects use the interface functions?

    Starting in version 4.0 of the Test Executive (version 2.2 of the TEXDLL), embedded breakpoint, message, warning, and error dialog boxes are available outside the context of a test. This means that the functions: texMessageBox(), texEmbedBreakpoint(), texYesNoBox(), texWarningBox() and texErrorBox() can be used without calling texStart() or texFinish(). Programs that do not call texStart() rely on the Test Executive to control the enable/disable modes for warnings and embedded breakpoints. When such a program is run stand-alone, warnings and embedded breakpoints are disabled by default.

16. How can I force the tests to run in a particular order?

    To make the tests run in a specific order, each test should have a prerequisite of the preceding test. This forms a "linked list" of tests that can only be run in the defined sequence. Also, it is best to disable the multiple selection option for the test list (MultiSelect = N).

17. How do I embed version information in my test objects?

    Version information is added to an executable through the use of a resource compiler. The Visual C++ sample included with the Test Executive has version information declared in a separate resource file. For Visual Basic 3.0, third party controls are available for embedding version information in an executable (e.g. VersionStamper by Desaware). Visual Basic 4.0 directly supports creating executables with embedded version information.

18. How can my test object detect an abort condition?

    When an abort occurs the test object is sent a WM_CLOSE message to all of its top-level windows. This message indicates that the test object is being halted because of an abort. For more information, see *Aborting a Test*. Test object DLLs have a different abort notification mechanism. See *DLL Loading and Execution*.

19. Sometimes texPrint messages seem to be queued and are not displayed until after a long delay completes in my code.

    Many of the TEXDLL operations are queued in order to allow the test program to execute as quickly as possible. Consequently, the Test Executive requires processing time to consume and display a texPrint message. Two texPrint messages in a row causes the test program to wait until the Test Executive can process the first message. Note that Windows 3.X is a non-preemptive operating system. This means that each application must release time to the system in order to allow other applications to operate. Executing for long periods without releasing time is considered a sign of a poorly behaved program.

20. Will there be any impact when dates go beyond 1999?

    All dates are manipulated and stored using Microsoft components (i.e. date variables and database table entries). These have been tested to operate with dates past 1999.

21. The MSN entry in the test configuration window does not accept underscore's.

    If a format mask is defined for the MSN in the TPD file, underscores are not accepted as part of the MSN. Underscores are allowed if no mask is specified.

22. How do I mouselessly move from the Test Result window to the Test Executive?

    Function keys **F8** and **F9** toggle between the Test Results and Debug Log Windows respectively.

23. How can I enable a subset of breakpoint types?

    Additional control over breakpoints can be achieved by adding conditional execution statements to the test program:

    ```
    if ( TraceLevel > 3 )
        texEmbedBreakpoint("Connection 34",TEX_BRK_TRACE)
    ```

    The TraceLevel value can be set by a command line argument or user prompt (e.g. texInputNumber) when breakpoints are active. The breakpoint levels can be determined by examining the test program's command line arguments.

24. I need to test every location in a 512K RAM. How do I bypass the 1000 verify limit?

    You don't. The purpose of the verify limit is to prevent excessive logging of insignificant information. Certainly 512K verifies for one device falls in that category. Note that each verify creates an Access database record (if enabled) and a PDEL entry (if enabled). Therefore, each verify has an associated cost in execution time, storage space and post-processing time/space.

    Under these circumstances it is important to understand the difference between testing and a verify function call. Besides performing a comparison on a measurement, a verify function also logs and displays the corresponding information. A device can be completely tested without a single verify function call. Therefore, if a specification calls for testing every location in a RAM, it doesn't mean that every location must be logged with a verify function call.

The simplest way to deal with a RAM is to test it and report the result with a PASS_FAIL verify type. If desired, texCompare() can be used for the testing. If more granular reporting is required, multiple verifies can report the results of each subtest (e.g. walking ones, walking zeros, checkerboard, etc.). If further granularity is desired, verifies can be added for each phase of a subtest (e.g. A12 walking ones - PASS, D3 walking zeros - FAIL, etc.). Verifies such as these provide more information, in less space, than simply verifying every location in a RAM.

25. How do I force the operator to enter TPC data each time a TPD file is opened?

    Information entered in the Test Configuration window is stored in a companion file to the TPD (with extension TPC). This data is automatically reloaded when the TPD file is opened. Deleting the TPC file forces this data to be reentered. This can be accomplished automatically by creating a PostLoad setup object that deletes the TPC file each time the TPD is closed by the Test Executive. Alternately, the setup object could just delete a few items in the file to force the reentry of specific data fields (e.g. Work Order and Operation). The Windows API function **WritePrivateProfileString** is aptly suited for this. For maximum reuse, the setup object should support a command line argument that specifies the TPC file to modify.

26. My test objects are running slower than I expected, what can I do?

    Many things affect the execution speed of test objects including: system configuration, memory size, data logging, the number of other applications running and the instrumentation being used. There are a few standard steps that can be taken to improve execution speed.

    The first step is to load DLLs that are common to your test objects with a PreLoad setup object. Most development tools have one or more DLLs that must be loaded for a program to execute (e.g. VB40032.DLL for Visual Basic, MFC40.DLL for Visual C++). Eliminating the DLL load/unload for each test object can save significant time during execution.

    The next step is to reduce the number of tasks the Test Executive has to perform by turning off any unnecessary breakpoint logging and data output (i.e. PDEL and/or Database). It's possible for the test object to be delayed while the Test Executive "catches up". Also, close or reduce the size of the Test Result window because updating its scrolling display consumes considerable processor time.

27. How do I share memory between processes in a 32-bit DLL?

    Each time a 32-bit DLL is loaded by a program, the DLL creates a new memory area for the exclusive use of the calling process. If several processes want to share information via the DLL, a shared memory area must be created. This can be accomplished in a DLL with a file-mapping object. Use the Windows API function **CreateFileMapping** to create and manage a block of named, shared memory. For further information on this, consult the Microsoft Developer Network or check out Microsoft's web page.

28. How do I make the Test Executive load a specific TPD file at startup?

    Pass the TPD file name to the Test Executive as a command line argument.

29. How can I use TOD files if I need to change some of the verify parameters?

Some test situations require dynamic changes to verify parameters. For example, a timestamp embedded in a verify description makes it easier to correlate a measurement to a telemetry stream. To accomplish this with a TOD file, use texGetTODVerify to first retrieve the verify parameters from the TOD file. Then the parameters can be modified before the verify is performed by calling the texVerify function.

10. Appendix B - Units and Prefixes

The texVerify functions supported by the TEXDLL allow prefix and unit strings to be specified. These strings are used when displaying and storing the results of the verify. The supplied program header files have constant definitions for prefixes and units that are recognized by the PDEL standard. The tables below list the PDEL recognized parameters and the abbreviations used when they are displayed. If no abbreviation is indicated, or the parameter is unrecognized, it is displayed capitalized. If the units string is specified as "HEX" the numeric parameters for the verify are displayed in hexadecimal notation.

Prefix	**Abbreviation**
EXA	
PETA	
TERA	T
GIGA	G
MEGA	M
KILO	K
HECTO	
DEKA	
DECI	d
CENTI	c
MILLI	m
MICRO	u
NANO	n
PICO	p
FEMTO	f
ATTO	a
NONE	-

Unit	Abbreviation
AMPERE	A
HENRY	H
HERTZ	Hz
OHM	
SIEMENS	S
VOLT	V
WATT	W
FARAD	F
CELCIUS	C
FAHRENHEIT	F
KELVIN	K
FOOT	ft
INCH	in
METER	m
MILE	
NAUTICAL_MILE	Nautical Mile
SECONDS	s
BEL	B
DEGREE	Deg
RADIAN	Rad
OUNCE	Oz
POUND	lb
GRAM	g
LUX	lx
LUMEN	lm
FOOT_CANDLE	Foot Candle
MAXWELL	
WEBER	Wb
BAR	
PSI	
LITER	l
NONE	-

11. Appendix C - Problems and Notes

11.1 Report Generator & Privilege Editor Crash on Exit

February 22, 2001

Several Windows NT 4.0 users have reported that the Report Generator and/or Privilege Editor crash when they are closed. The error typically appears as a message box that states:

> **The instruction at "0x04059ff9" referenced memory at "0x04059ff9".**
> **The memory could not be "read".**

The exact addresses may be different, and some systems may give other error messages. Except for the annoyance factor, this crash appears to do no harm.

This error is described in Microsoft's Knowledge base (ID: Q164819) and the problem is attributed to a database DLL (MSJT3032.DLL). The solution described is to update to version 3.00.3213 of this DLL. The Test Executive v6.0 installs version 3.00.5226 of MSJT3032.DLL.

This problem has recently reappeared, and the cause has been traced to updates to the Microsoft C Runtime Libraries (MSVCRT.DLL and MSVCIRT.DLL). These two DLLs are now at version 6.0, which corresponds to Visual Studio 6. So far, Microsoft has not acknowledged the reintroduction of this error.

A fix that has been successful on a couple of systems is to use a slightly earlier version of the database DLL. MSJT3032.DLL version 3.00.3305 was installed with Test Executive v5.3 and it works with the latest versions of the C Runtime Libraries (MSVCRT.DLL v6.00.8797 and MSVCIRT.DLL v6.00.8168). The latest versions of the C Runtime Libraries can be retrieved from Microsoft's web site (search for "VC6 Redistribution Library").

The installation for Test Executive v6.2 includes the earlier version of MSJT3032.DLL. This DLL is installed in the Windows System directory as well as the Test Executive home directory. This should ensure that the Report Generator and Privilege Editor use the correct version of this DLL.

11.2 Database "Out of Memory" Error

Some users have reported that the Test Executive and Privilege Editor generate an "**Out of Memory**" error when trying to access a database. This typically occurs on computers that have more that 512MB of random access memory (RAM). The amount of memory that causes this error is dependent upon the operating system and other factors.

This error is described in Microsoft's Knowledge base (ID: Q161255) and the problem is attributed to:

> "**problems with the Microsoft Jet Database Engine version 3.0 and 3.5. This problem no longer occurs in Microsoft Jet Database Engine version 3.51**"

Database operations have been moved to Jet v3.51 in order to correct this problem. The Test Executive v6.3 installation includes all of the necessary Jet Database Engine version 3.51 support files.

12. Appendix D - Glossary of Terms

Action Object	Executable application made available to the test operator
MSN	Manufacturer's Serial Number
PDEL	Parametric Data Exchange Language
Setup Object	Executable program designed for pre or post test operation
Test Object	An executable test designed to verify a UUT's ability to perform in an acceptable manner
Test Sequence	One or more test objects that are executed when the Run button is pressed.
TPC	Test Program Configuration
TPD	Test Program Definition file
TOD	Test Object Definition file
UUT	Unit Under Test

13. Appendix E - Formatting Mask

A format mask for test configuration data consists of placeholders and literal characters. The placeholders define the types of characters that can be entered (e.g. digits, letters or alphanumeric). The literal characters are typically separators (e.g. the hyphens in a phone number). The following table shows the characters available for defining a format mask.

Mask Character	Description
#	Digit placeholder
.	Decimal placeholder
,	Comma or thousands separator
:	Colon or time separator
/	Slash or date separator
\	Treats the next character in the mask as a literal. This allows the inclusion of the "#", "&", "?", ";" and "A" in the mask.
&	Character placeholder for ANSI characters in the following ranges: 32-126 and 128-255.
>	Convert all the characters that follow to uppercase.
<	Convert all the characters that follow to lowercase.
A	Alphanumeric character placeholder (entry required).
a	Alphanumeric character placeholder (entry optional).
9	Digit placeholder (entry optional).
C	Character or space placeholder (entry optional).
?	Letter placeholder
Literal	All other characters or symbols are displayed as themselves.

Note: When a format mask is defined for a data field, underscores are used as placeholders. Therefore, underscores are not allowed as part of the entered data. You must use "\;" to include a semicolon as a literal character.

When defining a TPD configuration entry mask that includes optional placeholders (e.g. 9 or a), it is necessary to also specify a minimum number of characters for the data (e.g. TestProcRevision = ; ?a; 1).

Here are some formats for commonly used data:

Phone Number (U.S.) = **(**###**)** ###**-**####
Social Security = ###**-**##**-**####
Date of Birth = ##**/**##**/**##

14. Appendix F - Version Changes

The following are lists of changes that have occurred from previous versions of the Test Executive.

Changes from Version 6.35.5 to Version 6.36

1. Added support to the Test Configuration dialog for drop-down list data entry. The contents of the drop-down lists are defined in the Configuration section of a TPD file. This includes the option to have an operator always **SELECT** a value after a TPD is loaded or to **RESTORE** the field to its most recent setting. The **SELECT** option has also been added to the custom Test Type list.

2. Added two optional Test Executive command line arguments for predefining an Operator ID ("***/op***") and UUT S/N ("***/sn***"). These are only intended for use during **development** as a way to speedup the TPD Load-Run cycle. Note that defining a UUT S/N in this way causes any **PreMSN** setup objects to be executed immediately following a TPD load.

3. Changed the verify display in the Test Results window to maintain better alignment of columns and headers when long verify descriptions are encountered. A similar change was also made to the Report Generator.

4. Made a slight change to how sequence data is retrieved from the database when printing a Last Run datasheet. Under certain circumstances the TPD's version check results would be printed rather than the results of the last test sequence.

5. Disabled entries in the Test Executive initialization file (**testexec.ini**) that allowed the toolbar and status bar to be hidden. Very occasionally system errors caused these options to be accidentally set and general confusion ensued.

Changes from Version 6.35.4 to Version 6.35.5

1. Corrected a problem with texAgent operation following a test object requested RELOAD_FINISH or RESET_FINISH.

Changes from Version 6.35.3 to Version 6.35.4

1. Added a new status value (**RESET_FINISH** = 0x5) for the **texFinish()** function. This causes the Test Executive to reload the TPD, as it does for **RELOAD_FINISH**, but the test object status is not restored.

2. The maximum abort delay that can be specified **via texSetAbortDelay()** has been extended from 2 minutes to 30 minutes (1800 seconds). If the abort delay is >15 seconds a message indicating the delay is displayed in the Test Results window. When aborting a test object DLL, the Test Executive GUI is disabled and a message box with the delay value is displayed in response to a mouse click. When aborting a test object executable, the Test Executive GUI displays a countdown value of the remaining time until the abort completes.

3. Corrected the **RELOAD_FINISH** behavior to restore the Recover section in the corresponding TPC file. This section is necessary when a test object retrieves a UUT's serial number via the **texGetTPC()** function.

4. Changed the way that Sequence and Test Object status is updated in the database. Previously, under certain rare conditions, the database would stop accepting the final Pass/Fail results for a test object or sequence.

Changes from Version 6.35.2 to Version 6.35.3

1. Corrected TexAgent message handling to work with Windows XP Service Pack 2 (SP2). Changes in SP2 caused problems with messages passed from the ETM to the Test Executive.

Changes from Version 6.35.1 to Version 6.35.2

1. Added support for printing multiple copies of datasheets on NT, Windows 2000 and XP systems (also supports page orientation).

Changes from Version 6.35 to Version 6.35.1

1. Added another formatting command to the custom datasheet header in the TPD file. This flag (**\l**) causes the custom header to be printed as the topmost item of a datasheet (i.e. above the line).

2. When a power monitor limit is reached, the ensuing abort setup objects are expected to turn UUT power off. If power was left on, the Test Executive control buttons were not properly disabled. This has been corrected.

3. Abort setup objects are now able to have full directory paths in the TPD file.

4. Embedded literal spaces are now supported for TPC data formatting.

5. An occasional "Change Directory" warning has been eliminated.

6. Corrected several problems associated with trailing spaces on TPD file entries.

7. Fixed Test Program List command line argument (**/tpl**) to display list before default TPD is loaded.

8. Corrected TPC window to only display Test Location field if it is specified in the TPD file.

Changes from Version 6.34.1 to Version 6.35

1. Time limits have been implemented on Test Executive menus and the Breakpoint Control dialog while testing is active. Previously, holding a menu open for over a minute would disrupt an active test. Now a menu is automatically closed after 10 seconds and the Breakpoint Control dialog is closed after 30 seconds (if a test is running).

2. The behavior of texGetTODVerify() has been modified (TEXDLL v3.30) for the **Get-Next** keyword. **Get-Next** now retrieves the next sequential verify based on the last texGetTODVerify() operation. Previously it ignored texGetTODVerify() calls that included a fixed verify id.

3. The order of events following a test object error has been modified. Previously the operator was notified of the error first, followed by the execution of any PostSequence setup objects. Waiting for an operator response to an error message needlessly delayed the completion of the PostSequence tasks. This order has been reversed.

4. Two test configuration fields, **Test Location** and **Custom1**, are now also supported in the Test Executive Configuration file. This provides additional opportunity to share common configuration settings with all TPD files.

5. The TPD revision value is now optionally displayed on the Test List window. This is controlled by a new keyword (**DisplayTpdRevision = Y**) in the Options section of the TPD file.

6. Action Objects (i.e. Tools) now directly support documents and some web references. These are viewed automatically when the Test Executive issues an "Open" command to the application associated with their file type (e.g. *.txt, *.pdf, *.htm). This avoids many of the cumbersome directory paths previously required when an application displayed a document or picture.

7. A new Test Executive command line argument (**/tpl**) has been added. This automatically presents the operator with the Test Program List when the Test Executive starts.

8. The Privilege Editor v1.05 has been modified to automatically encrypt all new privilege databases. An internal password is also assigned, to new databases, if the Test Executive Configuration file has an active **UsePassword** entry (UsePassword = Y).

9. Due to reoccurring problems with message boxes being hidden by other windows, several changes were made to message box behavior in the Test Executive and TEXDLL. All message boxes and dialogs generated by the TEXDLL are now set to be the topmost window. This prevents other windows from covering them. This includes all test object user interface functions (e.g. texMessageBox(), texInputString(), etc.) and verify breakpoints. Some common Test Executive error messages are also set to be topmost

10. To provide better visibility, a failing verify is issued if the maximum verify limit of 1000 is exceeded during a test object execution. The failure then appears in the Test List window and is tracked in the Report Generator. Warnings are issued for >1000 verifies, but they may be missed if warnings are disabled.

11. Similarly, an error finish is now issued by the TEXDLL, during texStart(), if greater than 1000 verify entries are found in a TOD file. This now matches the TEXDLL's response to other TOD file errors.

Changes from Version 6.34 to Version 6.34.1

1. A repeat count flag ("**/rpt 0:1:1**") is now passed to setup objects. This can be used by a PreSequence object to monitor the number of sequence loops that occur. Look in the *Test Object Command Line Arguments* section for more information on this flag.

Changes from Version 6.33 to Version 6.34

1. The Test Executive now supports a custom on-screen logo that is read from a file (**TexLogo.bmp**) in the TPD or application directory.

2. Custom headers, for printed datasheets, can now include an embedded graphic (\g) that is read from a file (**ReportHeader.bmp**) in the TPD or application directory.

3. Added new example projects to the SAMPLES subdirectory for National Instruments' LabVIEW and Geotest's ATEasy. These project directories also include application notes.

4. Made some changes to the reporting and display of sequence status when the sequence contains an aborted test object.

5. Corrected an error which caused a post-load failure if a new TPD was selected via the file dialog box.

6. A small correction was made to the presentation of binary comparison results in the verify breakpoint dialog box.

7. Two format mask characters ("**>**" and "**<**") are now interpreted correctly when validating the default minimum length for a TPC entry.

8. The Report Generator now supports **Ctrl-C** for copying selected text in the Test Results pane.

9. The automated Test Executive tutorial has been updated with overviews on test development and TOD files.

Changes from Version 6.32 to Version 6.33

1. A RELOAD_FINISH value is now supported for texFinish(). This causes the current TPD to be reloaded. This can be used by Abort Objects as a way to unload all resources being held by the TexDLLControl process. It can also be used by setup objects as a way to reload an altered TPD file (e.g. a TPD modified to match a UUT's configuration).

2. Test status returned by the Agent interface now accounts for sequences halted by SOF or test branching. Previously, a NONE status was returned if a sequence had halted early.

Changes from Version 6.3 to Version 6.32

1. Restarting the TexDLLController on TPD load (v6.3) broke interactive debugging of DLL test objects. This behavior has been modified to again allow interactive debugging of DLL test objects (for more information see *7.11.5 DLL Debugging*).

2. A Test Executive Quick Reference card, in Adobe Acrobat format, is now included with the installation.

Changes from Version 6.2 to Version 6.3

1. Visual Basic ActiveX DLLs are now supported as test objects. ActiveX DLLs are executed in the same process as C or C++ DLLs which allows them to share resources such as instrument handles. ActiveX functions are specified in a TPD file similar to DLL functions. The format is Project.Class::Function (e.g. ssTestObjX.Class1::MainTest). Source code for an example ActiveX test object is included in the installed \Samples\ActiveX directory. The source code comments include many details about the necessary behavior of an ActiveX test object. Note that an ActiveX DLL can be preloaded as a setup object, but it doesn't retain memory between executions.

2. Test results are now optionally logged when a test sequence is halted or aborted. This behavior is controlled by a keyword (LogOnAbort = Y) in the [Options] section of a TPD file. By default, PDL and database logging are suspended when an abort occurs. PDL logging also defaults to disabled when a test sequence is automatically or manually interrupted (e.g. by using the Quit button).

3. If a test object completes with an ERROR_FINISH, the resultant operator dialog box is now displayed as a topmost window. This prevents the error dialog from being "buried" by other application windows.

4. Some computers with more than 512MB of memory were unable to complete database operations. They typically reported the absurd message "Out of Memory". This was caused by errors in earlier versions of the Jet database engine. Database operations have been moved to Jet v3.51 in order to correct this problem.

5. New sections have been added to the documentation in order to better explain the interaction of the Test Executive and test objects. These new and updated sections include: *Test Executive Architecture*, *Test Object Design* and *Aborting a Test*.

6. Under certain circumstances, the execution thread of a test object DLL was not terminated when an abort occurred. This only happened if the DLL did not respond to an abort request. This has been corrected in TEXDLL v3.20.

7. Corrected a rare condition that crashed the Test Executive when a TPD file was loaded while the Debug Log window was full.

8. Changed Test Executive startup to restore to a maximized window if one was present during the previous shutdown.

9. During crash recovery, the Test Executive now performs a repair on the UUT database prior to restoring test status.

Changes from Version 6.01 to Version 6.2

1. The Test Executive now supports a data logging format that is compatible with IEEE-1545. The logging format is specified in the [Options] section of the Test Executive configuration file (TestExec32.cfg). IEEE-1545 is the default, but the original PDEL format is also available (DataLogType = PDEL). The logging format can also be overridden in the [Options] section of individual TPD files.

2. The Test Executive now supports conditional branching within the test list. This permits branching and looping based on the status of a test object or test group (e.g. Repeat(5) While Pass). An example TPD file (TxBranch.tpd) is included to demonstrate several aspects of conditional branching.

3. Three functions have been added to the TEXDLL for accessing information from TOD, TPD and TPC files. These functions are: texGetTODVerify, texGetTPC and texGetTPD. The example test objects have been modified to demonstrate these functions.

4. Due to the complexity introduced via conditional branching, a print option was added to print the results of the most recent test sequence (i.e. Last Run). Database logging must be active in order to support this print option.

5. Verify values larger than 10 characters are now automatically displayed, and stored in the database, in scientific notation (e.g. 9.83E+12).

6. Two optional fields have been added to the Test Configuration dialog. One is part of the IEEE-1545 specification (i.e. Test Location). The other is provided for general use (i.e. Custom1). Alternate labels and PDL headers for these fields are defined in the new [ConfigurationLabels] section of a TPD file (e.g. Custom1 = My Label; CUSTOM_HDR).

7. The management of PDL and database files is now controllable through the [Options] section of the Test Executive configuration file (TestExec32.cfg). Management options include number of files, age of files, operator prompting and type of removal (i.e. delete, recycle, never remove). PDL and database files are each managed separately. Databases can also be created with unique names (e.g. UniqueDatabaseName = Y).

8. An operator without "Flow" privileges is now restricted to running just the topmost item on the Test List (e.g. Full Run). This allows individual tests, such as diagnostics, to be isolated and reserved for higher-level operators.

9. If an operator starts the Test Executive without TPD loading privileges, the Test Executive no longer automatically loads the previous TPD. Instead, the Test Program List is automatically displayed for selecting a test program.

10. The command line arguments defined for a test object will now override the ones generated by the Test Executive. This gives a test developer control over breakpoint settings that a test object will receive. For example, a test object may have critical timing that must not be interrupted in a production environment. The test developer can disable verify breakpoints by including a 'disable verify breakpoint' flag ("/vn") in the test object's command line arguments.

11. User defined sections are now allowed in a TPD file. These can be used to hold additional control or test-related information. User defined section names must be prefixed with "usr" (e.g. [usrTestData]). User defined sections are permitted in TOD files if they are placed between the [Identification] and [Verifies] sections.

12. Unchangeable (i.e. static) configuration data is now being written to the TPC file. This change was made to improve access to all configuration data by the new TEXDLL function (texGetTPC).

13. Some additional information on aborting test objects has been added to the documentation. The updated sections are: Aborting a Test, Abort Delay and DLL Abort.

14. An additional document section has been added on DLL dependencies while preloading setup objects.

15. Additional format mask characters, for defining configuration data, have been documented.

16. LESS_THAN_EQ and GR_THAN_EQ data limits are now being correctly written to PDL files.

17. Certain conditions prevented test status from being recovered after a crash. This has been corrected.

Changes from Version 6.0 to Version 6.01

1. Some changes were made to DLL validation while loading, and default directories when executing.

2. A few changes were made to correct errors in operator ids and passwords.

Changes from Version 5.31 to Version 6.0

1. A Test Executive configuration file (TestExec32.cfg) is now supported. It optionally defines test station data (e.g. part number), a privilege database and a test program selection list.

2. If specified in the configuration file, a privileges database is used to control Test Executive behavior based on the operator's identity. Various capabilities of the Test Executive are individually assigned to each operator.

3. The optional test program selection list allows TPD files to be loaded from a descriptive list rather than by file name.

4. The Test Executive is now able to monitor power usage by test objects. A new function, texPowerMonitor(), allows test objects to report when they are applying power to a UUT. The Test Executive optionally displays this information and tracks power-on time against a set of limits specified in the TPD file. If the power-on time is exceeded, an abort is instigated and a recovery period is enforced.

5. Support has been added to the TEXDLL (texAbort) for an external application to initiate a Test Executive abort. This function should NOT be called by a test object because the resultant behavior is unpredictable.

6. The Test Executive now supports the printing of a detail datasheet. This detail datasheet contains the most recent results for each of the test objects listed in the Test Executive's test list. In addition, datasheets now have specific date and time information for each of the executed test objects. The "creation time" has been removed from datasheet headers.

7. Custom headers and footers, for printed reports, can now be defined in the Options section of a TPD file (i.e. ReportHeader and ReportFooter). Embedded formatting commands allow new lines (\n), tabs (\t), a time stamp (\@) and centered text (\c).

8. Data logging options in the TPD file now support a way to initialize user controlled logging to either off (USER;0) or on (USER;1).

9. An option for automatically resetting test status is now supported in the TPD Options section (e.g. AutoStatusReset = ALWAYS). When active, test status is reset whenever the Run button is pressed. The possible settings are the same as for data logging (NEVER, ALWAYS, USER, USER;0, USER;1).

10. The TEXDLL verify functions now support greater-than-or-equal (>=) and less-than-or-equal (<=) comparisons. TOD files similarly support LESS_THAN_EQ and GR_THAN_EQ limit types. The test result display and printed reports have an additional column to identify the comparison type.

11. The format of the Access database has been modified to include the custom report headers, report footers and verify limit types. A cascade delete has been added to each of the table relationships for easier maintenance.

12. The Test Executive now supports test program help files built with HTML (*.chm). This includes help files accessed from the Verify Breakpoint dialog.

13. Two new prerequisite types are now available. These are "Immediate Run" (**IR**) and "Immediate Pass" (**IP**). The purpose of these is to verify conditions within an execution sequence.

14. Several new command options have been added to the Report Generator. These include "Find Most Recent" and "Collapse/Expand All".

Changes from Version 5.3 to Version 5.31

1. The TEXDLL has been modified to cause test object prompts (e.g. texMessageBox) to always be the foreground window. This corrects a problem that test object DLLs had where the prompts were sometimes hidden under other windows.

2. Function verification, for test and setup object DLLs, has been disabled during TPD loading. Verification was occasionally failing because some DLLs would not load unless other DLLs were already present in memory (e.g. preloaded setup DLLs) or could be found on the system executable path. DLL functions are still verified when the test or setup object is executed.

Changes from Version 5.21 to Version 5.3

1. The 32-bit Test Executive now supports the use of DLLs as test objects. This allows the test object to execute in the same process as DLLs loaded during setup. Executing within the same process eliminates extraneous DLL loading and permits the sharing of memory and instrument handles. Test object DLLs are specified, in the TPD file, in the same field as a test object executable. The arguments field is used to define the test's function name and optional input arguments (e.g. usrExecuteTest(/f /ui)). For more information on test object DLLs, see *DLL Loading and Execution*.

2. Corrected several conditions that caused the Test Executive to fail. These include the Clear button on the Test Result window and systems set for 16M colors.

Changes from Version 5.2 to Version 5.21

1. Logging of user PDEL data (i.e. texLogPdel) has been enhanced for large sets of data. The data is now handled at file I/O speeds and no overhead is incurred if PDEL logging is disabled.

2. The ability to log measurement limits to a PDEL file has been added. This is enabled by adding an entry (e.g. PdelLimitType = SPECIFICATION) to the Options section of a TPD file. The specified string is used as the <limit type> for the DEFINE_LIMIT statement in the PDEL file.

3. PDEL file names have been made more unique for 32-bit systems. To allow extended tracking, PDEL file names are now composed of the UUT part number, UUT serial number, date and time of creation.

4. The printed and displayed reports are now more consistent in their test results notation.

5. A problem with the horizontal scroll bar on the Test Result and Debug Log windows has been corrected.

6. A problem with restoring and dispatching verify breakpoints has been corrected. Also, the Test Executive now inserts a space between the command flag ("/v") and the verify number or id (e.g. "/v 6"). This is more consistent with the other command flags and the documentation.

Changes from Version 5.0 to Version 5.2

1. Version 5.2 of the Test Executive is available for both 16-bit and 32-bit operation. There are separate installation programs for each type.

2. The Verify Breakpoint behavior has been enhanced to allow breakpoints on a specific verify id or verify number.

3. The requirement that Setup Object executables be located in the "Setup" subdirectory has been removed. They can now be specified with full or relative paths to other directories.

4. Initial breakpoint settings can now be controlled from within a TPD file. This aids creating test procedures because breakpoints are consistent when the TPD is loaded.

5. A new function, **texStartActive**, has been added to the TEXDLL. This function returns True if a test is active (i.e. **texStart** has been called) or False if it isn't. This is helpful when creating reusable software components that might be applied separately or with other modules.

6. Changes were made to the Test Executive to allow closer integration with the Environmental Test Manager (ETM). These changes include support for PDEL version 2.0, PDEL logging of custom test types, additional Agent interface commands (PDEL related) and a command line argument (/etm) for disabling the automatic TPD load.

7. The format of the Test Executive INI file has changed slightly. This includes file history delimiters and breakpoints. **Note, this file is not intended for external use and its contents may change without notice.** To start the Test Executive with a specific TPD file, pass it in on the command line.

8. The status of test groups, as reported through the Agent interface, was modified to prioritize for errors and incomplete testing. This better serves the needs of external control applications.

9. The Agent interface was extended to include Get:TestType and Get:Version operations. A timing problem for consecutive Reset and Run commands was corrected.

10. During PostLoad and PreLoad execution, the File Menu and associated controls are disabled. This prevents an operator from trying to load a different TPD during a lengthy PreLoad sequence. The Abort button is available during PreLoad to halt its execution.

11. Data format masks in a TPD file did not accept certain literal characters (e.g. \A, ; and \#). These are now handled properly (use '\;' for ';').

12. When executing in a continuous loop, occasionally the Test Executive reported that a test object completed without a **texFinish**. This has been corrected.

13. A Copy operation in the Test Results window did not always update the Windows Clipboard. This has been corrected. Also, an occasional problem with the Debug Log and Test Results horizontal scroll bar has been corrected.

14. A problem with data from the TPC file overwriting default values defined in the TPD has been corrected.

15. Several problems with the behavior of the Test List have been corrected.

Changes from Version 4.1 to Version 5.0

1. Version 5.0 of the Test Executive, and its related components, is written for 32-bit operation. This permits efficient operation in Windows 95 and Windows NT. Version 5.0 does not operate in Windows 3.11 or earlier.

2. The size available for revision fields in the Access database has been increased from 4 to 12 characters.

3. Help files can now be assigned to individual test objects in a TPD file.

4. The functions texInputString() and texInputNumber() now support word wrap and multiple lines for the prompt message.

5. A test program help file is now included with the example TPD files. The RTF source and project file are available in a subdirectory of the Samples directory.

6. The sample test object (TXDEMO32.EXE) now supports more operations through its command line arguments. For more information see *Appendix G - Sample Test Object*.

7. The Test Result window and Report Generator have been extended to support full display of 1000 verify results.

8. When the Quit button is pressed, a "Quitting" status is displayed until the current test completes. During that time, the Abort button remains enabled.

9. The technical sections of the user manual and help file have been extensively revised and enhanced. They now have more information on help file links, version checking and Windows operation.

10. A parsing error for static test configuration data in a TPD file has been corrected.

11. An occasional problem with the caption of the Test Result window has been corrected.

12. PDEL comments are now terminated by a semicolon.

13. The Test Executive can now print summary datasheets that extend beyond one page.

Changes from Version 4.0 to Version 4.1

1. The Run menu now has an entry that resets the displayed test status. This can also be activated through the agent interface (Cmd:Reset). This does not affect any test results that have already been logged.

2. The Test Configuration window can be optionally displayed every time a test is run. This capability is enabled by an option (TestConfigurationPrompt) in the TPD file.

3. DLLs loaded as part of the setup behavior can now be optionally unloaded in reverse order (i.e. opposite of their load order). This capability is enabled by an option (e.g. UnloadDLLs = Reverse) in the TPD file.

4. The test status is now reset when the UUT MSN is changed via the agent interface.

5. The Report Generator has been modified so that datasheets specifically label test results with their corresponding time and date.

6. The Test Executive has been modified to correct a general protection fault that occurred during remote polling by a client application.

7. A problem with the scrolling of the Test List Window has been corrected.

8. The TexAgent header file and example client application have been updated for LabWindows/CVI and VB 4.0 respectively.

9. A problem with starting a test by double-clicking while another test is running has been corrected.

10. Logging of invalid PDEL formats for 10, 100, 1000, etc. has been corrected.

11. Toggling between the Test Result window and the Test Executive is possible with the F8 key. This has been further documented and is visually indicated on the window.

12. All UNITS are now displayed and stored.

13. The TEXDLL has been corrected to handle values of 0xFFFFFFFF in a TOD file.

14. A problem while restoring the position of the Debug Log has been corrected.

15. The TEXDLL has been changed to accommodate negative values in a nominal percentage (NOM_PER) verify.

16. Corrected a problem in the Test Configuration window that caused fields with a data format, and minimum size, to duplicate characters.

Changes from Version 3.0 to Version 4.0

1. Multiple selections are now supported in the Test List. This allows several individual tests to be selected and executed. This capability is enabled by an option (i.e. MultiSelect = Y) in the TPD file.

2. The TPC data entry can be optionally restricted to specific formats. The data formats are set in the [ConfigurationInformation] section of the TPD file.

3. The Test Executive now supports a remote control (agent) interface. This allows other applications to retrieve information and execute tests.

4. Custom datasheet printing is supported by the Test Executive. TPD files now have a ReportPrinter option that indicates the program to execute when a datasheet is to be printed. This program can be created with Microsoft Access.

5. Test Objects specified in a TPD file now support an optional ID field. If present, this field is used to link to context sensitive test program help. This allows test references to be independent of test ordering and naming.

6. A new function, texSetAbortDelay, allows a test object to control the timing of the abort process. This function sets the time allowed between receiving a WM_CLOSE message and when the TEXDLL decides to forcibly terminate the test object.

7. The TEXDLL has been enhanced to allow access to breakpoint, warning and error functions by action objects (without requiring texStart or texFinish).

8. The Report Generator now saves and restores the font and point size for printing reports.

9. If a PreLoad or PostLoad setup object fails, the TPD load is canceled and the Test Executive is placed in an initialized state.

10. The expansion of the test list, when a TPD is loaded, can now be controlled from the TPD file. The option ExpandTestList can be set from 0 (no expansion) to 6 (expand all - default).

11. File verification and version checking is fully supported. Version checking results are logged to the database for each UUT. Version checking can be disabled in the [Options] section of a TPD file.

12. TEXDLL now supports texScale(), which scales measurements based on unit prefixes.

Changes from Versions 2.5 and 2.6 to Version 3.0

1. When starting, the Test Executive automatically reloads the latest TPD file from its file history. If the Test Executive previously crashed (e.g. due to a power failure), the user is prompted whether to restore test status from the database.

2. Test prerequisites are now supported. The two prerequisite options are: P = test object/group passed; R = test object/group run. Test objects also inherit prerequisites from parent test groups.

3. PDEL and database logging can now be controlled from an [Options] section of the TPD file (e.g. ALWAYS enabled, NEVER enabled, USER controlled). PDEL and database logging defaults to enabled.

4. Multiple setup objects of each type are now supported in TPD files. This includes the loading and unloading of DLLs for Pre- and Post- operations.

5. Components (i.e. VBX's, DLL's, etc.) required by the Test Executive are now checked and the operator is notified if they are not the correct version.

6. Holding the right mouse button down, in the Test List pane, selects the test object and pops-up information on its configuration (e.g. file name, part number, rev, prerequisites, etc.).

7. Double clicking on a test object now causes it to be run.

8. Trace breakpoints are now fully supported by the Test Executive.

9. The Debug Log and Test Result windows now support cursor operations and selective copying.

10. A full copy of the Debug Log is maintained in a file (DEBUG.LOG) until a new TPD or UUT MSN is entered.

11. Message box functions in TEXDLL now support an optional user-defined title.

12. The Verify Breakpoint dialog box has been enhanced to support a Help button and to remember its last position.

13. Updated TEXDLL.H with constants for significant digits, removed C++ comments and added #if check for _CVI_. Added constants for accessing TPC file contents.

14. The Report Generator now defaults to Arial 10 font.

15. The Report Generator now automatically selects all hidden subitems (i.e. children) when a parent is selected for printing.

16. The Report Generator and Test Executive can now access the database simultaneously.

17. Corrected TOD error that included "Verify=" in verify description.

18. Corrected error that occurred during PDEL logging of a zero value.

19. Corrected timing problem that occurred when a LabWindows/CVI test object was being logged to the database.

15. Appendix G - Sample Test Object

A sample test object executable (texdemo.exe or txdemo32.exe) is included with the Test Executive as part of the example TPD files. This test object is useful for demonstrating and testing various aspects of the TEXDLL interface and Test Executive operation. The behavior of this test object is controlled by a set of command line arguments. These commands allow the test object to emulate verify failures, optionally pass or fail, exercise user interface functions, finish with an error/abort status, operate with a normal window, execute a specified number of verifies or operate with a TOD file. Most of these options are exercised by an example TPD file (textest.tpd or txtest32.tpd).

The program's default behavior is as a test object with 10 verifies. Its behavior is modified based on the following command line arguments:

Flag	Behavior
Verify	
/f	Force a verify failure.
/ucf	Ask user if the test object should fail.
/vw	Force a verify warning.
/loop *Count*	Loop set of 10 verifies for *Count* times.
TOD	
/t *Number*	Execute the specified number of TOD verifies.
/tod *File*	Use contents of TOD *File* to create values for TOD verifies.
Interface	
/a	Pretend this is an abort program (no verifies).
/normal	Display test object window in normal mode (default is minimized).
/ui	Execute the user interface functions.
/pn *Name*	Print program *Name* and allow it to imitate different setup programs (no verifies).
texPowerMonitor	
/pon	Pretend that UUT power is on.
/poff	Pretend that UUT power is off (at end of test).
texFinish	
/ec	Finish program with an error status.
/en *Number*	Finish program with a negative error value/status.
/hc	Finish program with a halt status.
/ac	Finish program with an abort status.
/rc	Finish program with a reload status.
/sc	Finish program with a reset status.
/nf	Exit program without a texFinish().
Abort	
/adly *Num*	Set the abort delay to *Num* seconds.
/wm_close	Display a message box when WM_CLOSE is received (i.e. when Abort is pressed on the Test Executive).

This test object is a good starting place for exploring test object behavior and concepts. The complete source for this sample test object is included in a subdirectory of the Samples directory.

You should first compile and execute this sample project without changes. This verifies the operation of your development environment and the installation of the Test Executive. Then you can modify the sample project to meet your initial needs. Small incremental steps are highly recommended for this process.

Note: The key to modern development environments is the project, or "make", file. These define the several hundred options required to successfully compile, link and execute a program. While it is possible to correctly select each of these options yourself, it is much better to start with a working project file and modify it as necessary.

16. Serendipity Systems, Inc.

Serendipity Systems, Inc. was founded in 1984 to provide the ATE and electronics manufacturing community with test products and engineering services. For over 20 years Serendipity Systems has been involved in the development of custom software and hardware solutions for its clients. This has included ATE runtime software, test program translators, database systems, diagnostic program analysis software and handheld test equipment.

The company has produced custom GUI user interfaces, test programming post-processing tools, open architecture instrument integration strategies and fault isolation software in the form of fault dictionary and guided probe processors.

Serendipity Systems' staff is conversant in commonly used programming languages and systems including "C", "C++", Smalltalk, BASIC, Visual Basic, Pascal, DOS, Windows 95/98 and Windows NT/2K/XP.

Serendipity Systems has a variety of test products available. These include ATE runtime software, fault isolation tools, graphical display utilities and in-circuit test software and hardware.

Serendipity Systems, Inc.
PO Box 774507
Steamboat Springs, CO 80477

TEL: (720) 246-8925
FAX: (720) 246-8949

www.serendipsys.com

EMAIL: productinfo@serendipsys.com

Index

/

/rpt 82
/SeqPass 76
/ucf 215
/usr 82, 124

1

1545 117

A

Abort Delay 144
Abort Recovery 125
Aborting a Standalone Test Object 142
Aborting a Test 9, 141, 148, 158, 177
Aborting ActiveX DLLs 153
Access Functions 122
Action Objects 86
Action Objects Syntax 87
ActiveX DLL Test Objects 151, 152, 153, 154, 155
Agent Interface 53
Appendix A - Frequently Asked Questions 187
Appendix B - Units and Prefixes 193
Appendix C - Database Table Formats 113
Appendix D - Glossary of Terms 197
Appendix E - Formatting Mask 199
Appendix F - Version Changes 201
Appendix G - Sample Test Object 215
Appendix H - DLL Loading and Execution 145
ATEasy 119

B

Behavior Modes 123
branching 84, 85
Breakpoint Settings 75
Breakpoints 25, 51, 137, 162

C

C++ 119
client interface 53
Command Flags 123, 124, 178, 179
Command Line Arguments 54, 82
Complex TPD File 101
Component Validation 55
Conditional Branching 85
Configuration 93
Configuration Labels 97
Configuration Labels Syntax 97
Configuration Syntax 94
Control Buttons and Execution Status 9

Controlling Execution 8
crash recovery 106
Custom Datasheet Header/Footer 75
Custom Dialog Titles 135
Custom Logo Display 11
CVIDebug.dll 150

D

database 18, 29, 73, 113
Database "Out of Memory" Error 196
database repair 106
database size limit (2GB) 113
datasheet graphic 75
Debug Log 27
Debug Log File 62
Debugging An ActiveX DLL 154
Detail Datasheet Example 39, 40
Detailed Reload Behavior 127
Developers Application Notes 48
directory structure 56
DLL 76, 77, 79, 90, 145
DLL Abort 148
DLL Debugging 149
DLL Dependencies 146
DLL Error Recovery 150
DLL Example Test Object 145
DLL loading 145, 146
DLL Startup 147

E

Edit Menu 46
Embedded Breakpoints 137
Error Log File 62
example projects 119
exception handling for abort requests 148
Execution Status 10

F

FAQ 187
File Existence Checking 57
file management 56, 107, 110
File Menu 14, 34, 45
Files 39, 56, 57, 61, 62, 63, 65, 100, 103, 106, 107, 117, 138

G

Get-Current 168
Get-First 165
Get-Next 165
Getting Started 151
graphics 11, 75

H

Help Files 7, 48, 81
Help Menu 21

I

Identification 68
Identification Syntax 68
IEEE-1545 ..... 13, 18, 117
Information Functions 136
Introduction 1
Invalid Test Criteria 131

J

Jet Database 106, 113, 196

L

LabVIEW 119
LabWindows 119
Log File Management 56, 110
logo graphic 11
LogOnAbort 70

M

Microsoft Access 113
MSJT3032.DLL 195

N

National Instruments 119

O

Object Creation and Shared Memory 152
Object Naming 152
Options 69, 109
Options Menu 18, 36
Options Syntax 70
Other Files 91
Other Files Syntax 92
Out of Memory Error 196

P

Parametric Data Log File (IEEE-1545) 117
PDL/PDEL 117
Power Monitor 11, 88, 174
Power Monitor DLL 90
Power Monitor Syntax 88
preloading DLLs 146
Prerequisites 6, 48, 83
Printing a Datasheet 38
Printing to a File 39
Privilege Editor 43
Privilege Levels 44
Privileges 43, 44, 107, 111

R

Registering ActiveX DLLs 155
remote operation 53
repair database 106, 113
Report Generator 29
Report Printer 73

ReportHeader.bmp 75
RFD 76
Run Menu 16

S

sample projects 119
Sample TPD File 100
Security 111
Serendipity Systems Inc. 217
server interface 53
Setup Objects 76
Setup Objects Syntax 78
Significant Digits 132
software examples 119
Splash Screen 60
SSI 217
Status Bar 13, 33
Summary Datasheet Example 41

T

Test Branching 84
Test Configuration 23, 24
Test Control Options 12
Test Criteria 130
Test Description 81
Test Execution and Control 3
Test Executive Architecture 47
Test Executive Configuration File 107
Test Executive File Formats 61
Test List 30
Test List Status 5, 31
Test List Window 4
Test Methods 133
Test Object Command Flags 124
Test Object Command Line Arguments 82
Test Object Definition File 103
Test Object Definition Format 103
Test Object Design 49
Test Object DLL 145
Test Object Event Table 115
Test Object Example 145, 215
Test Object Help Files 138
Test Object Implementation 51
Test Object Operation 52
Test Object Programming Interface 119
Test Object Syntax 155
Test Objects 79
Test Objects and Microsoft Windows 139
Test Objects Syntax 80
Test Program Configuration File 106, 167
Test Program Configuration Table 114
Test Program Definition File 65, 168
Test Program Definition Format 67
Test Program Design 48
Test Program Help 7, 21, 48, 80
Test Program List 22, 112

Test Results 26, 32, 63
Test Sequence Table 115
Test Station 108
TestExec Initialization File 62
TestExec Log File 63
TestStation 108
texAgent 53
texCompare 158, 159
texCompareTOD 161
TEXDB30.MDB 113
TEXDLL Command Flags (switches) 124, 178
TEXDLL Function Definitions 157
TEXDLL Summary 120
texEmbedBreakpoint 162
texErrorBox 163
texFinish 164
texGetTODVerify 165
texGetTPC 167
texGetTPD 168
texInputNumber 170
texInputString 171
TexLogo.bmp 11
texLogPdel 172
texMessageBox 173
texPrint 174, 175
texScale 176
texSetAbortDelay 177
texStart 178
texStartActive 180
texVerify 181
texVerifyTOD 183
texWarningBox 184
texYesNoBox 185
TOD Files 81, 103, 138
TOD Identification 104
TOD Verifies 105
Toolbar 13, 33
Tools Menu 19
TPC 24, 93, 94, 106, 114, 167
TPD Directory Structure 56
TPD File 65, 66, 68, 69, 76, 79, 88, 91, 93, 97, 99, 168, 169
TPD Reload 125
try/catch for abort requests 148

U

User Control Options 72
User Controlled Failure 215
User Defined Section 99, 103
User Interface Functions 134
usr 99
usrAbortTest 141, 145, 148
UUT-Specific TPD 126

V

Verify Breakpoints 129
Verify Event Table 116

Verify/Compare Functions 128
Version Check 30, 55, 58
version check logging 55
Version Checking 55, 58
Viewing Test Information 6

W

Welcome 1
Window Menu 20
Windows Compliance for Test Objects 139
Windows GUI and Multitasking 140
WM_CLOSE 141, 148