
Customizing HP TestExec SL

Notice

The information contained in this document is subject to change without notice. Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP makes no warranties of any kind with regard to this document, whether express or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

Copyright © 1995 Hewlett-Packard Company. All Rights Reserved.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Microsoft® and MS-DOS® are U.S. registered trademarks of Microsoft Corporation.

Windows, Visual Basic, ActiveX, and Visual C++ are trademarks of Microsoft Corporation in the U.S. and other countries.

LabVIEW® is a registered trademark of National Instruments Corporation.

Q-STATS II is a trademark of Derby Associates, International.

Printing History

E2011-90016 — Software Rev. 2.10 — Rev. D (current with Rev. D of other HP TestExec SL manuals) - First Printing - May, 1997

E2011-90020 — Software Rev. 3.00 — Rev. E - January, 1998

E2011-90024 — Software Rev. 4.00 — Rev. F - August, 1999

About This Manual

This manual describes how to customize various aspects of HP TestExec SL, such as creating custom operator interfaces, writing switching handlers from scratch, customizing datalogging, and customizing online help.

Conventions Used in this Manual

Vertical bars denote a hierarchy of menus and commands, such as:

View | Listing | Actions

Here, you are being told to choose the Actions command that appears when you expand the Listing command in the View menu.

Items you must specify are italicized and enclosed by angle brackets, like this:

<filename.txt>

which you might replace by typing:

MyFile.txt

To make the names of functions stand out in text yet be concise, the names typically are followed by “empty” parentheses—i.e., `MyFunction()`—that do not show the function’s parameters.

Most programming examples use the C++ convention for comments, which is to begin commented lines with two slash characters, like this:

```
// This is a comment
```

C++ compilers also will accept the C convention of:

```
/* This is a comment */
```

The C++ convention is used here simply because it results in shorter line lengths, which make examples fit better on a printed page. If you are using a C-only compiler, be sure to follow the C convention.

Contents

1. Customizing the Operator Interface

About Operator Interfaces	2
What is an Operator Interface?	2
Which Operator Interfaces are Provided?	2
Why Customize an Operator Interface?	2
Which Programming Languages Can I Use?	3
What is the Best Way to Begin?	3
What Should an Operator Interface Look Like?	3
Overview	3
Know Your Audience	4
Keep the Appearance Simple	4
What Level of Access Should Operators Have?	5
Make the Layout Logical	5
Interacting With Operators	7
Overview	7
Providing Useful Prompts & Status Information	7
Minimizing Visual Clutter	8
Making Messages Clear	8
Preventing Common Errors Before They Occur	9
Using Shortcuts to Accommodate Different Styles	9
What About Multiple Languages?	10
What About Testing the Operator Interface?	11
About Automation Interfaces	12
What is an Automation Interface?	12
A Typical Scenario for an Automation Interface	12
What Tasks Does an Automation Interface Do?	13
Testing & Debugging an Operator Interface	14
How Should I Test and Debug an Operator Interface?	14
Using Sample Actions to Exercise an Operator Interface	14
Operator Interfaces Created in Visual Basic	15
What is the Standard Operator Interface in Visual Basic?	16
How Much Visual Basic Do I Need to Know?	17
How Does Visual Basic Interact with HP TestExec SL?	17
What is Inside the HP TestExec SL Control?	18

Adding the HP TestExec SL Control to a Project.....	19
Getting Online Help for the Control	21
Finding Items in Operator Interface Code.....	21
What is the Minimum Operator Interface to Run a Testplan?	21
Writing the Code for a Minimal Operator Interface	22
Why the Minimal Operator Interface is Not Enough.....	22
Understanding the HP TestExec SL Control's States & Methods....	23
Understanding the HP TestExec SL Control's Events.....	25
The Two Levels of Events	25
Events Associated with Testplans.....	25
Events Associated with Individual Tests	28
About Test-Level Events	28
Miscellaneous Events	30
Using the HP TestExec SL Control's Events.....	31
Understanding User-Defined Messages	31
Why Pass Information Between Processes?	31
Passing Information Between Processes.....	32
User-Defined Messages Reserved by Hewlett-Packard.....	39
Accessing Hardware Resources from an Operator Interface	39
When Do Operator Interfaces Access Hardware Resources?.....	39
Accessing the Hardware Resources	40
What About Concurrent Testing?	41
Miscellaneous Notes	43
Changing or Enhancing Existing Functionality	43
Changing the Configuration of an Operator Interface	43
A Quick Way to Hide Existing Functionality	43
Controlling the Information That Appears in Reports.....	44
Accessing the Default Information.....	44
What if Reports Need Additional Information?	45
What if Reports Need Different Information?.....	46
Changing the Language	47
Which Languages Can I Use?.....	47
Changing the Default Language	47
Switching Among the Built-In Languages	48
How Does Multi-Language Support Work?.....	49
What About Languages That Are Not Built In?	50
Adding Language Support for a New Control.....	53
Adding Language Support for a New Message.....	54

Contents

Prompting a System Operator from HP TestExec SL.....	55
Associating Testplans & UUTs with an Operator Interface	58
Using Peripherals with Operator Interfaces	61
Which Peripherals are Supported?	61
The “One Peripheral Per Form” Convention	61
Using Bar Code Readers	62
About Bar Code Readers.....	62
Changing the Processing of Bar Codes.....	63
Testing the Code for Bar Code Readers.....	64
Accessing Symbol Tables from an Operator Interface.....	65
Operator Interfaces Created in Visual C++	68
What is the Standard Operator Interface in Visual C++?.....	68
Inside an Operator Interface in Visual C++	69
Overview	69
How the Operator Interface Requests Service	70
Accessing Global Data from the Operator Interface.....	72
Interacting with the Test Sequencer	72
Creating an Operator Interface in Visual C++	73
Doing Specific Tasks with an Operator Interface in Visual C++.....	74
Responding to a “Run” Button.....	74
Beginning a Test Cycle	75
Displaying the Name of the Current Test.....	76
Displaying the Testplan and Test Timing	76
Displaying Messages.....	76
Beginning When the Testplan Name is Unknown	76
Creating an Automation Interface in Visual C++.....	77
Software Configuration for an Automation Interface	77
Choosing a Task Model in Windows	77
Using a Bar Code Reader	79
Monitoring Test Results	79
Displaying Messages to the User Interface	80
Responding to Keyboard and Mouse Commands	80
Generating Repair Information	81
Writing Repair Tickets	81

Signaling Downstream Devices	82
Datalogging	82
LAN Communications	82
Dealing with Problems	82

2. Creating a Hardware Handler

Writing a Hardware Handler	86
Modeling Your Hardware	86
Monitoring the Status of Hardware	87
Creating a Project for the Hardware Handler	89
Specifying the Path for Libraries	89
Specifying the Path for Include Files	90
Creating a New DLL Project	91
Specifying the Project Settings	92
Creating an Implementation File for the Hardware Handler	93
Writing the Routines for Functions in the Implementation File ..	94
Verifying the Project's Contents	99
Choosing Which Configuration to Build	99
Building the Project	100
Copying the DLL to Its Destination Directory	100

3. Customizing Datalogging

The Datalogging Configuration Editor	102
Modifying the Records & Fields in Datalogging Files	103
Enhancing Datalogging	104
Interacting with Symbol Tables	104
Knowing When a Datalogging File is Available	105
Using the Results of Datalogging in Custom Applications	106
Disabling the Writing of Datalogging Files	106
Custom Parsing the Results of Datalogging	107
Parsing the Results as a Testplan Runs	107
Parsing the Results in a Datalogging File	108
Changing the Name of the Datalogging File	109
Using the HP TestExec SL Datalogging Control	112
What is the HP TestExec SL Datalogging Control?	112
What's Inside the Datalogging Control?	112
Accessing Collections & Objects in the Datalogging Control	115

Contents

Adding the Datalogging Control to a Project	117
Getting Online Help for the Datalogging Control.....	118

Index

Customizing the Operator Interface

This chapter describes how to customize the user interface that operators of a test system use in a production environment. It also describes variations on operator interfaces for test systems that control automation equipment.

About Operator Interfaces

What is an Operator Interface?

Typically, you do not want operators of a test system in a production environment to access all the features that HP TestExec SL provides for developing testplans. For example, you probably do not want to let operators modify or delete the tests in a testplan. Or, your test system may need a simplified user interface so that non-technical operators can use it. Any of these variations on user interfaces intended to meet the needs of a specific set of system operators is an “operator interface.”

Which Operator Interfaces are Provided?

HP TestExec SL comes with two working user interfaces, one written in Visual Basic and one written in Visual C++, that are intended for use by production operators of a test system. These “ready to run” operator interfaces provide the basic control features and status information needed in a typical production environment. You can use one of these example operator interfaces as-is or customize it to meet your specific requirements.

Specific features of the example operator interfaces are described in greater detail later with related, language-specific topics.

Why Customize an Operator Interface?

Because its appearance and features define the tasks operators can do, to a large extent the operator interface dictates how operators interact with a test system. For example, a very simple operator interface might provide Start and Stop buttons as its only controls to deliberately limit operator interaction with a test system. It might include nothing more than a Pass/Fail indicator to provide status information if the operator’s task is simply to sort UUTs into groups of those that pass versus those that fail.

However, an interface intended for more sophisticated operators, such as those who do troubleshooting, probably needs more features. For example, a troubleshooter might need to know the name of the failing test and

information about how the test failed. Also, it might be useful to have an option that lets troubleshooters rerun the test or halt on the point at which it fails and manually take measurements with a DMM there.

Another good use for a custom operator interface is when you need to support multiple languages. For example, you could create separate interfaces for various languages or a single interface with an option that lets operators choose their preferred language.

Which Programming Languages Can I Use?

HP TestExec SL lets you create custom operator interfaces in either Microsoft Visual Basic or Microsoft Visual C++.¹ Generally speaking, using Visual Basic is the easier of the two. However, if you are familiar with Visual C++ and MFC (Microsoft Foundation Classes), you may prefer to develop operator interfaces in Visual C++.

What is the Best Way to Begin?

Regardless of which programming language you use, the best way to create a custom operator interface is to begin with one of the working examples provided with HP TestExec SL, and then customize it to meet your your specific needs.

What Should an Operator Interface Look Like?

Overview

Effective operator interfaces seldom just happen. Creating an operator interface can be analogous to writing a book insofar as designing an operator interface is like starting with a clean sheet of paper upon which you can write anything. In either case, you need to understand the needs of your audience (users) and create a clear, well organized end product that is appropriate for them. Time spent designing an operator interface that is easy

1. Besides Visual Basic, you can create an operator interface in any language capable of producing a Windows DLL. However, Visual C++ is the only language whose use we document.

Customizing the Operator Interface

About Operator Interfaces

to use will pay off in greater productivity and fewer errors by those who use the test system.

If you are not starting from scratch—e.g., you already have an operator interface that is used on other test systems—you have a couple of choices. If desired, you can implement that interface as HP TestExec SL's operator interface. Or, you can break with tradition and design a new—and perhaps easier to use—interface, especially if using HP TestExec SL causes you to move to the Windows platform and its conventions for user interfaces.

Something else you may want to consider is which other computer applications the operators of your test system use. If they already are familiar with another application, you can minimize operator training by designing a custom operator interface that works in a similar, familiar manner.

The next several topics describe design practices you should keep in mind when creating operator interfaces.

Know Your Audience

Be sure you know your intended audience. Are their skills nontechnical, semitechnical, or technical? Are they computer literate? Do they have enough familiarity with your product to understand product-specific terminology?

If your operators are nontechnical or not computer literate, you probably will need different wording than if they are. For example, it may not be safe to assume they are familiar with Windows terminology. Also, avoid jargon. Programmers might understand “Get the menu pick” but “Choose an item from the menu” would be more meaningful for most operators. When in doubt, simplify, because even operators with technical skills are unlikely to complain that your operator interface is too easy to use.

Keep the Appearance Simple

When planning the visual appearance of an operator interface, be conservative instead of making the interface fancy. Use large, legible fonts

and minimize the use of color and graphics unless they contribute *useful* information.

Do this...



Instead of this...



Making an operator interface too “busy” with numerous fonts, colors, and graphics can impair its usability because visual clutter makes it overly difficult for users to separate the significant from the insignificant.

Although it may seem that using all capital letters emphasizes text, studies have shown that overuse of capital letters makes text significantly more difficult to read. In fact, mixed-case characters increase reading speed and comprehension from 14-20% over all capitals. Thus, we recommend that you use mixed case in most text that appears in operator interfaces. Limit the use of all capitals to items that truly need emphasis.

What Level of Access Should Operators Have?

Given that an operator interface’s features set the boundaries of what operators can do with a test system, how much access to the test system do you want any given user to have? For example, an operator interface could let troubleshooters access a list of frequent failures and their causes. However, you may or may not want to let troubleshooters edit that database.

In general, you should allow as much access as operators need to do their jobs but probably no more than that. Instead of expecting operators to remember which features to *not* use, you probably should create a specific operator interface for each kind of user. Or, you could create a single interface with multiple personalities and display only one of those personalities at a time.

Make the Layout Logical

When laying out visual elements and controls, it helps to group related items and arrange them in a logical flow of tasks. Not only does this make it easier for operators to identify relationships, but it also reduces eye movement and hand movement when using a mouse. Also, try to group no more than five to

Customizing the Operator Interface

About Operator Interfaces

seven items at a time because that works best for retention from short-term memory. An example is shown below.

Do this...

Testplan Information	
Name	<input type="text"/>
Variant	<input type="text"/>
UUT Information	
Name	<input type="text"/>
Serial #	<input type="text"/>
Status Information	
System Status	<input type="text"/>
UUT Test Result	<input type="text"/>
Last Test Time	<input type="text"/>
Last Test Time	<input type="text"/>
Last Test Time	<input type="text"/>
# Passing Modules	<input type="text"/>
# Failing Modules	<input type="text"/>

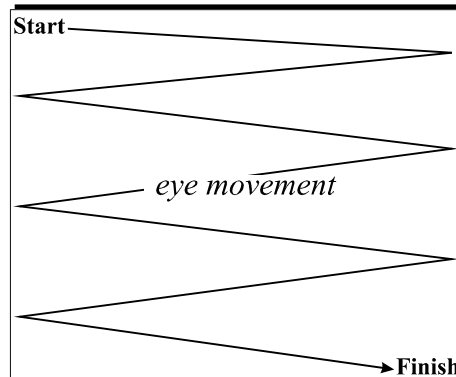
Instead of this...

Testplan Name	<input type="text"/>
Testplan Variant	<input type="text"/>
UUT Name	<input type="text"/>
UUT Serial #	<input type="text"/>
System Status	<input type="text"/>
UUT Test Result	<input type="text"/>
Last Test Time	<input type="text"/>
Average Test Time	<input type="text"/>
# Passing Modules	<input type="text"/>
# Failing Modules	<input type="text"/>

If your user interface must present a large number of features or a great deal of information, consider grouping the groups of items into additional group. Or, sidestep the problem with “information overload” by layering the information onto tabbed dialog boxes similar to the way in which the right pane of HP TestExec SL’s Testplan Editor window is organized.

If your operator interface contains a series of tasks for operators—i.e., specify a part number, specify a run number, specify which testplan to use, press the Start button, etc.—make those tasks flow according to the user’s expectations. Often, that expectation will be set by the user’s reading habits. For example, people who read in English read from left to right and from top to bottom. Thus, by default they tend to assume that a form “starts” at the upper-left corner and “ends” at the lower-right corner, as shown below. If

you follow this model when designing an interface for English-speaking operators, its users already will know something about using it.



If your intended audience is likely to assume a different model, then design the operator interface to match their assumptions.

Interacting With Operators

Overview

An effective operator interface provides its users with useful status information. For example, is the test system running or halted? What should the operator do next? How much progress has been made on the current task?

Providing Useful Prompts & Status Information

Ideally, your operator interface should prompt users when they need to do something, and provide status information at all times. For example, if testing is stopped the prompt might be “Press the Start button to begin testing.” If a test is running, you could display a message that says “Testing... Please wait” so operators will know what is happening and what is expected of them.

Another possibility is to combine status and prompt information directly on a button. For example, you could include the name of the UUT on the Start button and label the button “Start testing XYZ,” where XYZ is the type of UUT. Once testing begins, you could reprogram the button’s label to read “Testing XZY... Press to abort.”

About Operator Interfaces

Minimizing Visual Clutter

To reduce visual clutter—i.e., the presence of too many seemingly random elements on the screen at one time—you may want to reserve a region of the operator interface to display status messages, such as in a status bar at the bottom of the form. That way, operators always know where to look for status information.



Progress indicators, such as counters or graphical bars, are another useful way to keep users informed of what the test system is doing. This is especially true if there are other tasks that users could be doing while they wait for lengthy testplans to finish.

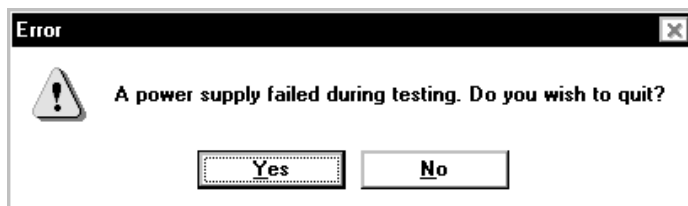


Adding a status bar or a progress bar can be as simple as dropping in a control provided with a programming environment, such as Visual Basic, and writing code that interacts with that control.

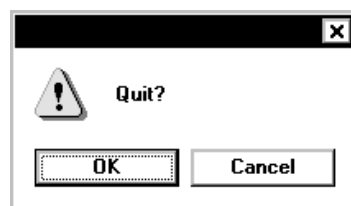
Making Messages Clear

When displaying messages for users, make them direct and precise. When offering choices, make it clear what the choices are and what they do.

Do this...



Instead of this...



Notice the differences between the two message boxes above. Without providing any information, the box on the right asks if the operator wants to quit. But which is the correct answer, OK or Cancel? Does OK mean it is okay to continue or okay to quit? Does the Cancel button cancel running or cancel quitting?

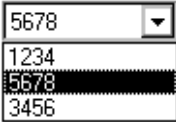
In contrast, the message box on the left provides information about why the box appeared and clear choices for what will be done if the operator presses a button.

Preventing Common Errors Before They Occur

Where possible, you should design the operator interface to prevent common errors as operators interact with it. For example, suppose operators need to specify the part numbers of UUTs prior to testing them. Instead of having operators enter the number via a keyboard, which is prone to error, you could provide a predefined, drop-down list on the operator interface or use a bar code reader to automate the process.


Do this...

Select the part number



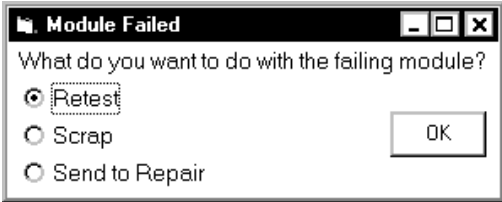
Instead of this...

Type the part number

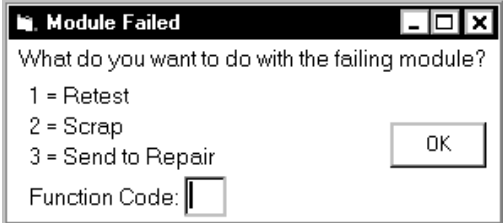


Another variation on the above is to let operators choose from predefined buttons instead of typing a response. Not only does this improve accuracy, but it usually is faster because it requires fewer actions. An example is shown below.

Do this...



Instead of this...



Using Shortcuts to Accommodate Different Styles

Some operators may find using a mouse convenient, while others—especially those with typing skills—may prefer using a keyboard. Because they provide keyboard equivalents for mouse commands, you may want to use shortcuts called “keyboard accelerators” to accommodate both preferences.

Customizing the Operator Interface

About Operator Interfaces

The Windows convention is that controls whose titles contain an underlined character can be operated by pressing the Alt key while typing the underlined character. The example below shows that the keyboard shortcut for the Start button is Alt-S.



Another useful shortcut for those who prefer using a keyboard is to take advantage of the “tab order”—i.e., the order in which the cursor advances from one control or field to the next when the Tab key is pressed—that most programming environments let you specify when designing forms. Specify a tab order that matches the logical progression of tasks done in the operator interface.

What About Multiple Languages?

It may be desirable for your operator interface to support multiple languages. If so, you must decide whether to have a separate version of the operator interface for each language or a single operator interface whose appearance varies according to whichever language option is chosen for it.

Be aware that most text grow longer when translated from English to other languages. This means that if you develop operator interfaces for the English language, you probably need to allow for the expansion or contraction of text in labels or messages if they will be used with other languages. The list below shows the approximate size of a typical, nontechnical paragraph of English text translated to various languages.¹

<u>Language</u>	<u>Size Comparison with English (100%)</u>
Arabic	88%
Chinese	61%
Czech	117%
Dutch	128%
Esperanto	93%

1. Source: George Sadek and Maxim Shukov, *Typography Polyglot*, New York: The Cooper Union, 1991

Farsi	100%
Finnish	104%
French	111%
German	109%
Greek	129%
Hebrew	83%
Hindi	91%
Hungarian	113%
Italian	110%
Japanese	115%
Korean	124%
Portuguese	110%
Russian	116%
Spanish	117%
Swahili	89%
Swedish	96%

What About Testing the Operator Interface?

Ultimately, the users of a test system will demonstrate how effectively your operator interface is designed. Often, you can reduce your overall effort by getting operators involved early. If possible, discuss their needs and expectations, and ask them to evaluate an informal prototype—such as a sketch on paper—of your proposed operator interface. Remember that time spent designing a useful operator interface may very well be time saved reworking an ineffective design later.

About Automation Interfaces

What is an Automation Interface?

An automation interface is a variation on an operator interface that has features to support a partially or fully automated production line. The general approaches to production automation supported by HP TestExec SL are:

- Centralized, in which a central computer controls the entire production line.
- Decentralized, or “peer-to-peer,” in which each machine in the production line is responsible for its own part of the process, performing a simple “hand-off” to the next machine when the current task finishes.

Depending on your specific hardware and production environment, you probably will use one of the following communication paths for the automation interface:

- RS-232 serial interface port used to communicate with a bar code reader, other automation devices, or a central computer for control or for gathering datalogging information.
- Digital I/O card for controlling relays and reporting the status of switches and sensors.
- LAN interface by some chosen communications protocol.

A Typical Scenario for an Automation Interface

One example of how HP TestExec SL could operate as part of an automation system is:

1. The HP TestExec SL system, under control of an automation interface, waits for notification that a UUT has been placed on the tester fixture and is ready to test.

2. The automation interface receives notification that a UUT is ready and receives a UUT identifier (usually a serial number scanned by a bar code reader).
3. The automation interface calculates from the identifier what type of UUT is present and which testplan the UUT requires.
4. If the testplan is not already loaded or if a different testplan is required, the automation interface loads the correct testplan.
5. The automation interface runs the testplan.
6. The automation interface determines the test results, including whether the UUT passed, failed, or caused a system exception.
7. The automation interface passes information to the automation system (whether that is a central computer or the next piece of automation equipment in the production line). This could include pass/fail information, repair ticket information, or datalogging information.
8. The interface waits for the next UUT to test or for a signal to shut down the test system.

What Tasks Does an Automation Interface Do?

Potential automation tasks you should keep in mind when designing an automation interface include:

- Displaying the system status to the user interface.
- Handling Windows-related events, such as any operator input by keyboard or mouse.
- Synchronizing testing activities with automation events, such as “UUT ready for test” or “UUT test complete.”
- Obtaining board identification and type and use this information to determine which testplan to run.
- Notifying automation equipment of the test outcome (pass, fail, or error).
- Recovering from errors that occur during automated testing.

Testing & Debugging an Operator Interface

How Should I Test and Debug an Operator Interface?

Thorough testing of an operator interface requires that you test every possible interaction between the operator interface and HP TestExec SL. This includes testing both HP TestExec SL's responses to actions initiated by the operator interface, and the operator interface's responses to events triggered by HP TestExec SL. For example, if you press the Run button on an operator interface you expect HP TestExec SL to begin running the currently loaded testplan. Also, if a testplan finishes running you expect to see a change in the status of the operator interface, such as a message to show whether the testplan passed or failed.

Testing how an operator interface handles errors is another aspect of testing and debugging. For example, an operator interface should not lock up if an instrument "times out" during testing. Also, an operator interface should handle typical errors such as a missing or incorrectly named testplan either through error checking or through careful design to prevent the errors from occurring.

Using Sample Actions to Exercise an Operator Interface

HP TestExec SL provides an easy way to rapidly construct simple testplans for exercising the functionality of an operator interface. Directory "*<HP TestExec SL home>\samples\uidebug*" contains a basic set of action definitions and their matching DLL that you can use to create passing and failing tests, raise exceptions, and more. For more information about the actions, use the Action Definition Editor or the Testplan Editor window to browse their descriptions.

Operator Interfaces Created in Visual Basic

Note

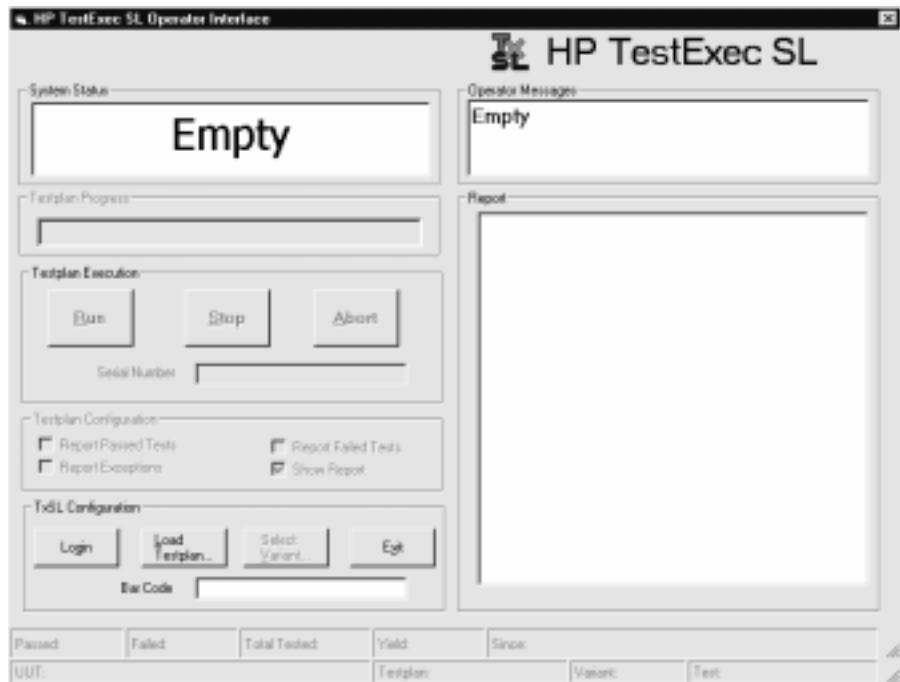
The sample operator interface provided with HP TestExec SL that is created in Visual Basic has some automation features built into it, while the sample operator interface created in Visual C++ does not.

Note

We recommend that you do not simultaneously run an operator interface created in Visual Basic and the Test Executive environment used to develop testplans. Running both at once can cause unpredictable behavior, conflicts, and loss of data.

What is the Standard Operator Interface in Visual Basic?

The sample operator interface created in Visual Basic 6.x, which is shown below, provides a useful example for use as-is or with minimal customization.



By default, it supports a bar code reader inserted inline with the keyboard cable. It also supports other peripherals and options via configuration settings described later under “Changing the Configuration of an Operator Interface.” You can find the code for the sample operator interface’s project in directory “<HP TestExec SL home>\samples\visualbasic\operatorinterfaces\typical”. A compiled, executable form of the operator interface resides at “<HP TestExec SL home>\bin\typicalopui.exe”.

Note

You can find the code for an even simpler operator interface in directory “<HP TestExec SL home>\samples\visualbasic\operatorinterfaces\simple”. The simple example is a good tool for learning about operator interfaces but lacks the range of features and error handling needed in a real application.

How Much Visual Basic Do I Need to Know?

The topics in this section assume you are familiar with:

- Visual Basic 6.x terminology and concepts, such as events, methods, properties, and event-driven programming
- Visual Basic 6.x’s integrated development environment, or IDE, which provides the tools used to create, edit, and debug programming projects
- The use of ActiveX™ controls with Visual Basic 6.x

If your experience with Visual Basic is limited, you probably will want to begin with simple customization tasks, such as changing the appearance of an operator interface but not changing its underlying functionality. This will help you become familiar with Visual Basic and with the sample code provided with HP TestExec SL. As your proficiency with both grows, you can begin doing more extensive customization tasks.

If you are new to Visual Basic programming, we recommend that you read the *Programmer’s Guide* available in Visual Basic’s online help.

How Does Visual Basic Interact with HP TestExec SL?

Operator interfaces created in Visual Basic use a special HP TestExec SL ActiveX control to interact with HP TestExec SL. As shown in the diagram below, code written in Visual Basic does the following:

- It calls the control’s methods to cause actions to occur, such as using the Run method to begin running a testplan.
- It uses the control’s properties to set or return attributes associated with HP TestExec SL, such as using the DataLogDirectory property to set or return the path used when datalogging during testing.

Customizing the Operator Interface

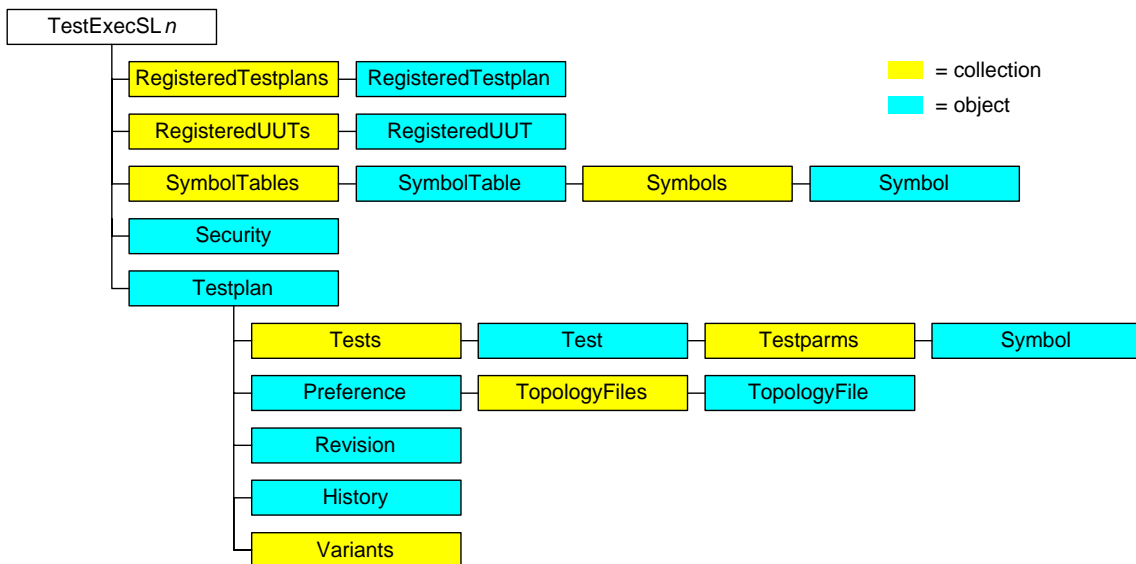
Operator Interfaces Created in Visual Basic

- It responds to events triggered by HP TestExec SL, such as the AfterTestplanStop event that indicates a testplan has finished running.



What is Inside the HP TestExec SL Control?

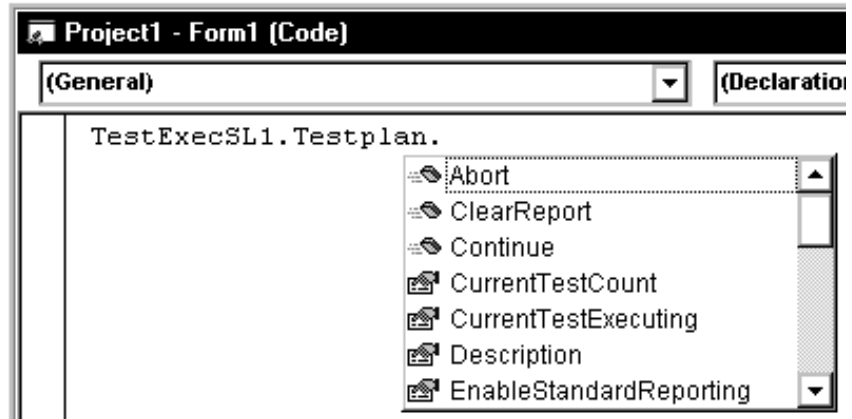
The HP TestExec SL Control is an automation object that contains additional automation objects and collections of automation objects. The hierarchy of internal objects and collections of objects is shown below.



Note

The properties of the HP TestExec SL control's internal objects do not appear in Visual Basic's Properties window. You must browse the control's online help or use Visual Basic's Object Browser to find descriptions of its internal objects and their properties and methods. Also, you can see them used in the sample operator interface provided with HP TestExec SL.

Visual Basic's Auto List Members feature provides an easy way to use the internal objects without having to remember their properties and methods. As shown below, when you type a period after the name of an object, a list of the object's properties and methods appears in Visual Basic.



Here, the list for the Testplan object is shown. Among its other attributes are an Abort method and a CurrentTestCount property.

Adding the HP TestExec SL Control to a Project

Assuming that HP TestExec SL is installed on your system, you can do the following to add the HP TestExec SL control to your project:

1. Open an existing or a new project in Visual Basic
2. Choose Project | Components in Visual Basic's menu bar.
3. When the Components box appears, be sure its Controls tab is selected.
4. Choose the Browse button and locate the HP TestExec SL control, which is in file "txslctl.ocx" in directory "<HP TestExec SL home>\bin". When

Customizing the Operator Interface

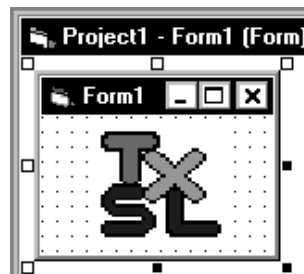
Operator Interfaces Created in Visual Basic

this file is selected, the HP TestExec SL control will appear in the list of controls, as shown below.



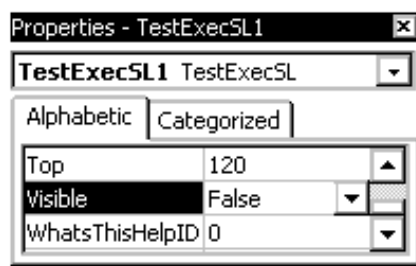
5. Choose the OK button.

Once the HP TestExec SL control appears in Visual Basic's Toolbox, you can use the mouse to place it on a form as you would any other control. When copied onto a form, the control looks like this:



Note

As shown below, you probably will want to set the HP TestExec SL control's *Visible* property to *False* to keep the control from appearing at runtime.

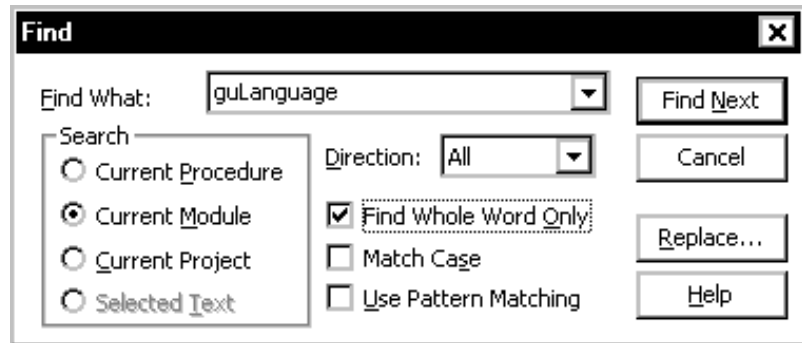


Getting Online Help for the Control

You can invoke online help for the HP TestExec SL control by selecting the control when it appears on a form and then pressing softkey F1.

Finding Items in Operator Interface Code

Given that the predefined operator interface provided with HP TestExec SL will contain a great deal of unfamiliar code at first, how can you find items of interest? Visual Basic's Find feature (Edit | Find) makes it easy to locate specified text in Visual Basic projects. The example below shows the Find feature being used to find occurrences of variable `guLanguage` in the current code module.



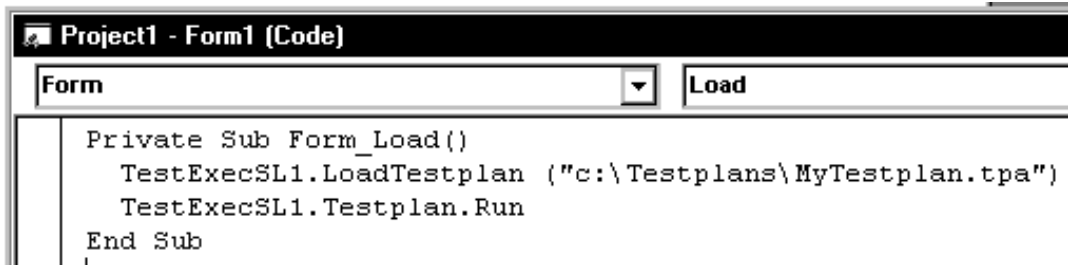
Similarly, you can search (and replace, if desired) within the scope of a procedure, a module, a project, or a selected region of code.

What is the Minimum Operator Interface to Run a Testplan?

Running a testplan from an operator interface created in Visual Basic requires nothing more than adding the HP TestExec SL control to Visual Basic's toolbox (see "Adding the HP TestExec SL Control to a Project"), copying an instance of the HP TestExec SL control onto a form, and writing two lines of code.

Writing the Code for a Minimal Operator Interface

The default name of the first instance of the HP TestExec SL control is “TestExecSL1”. Given that, the minimum code needed to load and run a testplan named “MyTestplan.tpa” looks like this:



```
Private Sub Form_Load()  
    TestExecSL1.LoadTestplan ("c:\Testplans\MyTestplan.tpa")  
    TestExecSL1.Testplan.Run  
End Sub
```

When the form that contains the control is loaded, the control’s LoadTestplan method is called and passed the pathname of an existing testplan to load. After the testplan has been loaded, a call to the Run method associated with the control’s Testplan object runs the testplan.

Why the Minimal Operator Interface is Not Enough

Although the example above works, it provides very limited functionality. For example, there is no error trapping routine to handle a simple problem like not finding the testplan at its specified location. Also, this operator interface provides no pass/fail status information, nor does it let the operator load and run a different testplan, or even the original testplan a second time.

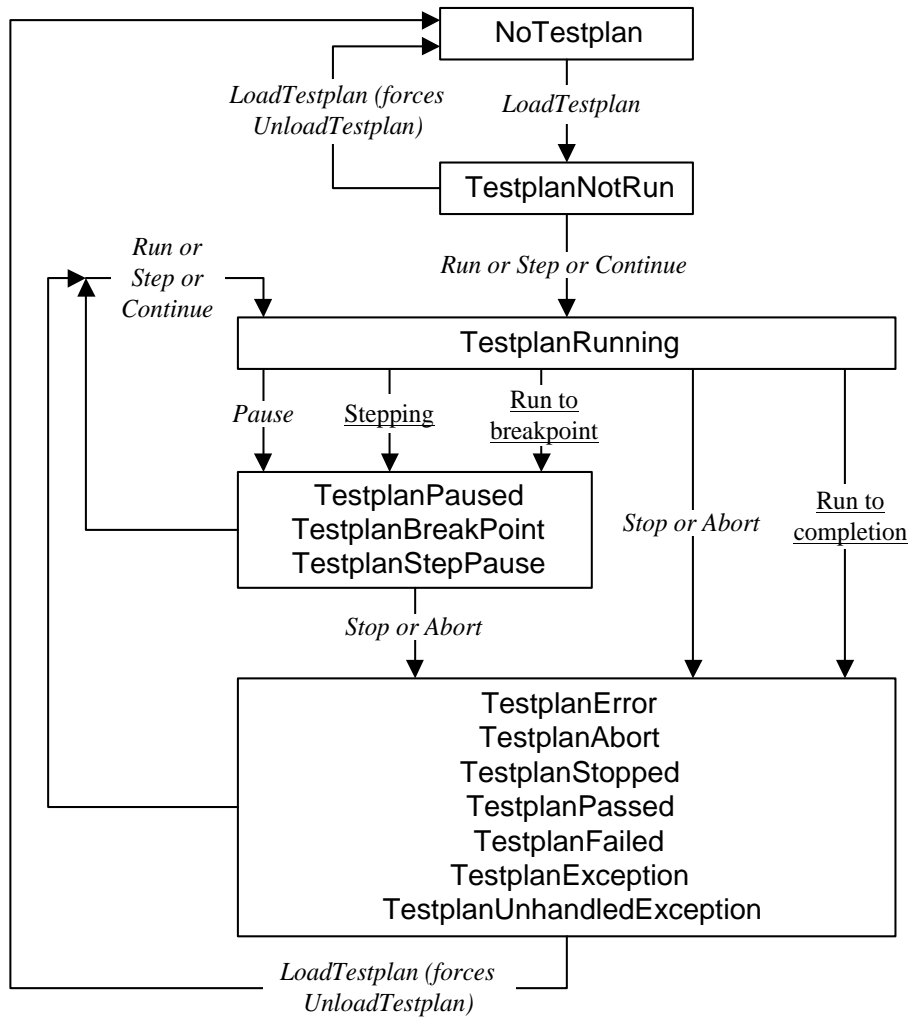
Note

Despite the minimal operator interface’s lack of features, it can be a useful tool for learning about the HP TestExec SL control. For example, you could have the minimal operator interface run a testplan containing a single action that displays a message box to indicate that the testplan is running. Then you could add Run and Stop command buttons to the operator interface, add a text box to display the testplan’s status, etc.

The more extensively you customize operator interfaces, the better you need to understand the HP TestExec SL control’s methods, properties, and events so you can use them programmatically. Also, you need to understand the various “states” of execution through which HP TestExec SL moves. States, methods, and events are described in more detail below.

Understanding the HP TestExec SL Control's States & Methods

As it loads, runs, and unloads testplans, HP TestExec SL moves through various “states” of execution identified by boxes in the diagram below. Each box contains one or more states; e.g., the first state is NoTestplan, the second is TestplanNotRun, and the third is TestplanRunning.



Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

HP TestExec SL moves from one state to another by either the normal sequence of execution or by calling methods in the HP TestExec SL control. For example, calling the `LoadTestplan` method causes HP TestExec SL to load a testplan and move from the `NoTestplan` state to the `TestplanNotRun` state. The names of methods are italicized in the diagram.

Some boxes contain more than one state. For example, the fourth box from the top contains `TestplanPaused`, `TestplanBreakPoint`, and `TestplanStepPause`. This means that any of these states is possible at this point in the flow of testing.

Which state actually occurs depends upon how this point is reached. For example, calling the `Pause` method while in the `TestplanRunning` state moves to the `TestplanPaused` state. Similarly, single-stepping while in the `TestplanRunning` state moves to the `TestplanStepPause` state because HP TestExec SL pauses after each step.

The associations between related states and methods are easy to identify because of the similarities in their names. For example, if HP TestExec SL is in the `TestplanRunning` state, then:

- Calling the `Stop` method moves to the `TestplanStopped` state
- Calling the `Abort` method moves to the `TestplanAborted` state

Notice the three labels that are underlined in the diagram: Stepping, Run to breakpoint, and Run to completion. Instead of being methods, these are normal paths of execution that result in movement from one state to another. For example, if a testplan executes without the number of failing tests exceeding the specified limit¹, it moves from the `TestplanRunning` state to the `TestplanPassed` state.

Also notice the two cases in which calling the `LoadTestplan` method causes an automatic call to the `UnloadTestplan` method because the current testplan must be unloaded before loading a different one. These

1. This limit is set by the “Halt on failure count” feature on the Execution tab in the right pane of HP TestExec SL’s Testplan Editor window.

occur when moving from `TestplanNotRun` to `NoTestplan` and from any of the states in the lowermost box to the `NoTestplan` state.

In some cases, any of several methods can cause movement from one state to another. For example, calling either the `Run`, `Step`, or `Continue` method causes movement from the `TestplanNotRun` state to the `TestplanRunning` state.

You can learn the current state of testing by examining the value of the HP TestExec SL Control's `State` property. For more information, see the description of the `State` property in the online help for the control.

Understanding the HP TestExec SL Control's Events

The Two Levels of Events

The HP TestExec SL control can trigger two levels of events in response to changes of state in HP TestExec SL. The first type is testplan-level events that are global to a testplan and always enabled, which means they potentially trigger each time a testplan is run.¹ The second level is test-level events, whose scope is individual tests and whose triggering you can enable or disable programmatically.

Why have two levels of events? Testplan-level events let an operator interface respond to major changes in a testplan's status. They provide useful status information without significantly slowing testing. On the other hand, test-level events provide a greater level of resolution but at the expense of increased testing time.

Events Associated with Testplans

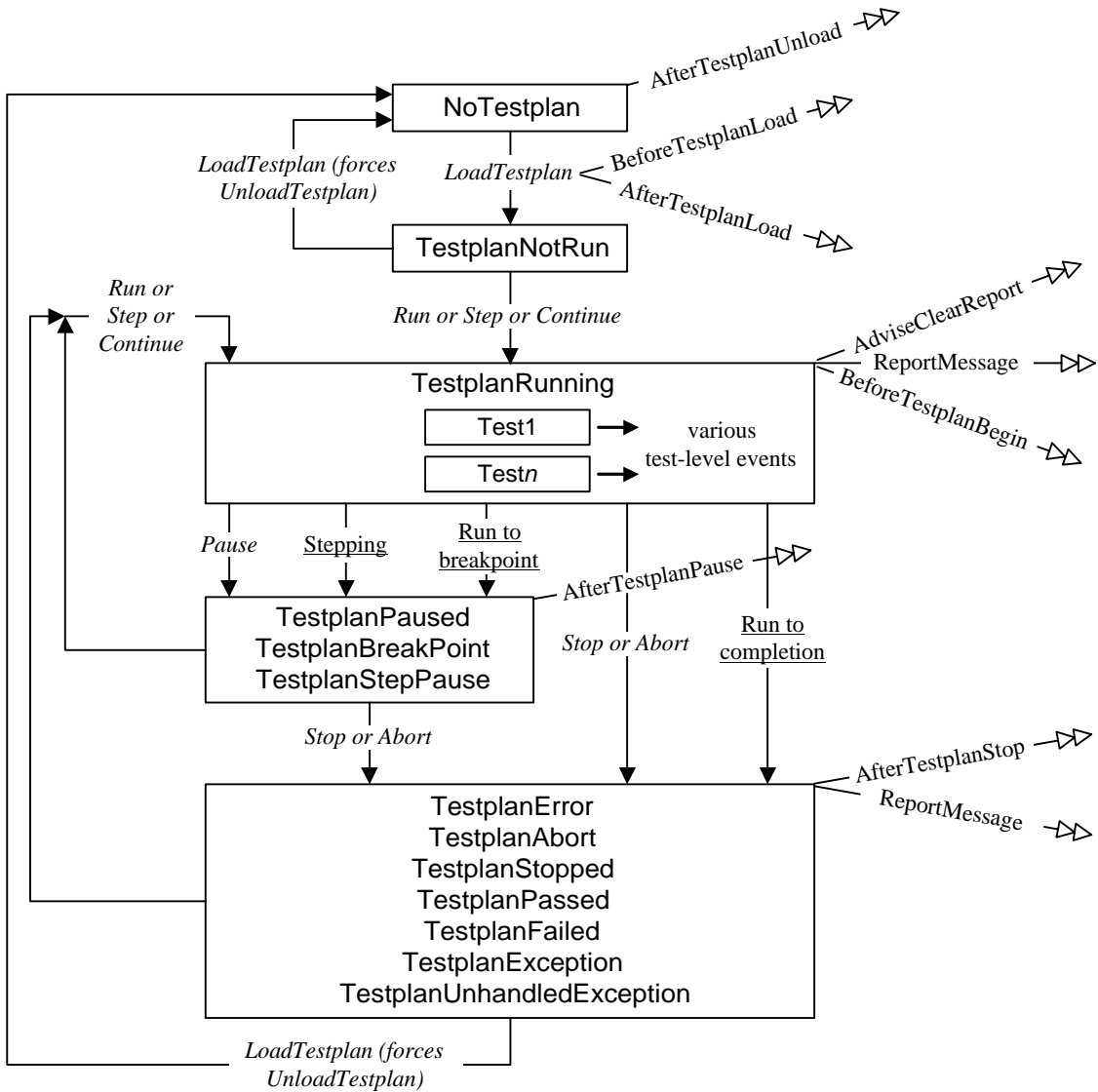
At various points when HP TestExec SL moves from one state of execution to another, the HP TestExec SL control triggers events you can use to execute routines written in Visual Basic. For example, there is an `AfterTestplanPause` event for which you could write code to change a status message in an operator interface from "Running" to "Paused" in

1. We say "potentially" because not every event will necessarily trigger each time a testplan is run. For example, an event that triggers when the testplan pauses will not trigger unless the testplan is deliberately paused. However, all of these events are enabled to trigger at the appropriate time.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

response to a call to the `Pause` method, upon reaching a breakpoint set in the testplan, or while single-stepping. The diagram below shows testplan-level events as arrows with double heads that are hollow.



Listed in alphabetical order, the testplan-level events are:

AdviseClearReport	Indicates that report output is being cleared at the beginning of a run of the testplan. Typically triggers once at the beginning of a run of the testplan, even if the testplan will loop.
AfterTestplanLoad	Indicates that a new testplan has successfully been loaded.
AfterTestplanPause	Indicates that HP TestExec SL has entered a paused state, and returns the reason why the testplan paused.
AfterTestplanStop	Indicates that HP TestExec SL has halted, and returns the reason why the testplan stopped.
AfterTestplanUnload	Indicates the testplan has been unloaded.
BeforeTestplanBegin	Indicates that a pass through the testplan sequence is about to begin. Also occurs when testplan execution resumes via calling the Continue method.
BeforeTestplanLoad	Triggered in response to a LoadTestplan method.
ReportMessage	Indicates a new "block" of report output has arrived.

Note

You can find detailed descriptions of these events in the online help for the HP TestExec SL control.

Notice that some states have more than one event associated with them. For example, the `TestplanRunning` state has the `AdviseClearReport`, `ReportMessage`, and `BeforeTestplanBegin` events associated with it. When there are multiple events, the events trigger in the order shown from top to bottom.

Most calls that cause a transition from one state to another return immediately; i.e., they are non-blocking. This means that in most cases you can think of testplan-level events as triggering immediately after the call.

Operator Interfaces Created in Visual Basic

For example, calling the `Run`, `Step` or `Continue` methods immediately triggers an `AdviseClearReport` method just prior to moving HP TestExec SL from the `TestplanNotRun` state to the `TestplanRunning` state.

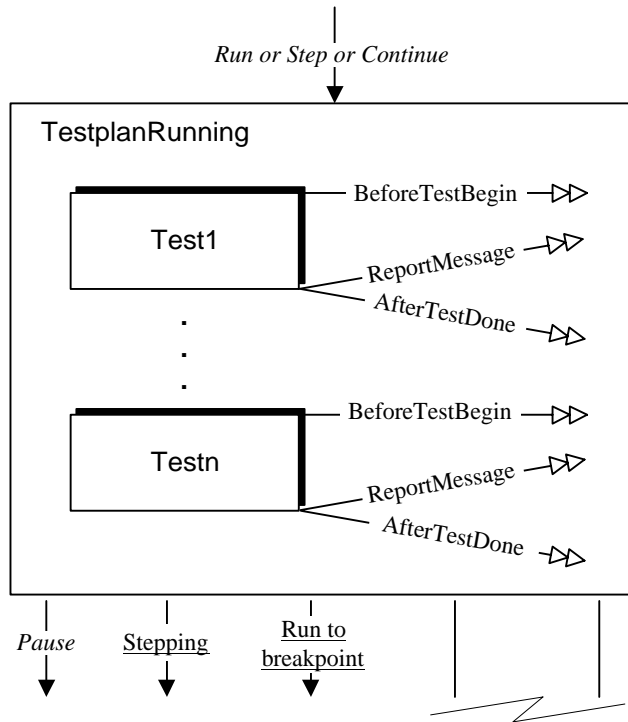
The `LoadTestplan` method is an exception to the above. When called, it blocks until its action is complete. This means that the `BeforeTestplanLoad` and `AfterTestplanLoad` events trigger in the order shown sometime during the call to `LoadTestplan`, but before HP TestExec SL enters the `TestplanNotRun` state.

Events Associated with Individual Tests

About Test-Level Events

Test-level events are triggered as the status of testing changes from test to test within a testplan. As shown in the expanded view of the `TestplanRunning` state below, test-level events are associated with the

beginning and end of individual tests during the `TestplanRunning` state of execution.



Listed in alphabetical order, the test-level events are:

- AfterTestDone** Triggered after the current test finishes executing.
- BeforeTestBegin** Triggered before the next test in the testplan begins executing.
- ReportMessage** Indicates a new "block" of report output has arrived.

By default, test-level events are disabled. You enable the `AfterTestDone` and `BeforeTestBegin` events by setting the value of the `TestEventsEnabled` property in the HP TestExec SL control to `True`. This lets the code in an operator interface or automation interface take some kind of action before and after the running of each test.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

The test-level `ReportMessage` events trigger under either of the following conditions:

- Passing or failing tests occur and the value of the `ReportPass` and/or the `ReportFail` property in the HP TestExec SL control is set to `True`
- An exception occurs and the value of the `ReportExceptions` property in the HP TestExec SL control is set to `True`

Caution

Enabling test-level events slows testing because it takes time to process events and broadcast them to their recipients. Thus, you probably will not wish to use test-level events when testing times are short or when timing is critical.

Note

You can find detailed descriptions of these events in the online help for the HP TestExec SL control.

Miscellaneous Events

The HP TestExec SL control provides a couple of additional, asynchronous events that do not fall into the category of testplan-level or test-level events. They are:

- | | |
|---------------------------|---|
| AdviseUpdate | Triggers to let the operator interface update its display in a way that does not interrupt the critical timing of a testplan. |
| UserDefinedMessage | Triggers to notify the operator interface that a user-defined message has arrived. |

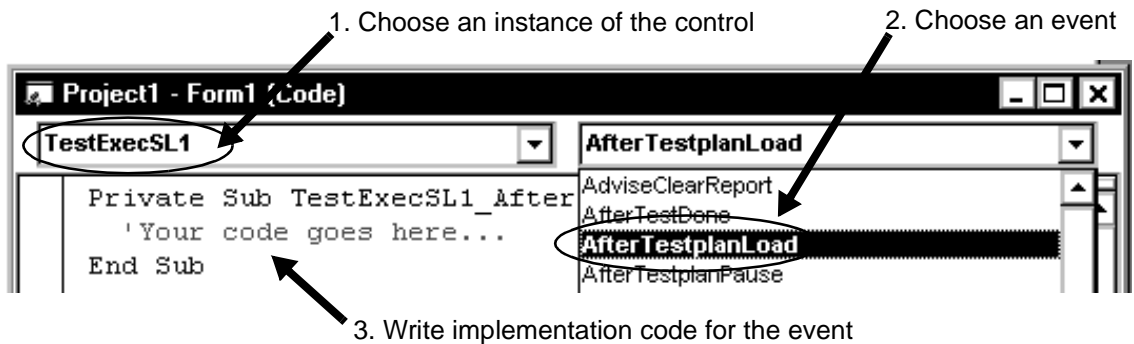
You can find more information about these events in the online help for the HP TestExec SL control. Also, see “Understanding User-Defined Messages” for more information about user-defined messages and “Miscellaneous Notes” for more information about using the `AdviseUpdate` event.

Using the HP TestExec SL Control's Events

As shown below, once you have created an instance of the HP TestExec SL control on a form,¹ you can:

1. Use the Object box in Visual Basic's Code window to choose the name of an instance of the HP TestExec SL control.
2. Use the Procedure/Events box to choose a specific event and add its declaration to the form.
3. Write code to implement what happens when the event is triggered.

An example of doing this is shown below..



Understanding User-Defined Messages

Why Pass Information Between Processes?

An operator interface written in Visual Basic executes apart from HP TestExec SL, which means that it and HP TestExec SL reside in separate processes. Interaction between these processes is handled by the

1. The sample operator interface provided with HP TestExec SL has the control on "frmMain".

Operator Interfaces Created in Visual Basic

HP TestExec SL control, which passes messages back and forth. Examples of the kinds of message passed include:

- Messages that originate in HP TestExec SL and cause the triggering of events in the HP TestExec SL control in Visual Basic
- Messages that originate in Visual Basic and are sent to HP TestExec SL when you call methods in the HP TestExec SL control

Because the HP TestExec SL control passes these messages in a predefined manner, you are not necessarily aware of them. Nor do you need to know anything about their contents. All you see is the results.

But what happens if you need to pass information between processes and none of the predefined messages—i.e., the HP TestExec SL control’s events and methods—is appropriate? For example, suppose you need to:

- Have an action written in C that is executing in a testplan in HP TestExec SL invoke a dialog box in an operator interface written in Visual Basic, and
- Wait for a "Yes/No" response from the operator of the test system, and
- Return the operator’s reply to the action.

However, the HP TestExec SL control has no “Display a dialog box and wait for a reply” event, nor does it have a “Return a reply from the dialog box” method.

Passing Information Between Processes

Instead of providing a large number of very specific predefined functions, events, and methods to pass information between processes, HP TestExec SL supports a more flexible approach called “user-defined messages.” At a conceptual level, user-defined messages work like this:

- A call to the appropriate API function or method broadcasts a user-defined message to potential listeners. Besides containing data, the message contains an identifier that identifies what kind of message it is. For example, you might choose a convention that defines an identifier of

5 for messages sent from an action to an operator interface that mean “display a dialog box.”

- Potential listeners receive various user-defined messages. In each case, they evaluate the message’s identifier to decide if the message is of interest. Continuing with the example of a message whose identifier is 5, when the operator interface receives a message whose identifier is 5, it displays a dialog box in response. When the operator interface receives other messages whose identifiers are not 5, it either ignores them or takes some other action appropriate for their identifiers.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

You use the following functions, events, and methods to send, receive, and respond to user-defined messages.¹

Functions for use in code written in C^a

UtaSendUserDefinedMessage()	An API function used in actions to broadcast a message to all potential listeners and does not wait for a response
UtaSendUserDefinedQuery()	An API function used in actions to broadcast a message to all potential listeners and wait for a response
UtaSendUserDefinedResponse()	An API function used in actions to respond to user-defined messages
AdviseUserDefinedMessage()	A function used in hardware handlers to respond to user-defined messages

- a. With the exception of AdviseUserDefinedMessage(), these functions are part of the C Action Development API.

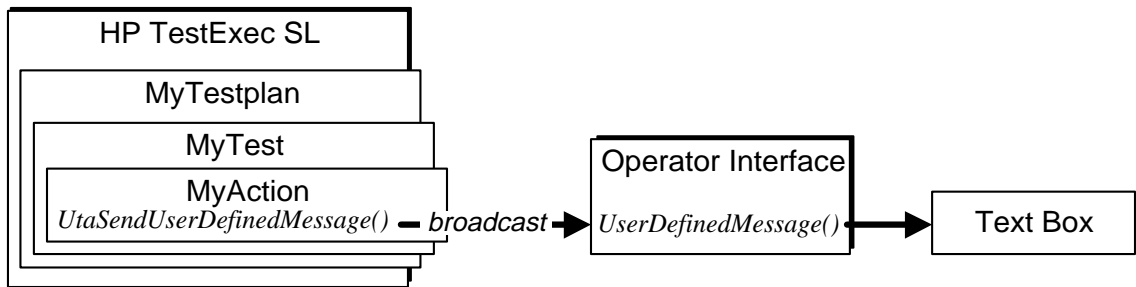
Events and methods for use in code written in Visual Basic

SendUserDefinedMessage	Method that broadcasts a message to all potential listeners and does not wait for a response
SendUserDefinedQuery	Method that broadcasts a message to all potential listeners and waits for a response
SendUserDefinedResponse	Method that responds to a user-defined message broadcast by SendUserDefinedQuery
UserDefinedMessage	Event that indicates a user-defined message has arrived

1. See “User-Defined Messages Reserved by Hewlett-Packard” for information about the range of message identifiers reserved for use by Hewlett-Packard.

Notice the parallels between the names and functionality of the functions and methods listed above. The C Action Development API has a set of functions that follow its conventions, and the Visual Basic environment its counterparts. Although you typically broadcast user-defined messages from one environment and listen or respond from another, there is nothing to prevent sending and receiving user-defined messages in the same environment.

Consider the following example of how a user-defined message might work.



Here, an action in a test executing in a testplan running in HP TestExec SL broadcasts a user-defined message to an operator interface, which then displays the message in a text box if the message is of interest.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

The pertinent code to do this inside an action routine, which uses a call to the `UtaSendUserDefinedMessage()` function, might look like this:

```
// Code in action routine written in C...
UtaSendUserDefinedMessage(5, "Hello from an action!");
// More code in action routine...
```

The code in the operator interface written in Visual Basic might implement the `UserDefinedMessage` event, which responds to user-defined messages, like this:

```
'Code in operator interface written in Visual Basic
Private Sub TestExecSL1_UserDefinedMessage (ID As Integer, TextBlock _
    As String)
    Select Case ID 'Evaluate the identifier
        Case 5 'The message is of interest
            txtMyTextBox.Text = TextBlock
        Case Else 'The message is not of interest
            Exit Sub
    End Select
End Sub
```

Besides evaluating `ID`, you could evaluate the message contained in `TextBlock`. For example, you could have `ID` identify a general class of messages, and `TextBlock` to carry the data associated with a specific message, as shown next.

```
// Code in action routine written in C...
// An ID of 2 is a pass/fail status message for a voltage measurement
if (Voltage > 5)
    UtaSendUserDefinedMessage(2, "passed");
else
    UtaSendUserDefinedMessage(2, "failed");
// More code in action routine...
```

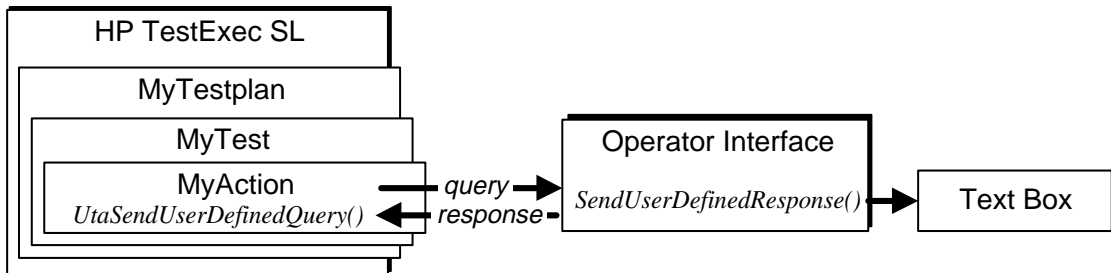
```
'Code in operator interface written in Visual Basic...
Private Sub TestExecSL1_UserDefinedMessage (ID As Integer, TextBlock _
  As String)
  Select Case ID 'Evaluate the identifier
    Case 2 'The message is of interest
      If (TextBlock = "passed") Then 'Evaluate the TextBlock
        txtTextBox.Text = "Voltage test passed"
      Else
        txtTextBox.Text = "Voltage test failed"
      Exit Sub
    End If
  End Select
End Sub
```

Similarly, TextBlock could contain other readable text, numeric data, or even patterns of bits for evaluation. Instead of writing a string to a text box, the code in the operator interface could do other tasks requested by action code executing in HP TestExec SL.

Note

An important concept to understand about user-defined messages is that they are broadcast to one or more *potential* recipients, or “listeners,” who must parse the messages to decide if their contents are of interest.

All of the examples so far have broadcast a message with no regard for whether the message was received or acted upon. Another variation on user-defined messages, which is shown below, lets you use the `UtaSendUserDefinedQuery()` function to broadcast a message to an intended recipient, and wait a specified length of time for the recipient to respond via a call to the HP TestExec SL control's `SendUserDefinedResponse` method.



Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

An example of code that does this is shown next.

```
// Code in action routine written in C...
// ID is 4 and timeout value is 2 seconds
if (UtaSendUserDefinedQuery(4, "Waiting for a response", 2) != NULL)
    ...code that does some task if response is received
else
    // exceeded time-out value or an error occurred
// More code in action routine...
```

The code in the operator interface might respond to a user-defined query like this:

```
'Code in operator interface written in Visual Basic...
Private Sub TestExecSL1_UserDefinedMessage (ID As Integer, TextBlock _
    As String)
    Select Case ID 'Evaluate the identifier
        Case 4 'The message is of interest & requires a response
            TestExecSL1.SendUserDefinedResponse 4, "Received your query"
        Case Else 'The message is not of interest
            Exit Sub
    End Select
End Sub
```

Note

You must use user-defined queries and responses as complementary pairs; e.g., you cannot use the `SendUserDefinedMessage` method to respond to a message broadcast by the `UtaSendUserDefinedQuery()` function.

The examples have shown user-defined messages and queries being sent from code in actions to operator interfaces, and user-defined responses being sent from operator interfaces to code in actions, but the opposite is possible. For an example, see “Accessing Hardware Resources from an Operator Interface.”

Note

You can find detailed descriptions of the C Action Development API functions in Chapter 2 of the *Reference* book and in HP TestExec SL’s online help. You can find detailed descriptions of the Visual Basic events and methods in the online help for the HP TestExec SL control.

User-Defined Messages Reserved by Hewlett-Packard

Hewlett-Packard has reserved numbers in the range of 10,000 to 99,999 for predefined user-defined messages used in operator interfaces provided with HP TestExec SL. If an existing message is appropriate for your use, feel free to reuse it. If you add new messages, be sure to add them somewhere outside the predefined range to avoid conflicts with operator interfaces from Hewlett-Packard.

You can browse the code in “frmPreDefinedTxSLUserMessages” in the sample operator interface for current definitions of the reserved messages.

Accessing Hardware Resources from an Operator Interface

Note

For an overview of hardware handlers, which are mentioned in the topics below, see “About Hardware Handlers” in Chapter 3 of the *Getting Started* book. Also, see “Monitoring the Status of Hardware” in Chapter 2 of this book.

When Do Operator Interfaces Access Hardware Resources?

Under what conditions does an operator interface need to access hardware resources, such as an I/O port that controls equipment or returns status information? Suppose that:

- You must have a safety shield in place before applying power to the UUT. The shield probably will have a safety interlock switch or sensor whose status your operator interface needs to know before it allows testing to begin.
- You want operators to control testing by pressing large, mechanical buttons marked Start and Stop on a separate “button box” near the test system instead of via graphical buttons on the operator interface.
- You want your operator interface to control colored lamps or other indicators that indicate the status of testing.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

- Your operator interface needs to control automated equipment without operators being present.

The alternatives when an operator interface written in Visual Basic needs to interact with hardware in typical scenarios like these are:

- The Visual Basic code in your operator interface can directly interact with hardware via an I/O strategy that is unique to the operator interface
- The Visual Basic code in your operator interface can interact with HP TestExec SL, which in turn interacts with hardware via its standard I/O strategy

For an operator interface written in Visual Basic, we recommend that you adopt the latter approach via the support for hardware handlers that is built into HP TestExec SL. See the sample code and “readme” file in directory “<HP TestExec SL home\samples\automate” for an example of a hardware handler that supports digital I/O operations.

Accessing the Hardware Resources

One way an operator interface can access hardware resources is by broadcasting a user-defined query to a hardware handler that is monitoring the status of hardware. Refer to the example below, which shows the code to implement a Run button that verifies closure of a safety interlock switch before testing begins.

```
'Code in operator interface written in Visual Basic
Private Sub cmdRun_Click()
    Answer = TestExecSL1.SendUserDefinedQuery 2, "", 5
    If Answer = "Yes" Then
        ...code that begins testing
    Else
        MsgBox "Please close the safety interlock", vbOKOnly
    End If
End Sub
```

Clicking the Run button calls the `SendUserDefinedQuery` method, whose associated code causes the operator interface to send a user-defined query whose identifier is 2, which we will assume is defined as an inquiry about the status of the safety interlock. If the query receives a valid response,

the response string is evaluated to determine the status of the safety interlock, which is used to decide if testing can begin.

As shown next, when the `AdviseUserDefinedMessage()` function in the hardware handler receives the message, it evaluates it, checks the status of the safety interlock, and returns that status in a user-defined response to the user-defined query sent by the operator interface.

```
// Code in hardware handler written in C
void UTADLL AdviseUserDefinedMessage(HUTAHWMOD hModule,
                                     HUTAPB hParameterBlock,
                                     LPVOID pUserInitData,
                                     long lID)
                                     LPCSTR lpszMessage)
{
  if (lID == 2)
    ...get status of switch via some I/O strategy
    if (SwitchClosed == 1)
      (UtaSendUserDefinedResponse(2, "Yes");
    else
      (UtaSendUserDefinedResponse(2, "No");
}
```

What About Concurrent Testing?

Note

This is an advanced topic that explores the concept of concurrent testing but does not fully describe how to implement it because the details can vary extensively from system to system.

Note

Concurrent testing, which runs multiple instances of HP TestExec SL—one per UUT—is not the same as multi-UUT testing, which tests multiple UUTs with a single instance of HP TestExec SL. Concurrent testing requires multiple sets of hardware resources, such as instruments that provide stimuli and measure responses, while multi-UUT testing lets you test multiple UUTs with a single, shared set of hardware resources. Multi-UUT testing is described in Chapter 10 of the *Using HP TestExec SL* book.

If desired, your operator interface can support concurrent or “parallel” testing if you need to simultaneously test multiple UUTs. It does this by

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

running two or more instances of HP TestExec SL at once. During concurrent testing, a single testplan controls different sets of hardware resources via multiple instances of the HP TestExec SL Control. This is done by having each set of hardware resources associated with its own switching topology file for the "fixture" layer in the switching topology.

A simple example of code to run two, simultaneous instances of HP TestExec SL is shown below. The example assumes that the associated form contains two instances of the HP TestExec SL Control, TestExecSL1 and TestExecSL2.¹ Notice how each instance of the HP TestExec SL Control has a unique topology file and directory for datalogging files.

```
Private Sub Form_Load()  
    'Set up & run first instance of testplan  
    TestExecSL1.Testplan.Preference.TopologyFiles("fixture").filename _  
        ="c:\Testplans\Fixture1\fixture1.ust"  
    TestExecSL1.Testplan.Preference.DatalogDirectory _  
        = "c:\Testplans\Fixture1"  
    TestExecSL1.LoadTestplan ("c:\Testplans\MyTestplan.tpa")  
    TestExecSL1.Testplan.Run  
  
    'Set up & run second instance of same testplan  
    TestExecSL2.Testplan.Preference.TopologyFiles("fixture").filename _  
        = "c:\Testplans\Fixture2\fixture2.ust"  
    TestExecSL2.Testplan.Preference.DatalogDirectory _  
        = "c:\Testplans\Fixture2"  
    TestExecSL2.LoadTestplan ("c:\Testplans\MyTestplan.tpa")  
    TestExecSL2.Testplan.Run  
End Sub
```

Comprehensive implementations of concurrent testing can be complex if you need to share hardware resources, such as instruments. If you need to share instruments or other resources, you must ensure that one instance of HP TestExec SL does not conflict with the other. For example, both instances cannot simultaneously use a single instrument to make a measurement. Instead, your operator interface or instrument drivers must implement a cooperative strategy for sharing resources.

1. You also could create a separate form for each instance of the HP TestExec SL Control, perhaps via Visual Basic's multiple-document interface (MDI).

Note

User-defined messages cannot communicate between instances of HP TestExec SL.

Miscellaneous Notes

- If you set a breakpoint in Visual Basic when debugging an operator interface, HP TestExec SL will not be aware of it. Because it is not aware of the breakpoint, HP TestExec SL will continue to send events to Visual Basic and those events will be lost as long as Visual Basic remains at the breakpoint.
- Do not use a Visual Basic Timer control to interact with the HP TestExec SL Control when the control is busy doing some task. As an alternative, you may wish to use the `AdviseUpdate` event, which triggers periodically when the control is not busy. For more information about the `AdviseUpdate` event, see the online help for the HP TestExec SL Control.
- Be aware that if code in your operator interface invokes a modal form, such as a modal dialog box or a message box, you may lose events triggered in the HP TestExec SL Control while the form is displayed. In other words, the control keeps generating events even if code in the operator interface cannot respond to them.

Changing or Enhancing Existing Functionality

Changing the Configuration of an Operator Interface

Module “modConfiguration” contains code that declares and defines the initial values of configuration variables used throughout the code in operator interfaces created in Visual Basic. See the code and comments in that module for more information.

A Quick Way to Hide Existing Functionality

Suppose you wish to remove a command button or other control from an operator interface to keep operators from using it. One way to remove the control is to actually delete it from a form and, ideally, also remove the code that implements its functionality. However, if you do this and later change

Operator Interfaces Created in Visual Basic

your mind, you must add the control again and replace the code you removed.

A more benign alternative to removing a control is to simply make it invisible at runtime by setting its `Visible` property to `False`. If desired, you also can resize it smaller and move it out of the way of other controls. This way, you can easily reuse the control if your needs change.

Note

If you actually remove a control instead of simply hiding it, be sure to search for all references to that control in the code for the operator interface and remove them as needed.

Controlling the Information That Appears in Reports

Accessing the Default Information

Report information is accumulated by HP TestExec SL during testing. In the Test Executive environment used to develop testplans, this information appears in the Report window, and you have the option of printing it. This same report information also is potentially useful in operator interfaces because it can be used to print status or repair tickets suitable for attaching to UUTs.

By default, the stream of report information sent to an operator interface is identical to that which appears in HP TestExec SL's Report window. As shown below, all it takes to make this information appear in a text box is several lines of code that define what happens when the `ReportMessage` event triggers at the end of a run of a testplan.

```
Private Sub TestExecSL1_ReportMessage(ByVal Message As String)
    'Assumes a text box name 'txtMyReportBox' appears on the form.
    'New message is appended to existing text to create a cumulative log
    txtMyReportBox.Text = txtMyReportBox.Text & Message
End Sub
```

Note

A `RichTextBox` automatically formats strings but a simple `TextBox` does not. For example, a `RichTextBox` correctly handles embedded carriage return/line feed characters in messages. In contrast, a `TextBox` requires you to format messages into individual lines by searching for these characters in

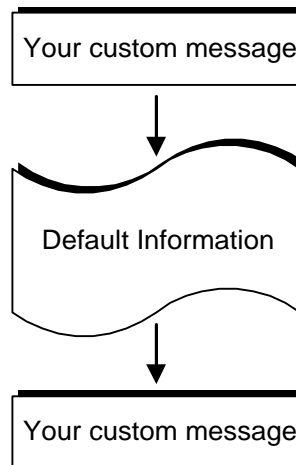
substrings and processing them. Thus, it is usually advantageous to use a RichTextBox when displaying report information.

Although the previous example is easy to understand, it is best suited to handling small amounts of report information because it is slow. If you need to work with a larger amount of report information, the example shown below is much faster.

```
Private Sub TestExecSL1_ReportMessage(ByVal Message As String)
    'Assumes a text box name 'txtMyReportBox' appears on the form.
    'New message is appended to existing text to create a cumulative log
    txtMyReportBox.SelStart = 2147483647 'Some large number <= MAXINT
    txtMyReportBox.SelLength = 0
    txtMyReportBox.SelText = Message
End Sub
```

What if Reports Need Additional Information?

While the default level of report information is useful in many cases, there may be others where you wish to customize the report information that appears. The easiest way to customize report information is to enhance it by adding additional information to the default stream of information. While you cannot add information within the default stream, you can easily add supplemental information before or after the default stream, as shown below.



Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

A single line of code can call the `SendReportMessage` method and specify the message to be sent, like this:

```
TestExecSL1.Testplan.SendReportMessage "Here is my custom message..."
```

You can call this event as needed from your operator interface. For example, placing this call in the procedure for the `AfterTestplanLoad` event as shown below would add your message to the beginning of the report stream each time a new testplan was loaded.

```
Private Sub TestExecSL1_AfterTestplanLoad(ByVal Path As String)
    TestExecSL1.Testplan.SendReportMessage "Testplan " & Path & _
        " was loaded." & vbCrLf
End Sub
```

And adding this message to the `AfterTestplanStop` event as shown in the following example would send your message to the end of the report stream when a testplan finished.

```
Private Sub TestExecSL1_AfterTestplanStop(ByVal Reason As _
    HPTestExecSL.TestplanState)
    If Reason = TestplanPassed Then 'If testplan finished successfully
        TestExecSL1.Testplan.SendReportMessage "Testplan passed at " & _
            Time & vbCrLf
    End If
End Sub
```

Similarly, you can call `SendReportMessage` from other events.

What if Reports Need Different Information?

If your need for report information differs greatly from the default, you have two choices. You can:

- Selectively parse the existing stream of report information collected by HP TestExec SL
- Ignore the default stream of report information collected by HP TestExec SL, and instead accumulate your own data during testing and manipulate it to provide report information

Because the parsing of the existing report information can be quite complex, we recommend that you adopt the latter approach. You can do this by:

- Turning off standard reporting by setting the value of the `StandardReportingEnabled` property to `False`, and
- Writing routines for the various events associated with the HP TestExec SL Control, and
- Having those routines read the values of properties associated with objects in the HP TestExec SL Control, and
- Sending the values directly to the operator interface, or manipulating the values—to provided calculated values, perhaps—and then sending the results to the operator interface.

Changing the Language

Which Languages Can I Use?

The language options built into the sample operator interface written in Visual Basic are English, German, and Spanish. This support includes multi-language captions for labels, such as those that appear on controls, and status messages. If desired, you can add new messages or support for additional languages but you must provide the message strings yourself.

Changing the Default Language

Changing languages is simple if the new language you wish to use is one of those provided with the predefined operator interface, and if your operator interface does not require additional controls or messages beyond those already defined.

The value of a global variable named `guLanguage` in module “`modConfiguration`” determines which language is used; i.e., which set of messages is used from an array of messages in various languages. To change the default language, simply change the value of this variable. For example, the entry that sets the default language to English looks like this:

```
guLanguage = English
```

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

To change the default language to Spanish, you would edit the entry as shown below.

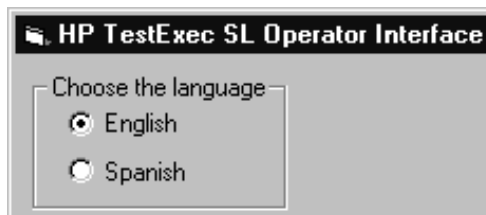
```
guLanguage = Spanish
```

Note

When changing languages you may need to resize the fields that display labels and messages to accommodate their lengthened or shortened captions. For more information about how text expands or contracts from language to language, see “What About Multiple Languages?”

Switching Among the Built-In Languages

As described above, the value of a global variable named `guLanguage` in module “`modConfiguration`” determines which language appears in the operator interface. If desired, you can dynamically change the value of this variable and call a subroutine to refresh the captions of controls used in the operator interface. For example, suppose those who use your operator interface need the ability to switch between languages “on the fly” as shown below.



The implementation code for the two `OptionButton` controls used above is shown below.

```
Private Sub optEnglish_Click()  
    guLanguage = English 'Change the language  
    UpdateControlCaptions 'Update the control captions  
End Sub
```

```
Private Sub optSpanish_Click()  
    guLanguage = Spanish 'Change the language  
    UpdateControlCaptions 'Update the control captions  
End Sub
```

In a similar fashion, you could switch languages based upon some other means of identifying which language is required, such as associating a specific language with each operator's login.

How Does Multi-Language Support Work?

Traditionally, supporting multiple languages in applications written in Visual Basic has required using a separate resource compiler, such as the one provided with Visual C++. However, the operator interface created in Visual Basic that is provided with HP TestExec SL supports multiple languages without using a resource compiler. It relies on an array of strings, `gsLangArray`, in module "modLocalization" that contains sets of messages in various languages. These messages determine the captions for the labels that appear for buttons, text boxes, and such.

An excerpt from the declaration of the array's contents looks like this:

```
guLanguage = English  
... more entries  
gsLangArray(gnRun, guLanguage) = "Run"  
gsLangArray(gnRun1, guLanguage) = "&Run"  
... more entries for the English language  
  
guLanguage = German  
... more entries  
gsLangArray(gnRun, guLanguage) = "Laufen"  
gsLangArray(gnRun1, guLanguage) = "&Laufen"  
... more entries for the German language
```

Notice how the definition for each message is repeated in both languages shown. For example, the first entry in the set of English messages, "Run", has a corresponding entry named "Laufen" in the set of German messages. Each string whose value is declared in the array corresponds to a label on a control or a status message used by the operator interface.

Note

You should duplicate these definitions line for line across the various languages. For example, if the definition for the English language defines a

Operator Interfaces Created in Visual Basic

dozen strings—i.e., has a dozen lines of code in it—so should the definitions for the other languages. Although you must specify a value for each string in the default language, you have the option of specifying a null string (“”) for other languages. If you specify a null string for an entry, then the definition of that entry in the default language will be used.

Where a simple label has a single entry associated with it, a command button has two entries. For example, the “Run” button is defined as:

```
gsLangArray(gnRun, guLanguage) = "Run"  
gsLangArray(gnRun1, guLanguage) = "&Run"
```

The first entry defines the button’s name as it appears in status messages, and the second defines how the name appears on the button itself. This duplication of entries is needed to support access keys¹; e.g., the ampersand character (&) preceding the second entry defines “R” as the “Run” button’s access key, but you would not want “&Run” to appear in messages that refer to the button.

What About Languages That Are Not Built In?

Note

Module “modLocalization” contains predefined, commented entries that you can uncomment and use as a starting point when adding support for a new language.

If your operator interface needs to support a language that is not already built into the operator interface provided with HP TestExec SL, and if your operator interface does not add new controls or messages, do the following in module “modLocalization” for each new language:

1. Find the `MAX_LANG` constant in the general declarations at the beginning of the module. Increase its value by one for each new language you add. For example, if the operator interface already supported three languages you would enter:

```
Const MAX_LANG = 4
```

-
1. An “access key” lets you press the ALT key and type a designated letter to activate a control or menu item.

2. Find the declaration for enumerated type `TxSLOPUIInterfaceLanguage` in the general declarations at the beginning of the module and add an entry for the new language to it. For example:

```
Public Enum TxSLOPUIInterfaceLanguage
    English = 1
    German = 2
    Spanish = 3
    NewLanguage = 4
End Enum
```

3. Find the declaration for enumerated type `txslLangIndex` in the general declarations at the beginning of the module, as shown below.

```
Public Enum txslLangIndex
    gnAbort = 1
    gnAbout = 2
    ...more declarations
End Enum
```

Near the end of the existing declaration, add an entry for your new language. Make its value one greater than the last entry in the existing list. For example, the entry for the German language looks like this:

```
gnGerman = 69
```

Your new entry in `txslLangIndex` might look like this:

```
Public Enum txslLangIndex
    gnAbort = 1
    gnAbout = 2
    ... more declarations
    gnYieldlparam = 363 ` Last existing entry
    gnNewLanguage = 364 ` Your new entry
End Enum
```

4. In subroutine `InitializeLangArray`, find the entries that specify the English language members of array `gsLangArray`. Copy all of these entries to the bottom of the subroutine, where they will subsequently be modified to support the new language.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

Note

Be sure to paste the entries ahead of `guLanguage = tempLanguage`, which appears at the very end of the subroutine.

5. Change the text in all the messages to their equivalents in the new language, as shown in the examples in “How Does Multi-Language Support Work?”
6. Add an entry in the form of

```
gsLangArray(gnNewLanguage, guLanguage) = "Language"
```

for your new language to the definition for each language in array `gsLangArray`. The value at the index in `gsLangArray`, *Language*, should be in the local language.

For example, the entries in the definition for the English language look like this:

```
gsLangArray(gnEnglish, guLanguage) = "English"  
gsLangArray(gnFrench, guLanguage) = "French"  
gsLangArray(gnGerman, guLanguage) = "German"
```

And the entries in the definition for the German language look like this:

```
gsLangArray(gnEnglish, guLanguage) = "English"  
gsLangArray(gnFrench, guLanguage) = "Französisch"  
gsLangArray(gnGerman, guLanguage) = "Deutsch"
```

Notice how the value at the index in `gsLangArray` varies from language to language. Because this example demonstrates support for three languages, the third being French, the definition for the French language also would contain an equivalent set of these entries.

7. At the beginning of the new section, change the value of `guLanguage` to the new language. For example, the entry for the German language looks like this:

```
guLanguage = German
```

8. If you want the new language to be the default, change the value of `guLanguage` in subroutine “Configure” in module “modConfiguration”. For example, to make the German language the default you would enter:

```
guLanguage = German
```

If your operator interface needs to support a language whose characters cannot be represented in an 8-bit character set—such as Japanese, Chinese, or Korean—keep the following in mind:

- Do the language conversion on a system that has the appropriate fonts and tools installed for the new language. For example, if your operator interface needs to support the Japanese language, the system you use probably would have a Japanese version of Microsoft Windows, Japanese fonts selected, and a keyboard that lets you enter Japanese characters.
- Each control affected by the new language should have its font changed to one that supports the new language. Assuming the appropriate font is installed, you can do this by selecting the control and changing its font in Visual Basic’s Properties Window.
- If you need to switch between fonts while the operator interface is running, see the example in subroutine `ChangeFonts` in `modLocalization`.

Adding Language Support for a New Control

The definitions in `gsLangArray` contain entries for each of the controls used in the sample operator interface. If you add more controls to an operator interface, you must add support for them by doing the following:

1. In the general declarations section of module “modLocalization”, add new entries for each of the new controls. Assign their numeric values in sequence after the existing definitions.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

The example below shows added entries for a new command button named MyButton. The values 12 and 13 assume they are the twelfth and thirteenth entries in the list.

```
...at the end of the existing definitions  
'Begin user-defined controls  
Public Const userMyButton As Integer = 12  
Public Const userMyButton1 As Integer = 13
```

Note

As shown above, we recommend that you prefix your additions with “user” so it is obvious that they are custom entries. This makes your changes easier to identify if a future version of HP TestExec SL provides an updated version of module “modLocalization” in which you wish to reuse your custom code.

2. Expand array gsLangArray in subroutine InitializeLangArray by adding entries for the new control. The listing below continues with the MyButton example begun above.

```
...at the end of the existing definitions  
'Begin user-defined controls  
gsLangArray(userMyButton, guLanguage) = "My Button"  
gsLangArray(userMyButton1, guLanguage) = "&My Button"
```

Here, we are assuming that the entries for “My Button” and “&My Button” are the twelfth and thirteenth lines in the definition, which corresponds to the values 12 and 13 assigned to them in module “Localization.bas”.

3. In routine UpdateControlCaptions in module “modAppSpecific”, add an entry to update the label on each new control, like this:

```
...at the end of the existing definitions  
'Begin user-defined controls  
Form1.cmdMyButton.Caption = LangLookup(userMyButton1)
```

Adding Language Support for a New Message

In most respects, strings that contain messages for use in text boxes are no different from strings that contains captions for controls. Each message has

its own definition in `gsLangArray`, and each definition is associated with a numeric value. If the language is changed by altering the value of variable `guLanguage`, the appropriate string is read. You can add new message strings to the array as needed.

The static strings described above work well for simple, unvarying messages. However, some messages must change dynamically to be informative. For example, suppose you wished to display a message that read:

```
Test MyTest was started at 11:50:00 AM
```

You probably would not want to have the name of the test permanently “hard coded” into your operator interface. Also, you would want to dynamically update the time, perhaps by calling Visual Basic’s `Time` function.

To address this need, the predefined operator interface provided with HP TestExec SL includes utility routines whose names begin with “`txslFormatString`” (such as `txslFormatString1`) that are used to format messages that contain replaceable parameters. Each time the message is displayed, its definition is looked up in `gsLangArray` for the appropriate language and the current values of its replaceable parameters are updated.

At design time (in code as opposed to at run time), the message string for the example above might look like this:

```
Test %1 was started at %2
```

See the code in the predefined operator interface for more comprehensive examples of how this works.

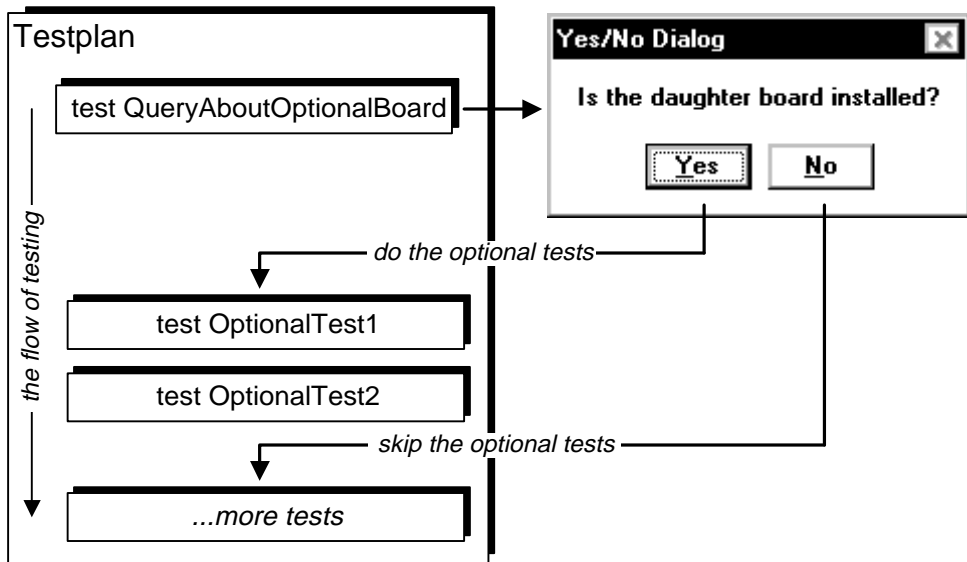
Prompting a System Operator from HP TestExec SL

There may be times when an operator interface is running a testplan and the testplan needs to display a prompt, let the operator respond to the prompt, and then take action based upon the operator’s response. For example, suppose you are testing a module that may or may not have an optional daughter board attached to it. If the daughter board is present, you need to run additional tests that exercise the board’s functionality. Also, suppose there is no reliable way to programmatically test for the presence of the daughter board.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

At the beginning of testing, you need to prompt the operator to visually inspect each module and indicate whether the optional daughter board is present. A straightforward way to do this is by displaying a dialog box that contains an appropriate prompt, such as “Is the daughter board installed?,” and provides Yes and No buttons for the operator’s response.¹ A conceptual diagram of this is shown below.



HP TestExec SL provides several predefined actions that support prompting of system operators from testplans. Located in directory “<HP TestExec SL home>\actions”, they are:

StdDialogOkay	Displays a prompt in a dialog box. The only possible response is OK.
StdDialogOkayCancel	Displays a prompt in a dialog box. Possible responses are OK and Cancel.

1. Alternatively, the operator could respond by pressing a mechanical Yes or No button connected to an I/O port that interfaces with an appropriate hardware handler.

StdDialogYesNo	Displays a prompt in a dialog box. Possible responses are Yes and No.
StdDialogYesNoCancel	Displays a prompt in a dialog box. Possible responses are Yes, No, and Cancel.

Note For more information about these actions, see their descriptions in HP TestExec SL's online help.

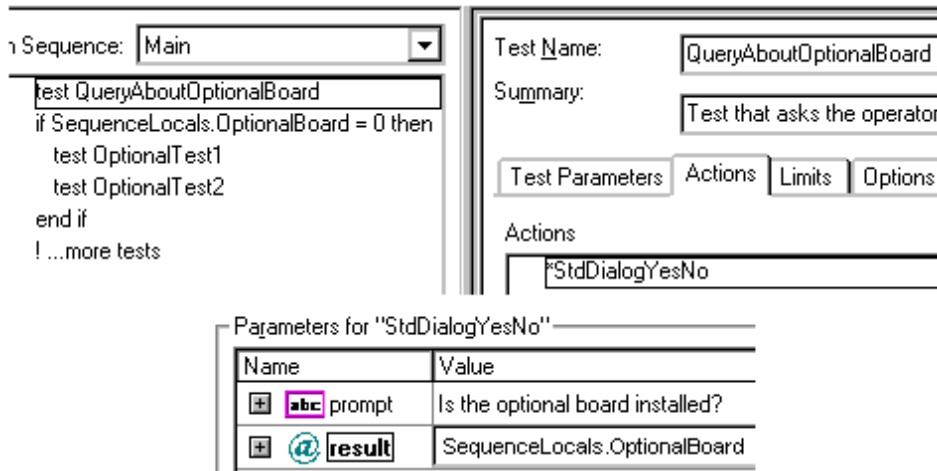
Each of these actions broadcasts a user-defined query to see if a listener, such as an operator interface, is available to display a dialog box. If it receives a response from a listener, the action broadcasts a second user-defined query and waits for a listener to respond by indicating that a button was pushed in response to the query. If the action does not receive an initial response from a listener, it displays its own dialog box.

Note For more information about user-defined queries and responses to them, see "Understanding User-Defined Messages."

Continuing with the example begun earlier, you could use the `StdDialogYesNo` action to let the operator specify whether the optional board is present. The excerpts from HP TestExec SL's Testplan Editor window shown below assume that a test in the testplan contains the `StdDialogYesNo` action, and that the result returned by the action is associated with a symbol named `OptionalBoard` in the `SequenceLocals` symbol table.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic



If the operator presses the Yes button in response to the query, the `StdDialogYesNo` action returns a value of zero to `OptionalBoard`. Testing the symbol's value with the "if...then" statement causes additional tests, `OptionalTest1` and `OptionalTest2`, to execute. Had the operator chosen the No button, the optional tests would have been skipped.

We stated earlier that if the action does not receive an initial response from a listener, it displays its own dialog box. Given that, you may wonder why you would want code in the operator interface to display a dialog box instead of simply letting the action display the box. A major benefit of having the operator interface display the dialog box is that operator interfaces created in Visual Basic support multiple languages. Thus, text in a dialog box displayed by the operator interface will appear in whichever language the operator interface is currently using.

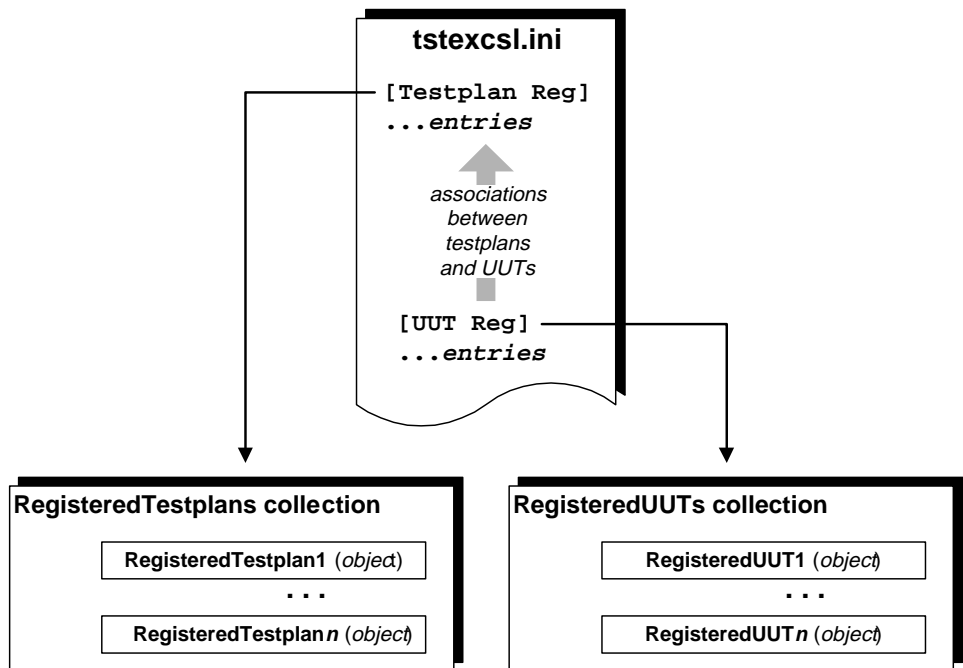
Associating Testplans & UUTs with an Operator Interface

The HP TestExec SL Control's `RegisteredTestplans` collection can contain one or more `Testplan` objects that identify which testplans the code in an operator interface can load and run. Similarly, the control's `RegisteredUUTs` collection can contain one or more `RegisteredUUT` objects that identify which UUTs are available for testing.

Because each testplan can be used to test one or more UUTs, a relationship exists between testplans and UUTs. For example, a testplan named

“Testplan1” might be used to test “UUT_TypeA” and “UUT_TypeB”, while another testplan—“Testplan2”, perhaps—tests only “UUT_TypeC”. Thus, “Testplan1” has “UUT_TypeA” and “UUT_TypeB” associated with it, while “Testplan2” has “UUT_TypeC” associated with it.

These associations let code in the operator interface automatically load the correct testplan when a UUT’s bar code is read. Associations between testplans and UUTs are made via entries in sections of the HP TestExec SL initialization file, “<HP TestExec SL home>\bin\tstexsl.ini”, as shown below.



Also, the initialization file’s contents define the items that code in an operator interface uses to populate the HP TestExec SL Control’s `RegisteredTestplans` and `RegisteredUUTs` collections.

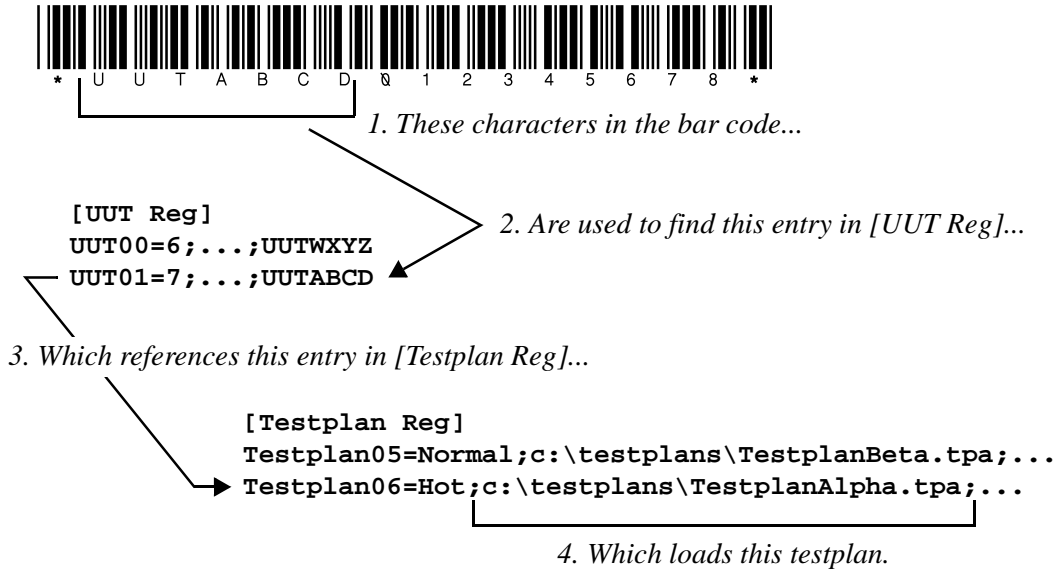
Note

You must manually edit the initialization file to define which testplans and UUTs are available, and any relationships between them. See “To Register a Testplan for an Operator Interface” and “To Register a Testplan for an Operator Interface” in Chapter 1 of the *Using HP TestExec SL* book.

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

The example below provides an overview of how these associations work when reading a bar code that identifies a unique UUT. Suppose the first seven characters of the bar code contain the type of UUT, which in this example is “UUTABCD” The remaining characters are the serial number of an individual UUT.



When the bar code is scanned, code in subroutine `HandleBarCode` in form `frmMain` in the operator interface parses it and stores the first seven characters in string variable `UUT_Type`. The other nine characters are stored in string variable `UUT_SerialNumber`. The value of `UUT_Type` is used to look for a matching string of characters in the “name” field in an entry in the `[UUT Reg]` section of the initialization file. Here, the possible names are “UUTWXYZ” and “UUTABCD”.¹

The value of `UUT_Type` matches “UUTABCD” in the entry in `[UUT Reg]` that contains `UUT01=7`, where 7 is an index that references a related entry in the `[Testplan Reg]` section of the initialization file. Because the index is 7, the related entry in `[Testplan Reg]` is `Testplan06`,² which

identifies “TestplanAlpha.tpa” as the testplan to load for the UUT whose type is “UUTABCD”.

Note

For a more detailed description of the syntax of entries in the initialization file, see the comments for the [Testplan Reg] and [UUT Reg] sections in that file.

Using Peripherals with Operator Interfaces

Which Peripherals are Supported?

The sample operator interface created in Visual Basic supports the following peripheral devices:

- Strip printer with serial interface, such as the HP E1199B
- Bar code reader that connects inline with the keyboard, such as the Symbol Technologies, Inc. model LS 3603MX connected to a keyboard “wedge” interface
- Bar code reader with serial interface, such as the Symbol Technologies, Inc. model LS 3603MX connected to a serial interface
- Keypad that connects inline with the keyboard, such as the Cherry model ML4700
- Digital I/O hardware, such as the HP E1330B VXI Quad 8-Bit Digital Input/Output module or an M-Module

The “One Peripheral Per Form” Convention

In general, each peripheral device has its own form in Visual Basic. The code in a form associated with each peripheral handles interaction with the peripheral even though other forms or modules may be the actual users of the peripheral. For example, form “frmStripPrinter” acts as a container for controls used to interact with a strip printer. Also, the form contains code

-
2. There is an offset of 1 between indexes in [UUT Reg] and numbered entries in [Testplan Reg].
-

Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

that configures the I/O port and does some formatting of report output. However, the actual report information sent to the strip printer is acquired elsewhere from HP TestExec SL.

The forms provided for interacting with peripherals include:

frmStripPrinter	Used with a strip printer
frmSerialBarcode	Used with a serial bar code reader
frmTxSLSharedIO	Used with I/O resources that are shared by an operator interface and HP TestExec SL, such as a digital I/O card used to read external Run and Stop buttons

Note

Peripherals that connect inline with the keyboard, such as keypads and “keyboard wedge” bar code readers, do not have separate forms because they simply emulate the behavior of the keyboard.

If you add other peripherals, we recommend that you follow this “one peripheral per form” convention so you can quickly identify the sections of code associated with peripherals. Also, you probably will want to include Boolean variables in module “modConfiguration” to indicate the presence or absence of any optional peripherals whose forms you include in an operator interface.

Using Bar Code Readers

Note

See “Associating Testplans & UUTs with an Operator Interface” for information about how scanning a UUT’s bar code can automatically load the appropriate testplan.

About Bar Code Readers

A bar code reader provides quick and error free entry of information that is used repetitively, such as operator logins or serial numbers. Some bar code readers resemble guns that you point at the bar code being read, while others

are wands that you wave across the bar code or horizontal scanners over which you pass the bar code.

Regardless of which type of bar code reader you use, their typical characteristics include:

- Support for some method of triggering that tells them when to read the bar code. For example, a bar code “gun” usually has a physical trigger to pull, while a serial bar code reader may expect a signal that tells it when to read the bar code.
- They interface with a PC via some I/O strategy, such as inline with its keyboard (usually the easiest to implement) or via a serial port.
- They can be programmed to specify which, if any, optional leading or trailing characters are sent to help identify what is contained in the bar code, or to identify the end of transmission. For example, the bar code reader may append a carriage return/line feed to a bar code. When code in the operator interface sees this combination of characters, it knows the complete bar code has been received.

Because these characteristics can vary with the model of bar code reader, make sure you are familiar with how your bar code reader operates, and how to program it appropriately.

Changing the Processing of Bar Codes

When a bar code is read, code in subroutine “HandleBarCode” in form “frmMain” processes it. This processing includes checking the bar code’s validity, parsing it into separate strings that contain a UUT type and serial number, and loading the appropriate testplan according to the type of UUT.

Browse the code in “HandleBarCode” to see how bar codes are processed by default, and modify the code as needed if your bar code scheme is different from the default.

Testing the Code for Bar Code Readers

HP TestExec SL provides several sample testplans and related files you can use to see if your bar code reader can successfully read bar codes. Located in directory “<HP TestExec SL home>\samples\uidebug\testplans”, they are:

5Passes.tpa	Testplan with five tests that pass
5Failures.tpa	Testplan with five tests that fail
TypesAndLimits.tpa	Testplan that illustrates simple data types and the limit checkers used with them. Also, sends a message to the stream of report data during testing. Has two variants, Normal and WithDelay, where WithDelay inserts delays into tests to slow them so you can watch the testplan as it executes.

Note

You may need to modify the search paths for actions and DLLs to make these testplans work; see “Specifying the Search Path for Libraries” in Chapter 5 of the *Using HP TestExec SL* book.

Shown below are bar codes you can use to test bar code readers. They work with the testplans described above, and with the default associations between testplans and UUTs specified in HP TestExec SL's initialization file (see "Associating Testplans & UUTs with an Operator Interface") Below each bar code is the text it contains.

Loads the 5Passes testplan



Loads the 5Failures testplan



Loads the 5Failures testplan



Loads the TypesAndLimits testplan



Loads the TypesAndLimits testplan



Loads the TypesAndLimits testplan with the WithDelay variant



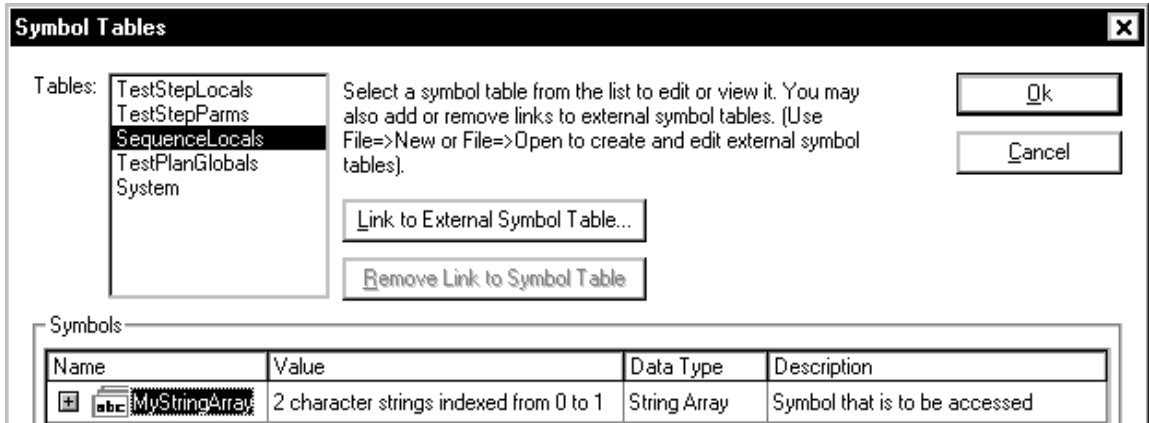
Accessing Symbol Tables from an Operator Interface

If desired, you can access a symbol in one of HP TestExec SL's symbol tables from code written in Visual Basic. Suppose the SequenceLocals

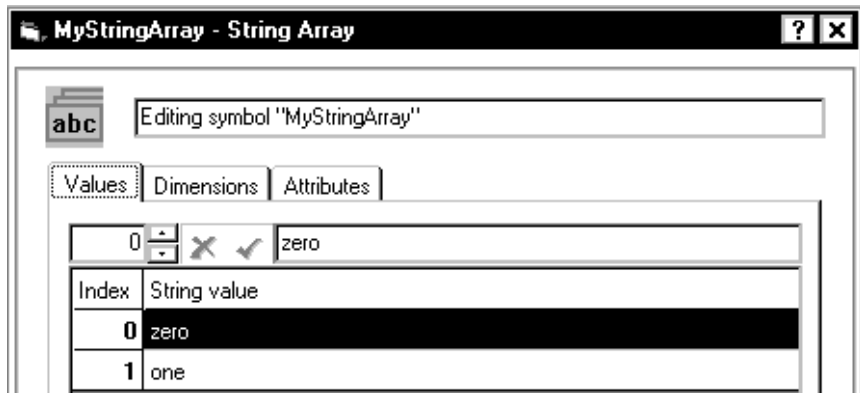
Customizing the Operator Interface

Operator Interfaces Created in Visual Basic

symbol table has a symbol named `MyStringArray` defined in it, as shown below.



Also, suppose that the symbol's definition looks like this:



You could use the following code associated with the “click” event for a command button in an operator interface created in Visual Basic to read the value of the symbol and display it in Visual Basic’s Immediate window.

```
Private Sub cmdTestStringArrayAccess_Click()  
    Dim SymbolTableArray As Variant  
    SymbolTableArray = TestExecSL1.SymbolTables _  
        ("SequenceLocals").Symbols("MyStringArray").Value  
    Debug.Print "This is the value at index 0: " & SymbolTableArray(0)  
    Debug.Print "This is the value at index 1: " & SymbolTableArray(1)  
End Sub
```

The output in the Immediate window would be:

```
This is the value at index 0: zero  
This is the value at index 1: one
```

Note the need to define a variable of type Variant when accessing an array, and then read the value from the symbol table into the variable before using it. If desired, you could also write to the symbol and modify its value; for example,

```
SymbolTableArray(1) = "NewStringValue"
```

For more information about symbol tables, see Chapter 5 in the *Using HP TestExec SL* book.

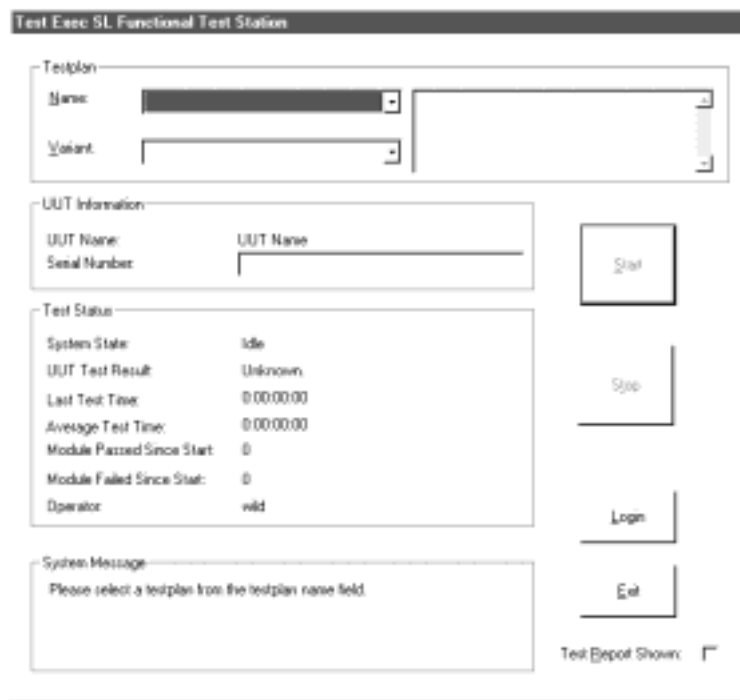
Operator Interfaces Created in Visual C++

Note

The sample operator interface provided with HP TestExec SL that is created in Visual Basic has some automation features built into it, while the sample operator interface created in Visual C++ does not.

What is the Standard Operator Interface in Visual C++?

The standard operator interface created in Visual C++ is shown below. The code for its project is in directory “<HPTestExec SL home>\opui”.

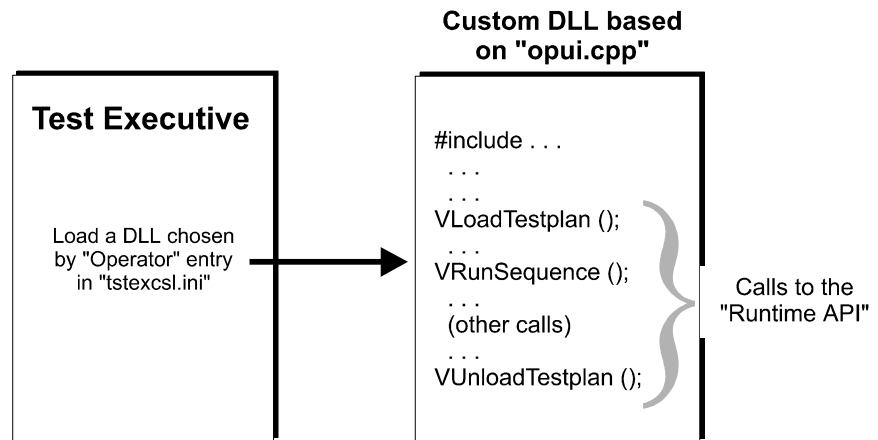


Inside an Operator Interface in Visual C++

Before you create a specific operator interface in Visual C++, you should be familiar with how operator interfaces work in general when using Visual C++. The next several topics describe the underlying concepts for operator interfaces developed in Visual C++.

Overview

Refer to the illustration below. When someone logs in to HP TestExec SL they are identified as the member of a group, such as “Operator.” The Test Executive loads whichever user interface, or “personality,” is associated with the user's group in file “<HP TestExec SL home>\tstexsl.ini”. By default, the code for the operator interface is stored as a DLL in file “opui.dll”.



The DLL for the operator interface contains calls to functions in what is called the “Runtime API” because it is an API that lets you vary how the Test Executive appears to its users. This custom API provides the functions a user interface needs to interact with the Test Executive. Besides calls to the Runtime API, the DLL also contains whatever supporting code you write for use with the calls, such as functions to display status information or make buttons appear on the screen.

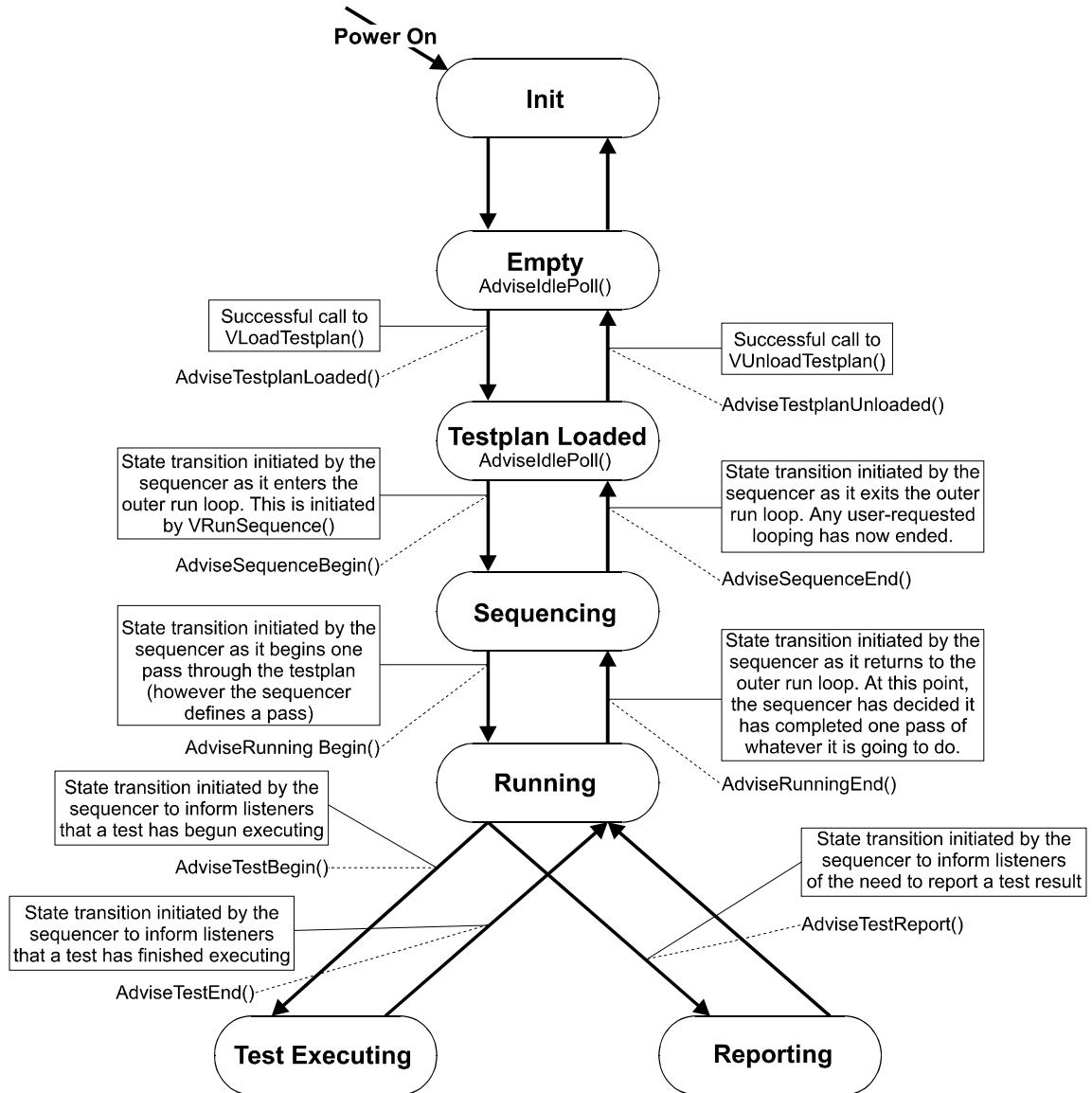
For descriptions of the functions in the Runtime API, see Chapter 5 in the *Reference* book.

How the Operator Interface Requests Service

When the custom DLL containing code for the operator interface is loaded, the Test Executive acts as a server that responds to requests from the operator interface. For example, if the Test Executive is acting as a server then clicking a button on the user interface could have the DLL call a function in the Runtime API that causes the Test Executive to begin executing a testplan.

Shown next is a state diagram for the Test Executive when it acts as a server for an operator interface created in Visual C++. The states it can assume are

Init, Empty, Testplan Loaded, Sequencing, Running, Test Executing, and Reporting. Arrows show transitions from one state to another.



A transition from one state to another occurs in response to a request made by a call to the appropriate function. A testplan must be loaded before

Operator Interfaces Created in Visual C++

proceeding to any of the run states. Thus, a normal start-up process is to use the `VLoadTestplan()` function to bring the Test Executive from the Empty state to the Testplan Loaded state. Once this has been done, you can begin running a testplan by calling function `VRunSequence()` in response to a “Run” button or an automation handler. When the test sequencer exits at the end of the testplan, the Test Executive returns to the Testplan Loaded state and is ready to accept a new request to run the testplan.

Most state transitions call a related internal function that generates an advisory message you can receive via a callback routine if the code in your operator interface has registered interest in that event. For example, the initial transition from Sequencing to Running (down the left side of the state diagram) has routine `AdviseRunningBegin()` associated with it. If you use the related callback routine, `VRegisterRunningBegin()`, in your operator interface to register interest in this event, the operator interface will be notified when this state transition occurs. You could use knowledge of this event to print a “Testing...” status message for operators to let them know when testing is underway. In a similar fashion, you can register interest in other events and act upon them.

Accessing Global Data from the Operator Interface

Besides requesting service from the Test Executive when it acts as a server, an operator interface also needs to interact with global data for the test system. For example, the operator interface may need to know the name of the current testplan, the pass/fail status of the previous test, or the results from a measurement.

A specialized set of functions in the Runtime API lets your code in the operator interface access global data stored in the System symbol table; see “Predefined Symbols” in Chapter 5.

Interacting with the Test Sequencer

The standard Test Executive provides a simple sequencer that is optimized for the needs of high throughput, “Go/No-Go” testing applications. For more demanding applications, you may want to augment that sequencer’s functionality. Although the basic sequencer will always be there because it houses data structures used by HP TestExec SL, you can cause your own sequencer to be run instead of the one provided by Hewlett-Packard.

The framework in which test sequencers operate in HP TestExec SL provides a set of built-in controls whose settings are accessible via API functions. These controls are available to any sequencer that is loaded, and we recommend that your replacement sequencer honors these controls to the extent it can.

API functions you can use to determine under what conditions the sequencer halts, and in which state it halts, are:

```
VConfigureHaltOnFailure()  
VConfigurePauseOnFailure()  
VConfigureNoHalt()  
VGetHaltMode()  
VGetFailCountLimit()
```

Two other API functions let you run the testplan repeatedly:

```
VConfigureCountedLoops()  
VConfigureTimedLoops()
```

The last two APIs are mutually exclusive; i.e., you cannot simultaneously use counted loops and timed loops.

See Chapter 5 in the *Reference* book for the full syntaxes of these API functions.

Creating an Operator Interface in Visual C++

You must be proficient with Visual C++ and the Microsoft Foundation Class Library (MFC) to do the tasks described below.

Note

Details of the steps described below will vary according to which brand and version of compiler you use.

1. Create a new directory and copy the project files for the operator interface supplied by Hewlett-Packard to it.

The original project files are located in subdirectories beneath directory “<HP TestExec SL home>\opui”.

Operator Interfaces Created in Visual C++

2. Rename the new project. For example, the original project is named “opui” and it creates “opui.dll” as its output. Given this, you might call yours “my_opui”.
3. Start your compiler and open the project in the directory you just created.
4. In your compiler, specify the locations of the project's files in the directory you created.
5. Edit the source files as needed to modify the operator interface.

If you want to change the underlying functionality of the operator interface, edit file “opui.cpp”. To modify the appearance of the operator interface, edit the visual resources associated with it.

6. Compile the source files into a DLL.

For convenience, you can compile a debug version of the file, use it, and not bother recompiling it as a release version. The additional overhead from debug code in this single DLL is negligible.

7. Copy the modified DLL to HP TestExec SL's “bin” directory so that it replaces the existing file “opui.dll”.

Tip: If desired, you can give your new DLL (and its project) an entirely different name from the original. If you do, be sure to edit “tstexcs1.ini” so the “Operator=” entry in its [Components] section specifies the name of the new DLL for the operator interface.

Doing Specific Tasks with an Operator Interface in Visual C++

Responding to a “Run” Button

During its init, the personality registers callbacks for AdviseSequenceBegin, AdviseSequenceEnd, and AdviseRunningEnd.

The user or system integrator creates an operator interface form as a DLL following the Runtime API guidelines. It is probably a MFC-based DLL. The operator interface presents a Run button as a control on the form. A

function named `OnRunClick()` is created and associated with the click event of the Run control. The `OnRunClick` function calls `VRunSequence()` (Runtime API routine). If the system is successful in bringing the sequencer to the Running state the operator interface gets back its `AdviseSequenceBegin` callback. The interface should note the running state on its display (e.g. light the Run button) and return. At the end of the testplan the interface code will get first the `AdviseRunningEnd` callback and then the `AdviseSequenceEnd` callback. The testplan end callback will be the only one received if the user pressed Pause, assuming it was provided by the interface. At this point control returns to the statement following the `VRunSequence` call.

Beginning a Test Cycle

During init, the controller registers the `AdviseIdlePoll`, `AdviseSequenceBegin`, and `AdviseSequenceEnd` callbacks.

Let's take a simple scenario first. Assume only one testplan is being run on this station for a long time.

Assume the init of the personality reads the fixture code and loads the uses it to determine the testplan to load. Perhaps it has a ".ini" file that provides the lookup function. It then reads the path of the testplan to automatically load and manually loads it (`VLoadTestplan()`).

At this point the system is in the Testplan Loaded state and the `AdviseIdlePoll` is being polled continuously during the system idle time. When the automation interface detects a load command from the automation handler, it reads the bar code of the UUT, decodes it, verifies that the module type is correct, and stuffs the serial number into the proper symbol table parm for use by the rest of the system. It then calls the `VRunTestplan()` API.

The sequencer takes charge now and calls the `AdviseSequenceBegin` callback as it begins executing the testplan tests. When the testplan completes it calls the `AdviseSequenceEnd` callback. The automation interface now does whatever operation is necessary to release the UUT, and signals the automation handler. Control returns to the original `AdviseIdlePoll` function where the run was initiated. It simply returns. The system is now back in Testplan Loaded state polling for automation activity.

The complex scenario is just like this except that when the interface reads the module type it determines that a different testplan is required. It calls

VUnloadTestplan() and calls VLoadTestplan() for the new one. Now VRunTestplan() can be called to pick up the scenario just as the simple case.

Displaying the Name of the Current Test

Register to receive the AdviseTestBegin and AdviseTestEnd callbacks. The AdviseTestBegin callback will supply the handle of the currently executing test. The operator interface form can then use VGetTestName () to get the name of this test statement and display it on the form. The end callback will inform the interface to cancel it.

Displaying the Testplan and Test Timing

Register to receive the AdviseRunningBegin, AdviseRunningEnd, AdviseTestBegin, and AdviseTestEnd callbacks. The testplan begin/end calls will bracket the time to execute one pass through the testplan. If the user wishes to track average passing time and average time to first failure they can distinguish them by tracking the overall pass/fail state by looking at the state of each test at the AdviseTestEnd callback. The execution time of each test may be determined by the test begin/end bracketing calls.

Displaying Messages

The user interface can register its interest in the AdviseUserDefinedMsg event. When the test programmer broadcasts a message via the VSendUserMessage() function, it will be routed to the callback function. It is the responsibility of the user interface provider to display the message. The tag parameter can be used to communicate any user defined information, such as screen location to post in, “severity”, color, etc.

Beginning When the Testplan Name is Unknown

There are times when the actual testplan to use cannot be identified until a UUT is loaded or scanned. In this situation, the personality will only bring the system up to the Empty state and wait there until the user presses run or the handler indicates it is ready. Then the UUT type/testplan can be identified, the proper testplan loaded, and the run initiated.

Creating an Automation Interface in Visual C++

The topics in this section suggest ways in which you might handle the implementation of an automation interface in Visual C++.

Note

Note: Look in either of two places for descriptions of the API calls referenced in subsequent topics in this section. Calls whose names begin with “Uta” belong to the C Action Development API, and calls whose names begin with “V” are part of the Runtime API. Both APIs are documented in the printed *Reference* book and in online help.

Software Configuration for an Automation Interface

The “testexsl.ini” file contains configuration options you must set when an automation interface is present:

- A flag to indicate the presence of an automation interface. This tells the test system to skip the normal log-in sequence
- A pointer to the DLL that contains the automation interface.
- A flag that causes failure ticket information to be sent to the default printer.

See also: “Setting Up an Automation Interface” in Chapter 6 of the *Using HP TestExec SL* book

Choosing a Task Model in Windows

Your choice of a Windows task model affects the construction of your automation interface. For example, Windows applications can use a task model in which there is only a single thread of execution. When a process has control of the execution thread, other processes cannot run. This means that under the single-thread model you cannot assume that your automation interface is actively running in the background while other Windows processes are active.

This can also create conflicts with the Windows “event-driven” approach in which nothing happens until an event (such as the user clicking a button) occurs. The nature of the event determines the next system action. If the

Customizing the Operator Interface

Operator Interfaces Created in Visual C++

automation interface does not keep this “event loop” alive, the system may stop responding to events.

If you implement your automation interface as a modal dialog, it will control its own event loop. To poll for external events that would initiate a test cycle, the interface must coordinate the test system execution thread with the Windows event loop. You can find many techniques for implementing this approach in Windows programming references.

A pseudo-code example of the method looks like this:

```
Msg msg;
while (m_bKeepGoing)
{
    if (pollAutomationHandler())
        initiate Test();
    if (PeekMessage (&msg, hDialog, 0, 0, PM_REMOVE))
        if (!IsDialogMessage(hDialog, &msg))
        {
            Translate Message (&msg);
            Dispatch Message (&msg);
        }
}
```

A simpler technique uses some of the built-in support provided by functions in HP TestExec SL's Runtime API. If you implement your automation interface as a non-modal dialog, the normal HP TestExec SL event loop stays active. You can register a request for idle polling and the API will continually call your poll routine as long as the system is idle.

Here is a pseudo-code example of this technique:

```
void MyIdlePoll (WORD state)
{
    if ((vstate)state < VSequencingState)
        if (pollAutomationHandler())
            initiate Test();
}
```

Related API Functions are:

- VRegisterIdlePoll()
- UtaKeepAlive()

Using a Bar Code Reader

In an automated production line, bar code readers usually connect to the test system by an RS-232 serial interface port. In a typical scenario, the automation interface receives a signal that a board has arrived. The automation interface then triggers the bar code reader to scan an identifier label on the UUT. The automation interface uses a symbol table to look up the UUT type and the corresponding testplan. HP TestExec SL then loads and runs the appropriate test for the UUT.

If the bar code reader is under the control of the test system, then it can be triggered to read a bar code UUT identifier when a UUT arrives to test. The bar code reader typically responds by sending the identifier read back to the test system by a serial port. The buffer size of the serial port can be an issue if the identifier is longer than the 16 characters handled by a standard 16550 UART in a test system's serial interface. For longer identifiers, you may have to use a serial interface card with a larger buffer to avoid losing UUT identifier characters.

Related API Functions are:

```
VLoadTestplan()  
VRunSequence()
```

Monitoring Test Results

Once started by the `VRunSequence()` function, the testplan runs to conclusion (or continues until interrupted by the user) before returning control to the automation interface. When the testplan finishes, it halts and returns a code. The code indicates whether the system had a normal or abnormal completion. You can use this code as a reasonably accurate pass/fail indicator, but the code may not adequately reflect the true situation. For example, you may have set the test system options to ignore all failures or to continue until a given number of failures has accumulated.

A better way to determine the test results is to register a callback for the Report event using the `VTestJudgement()` function to determine whether the test passed or failed. You can accumulate information about the pass or fail status of each test, then use that information to decide the next action for the automation interface (display a message, notify an automation controller, etc.). For example, a UUT may have to fail tests 2 and 4 out of tests 1-4 before the overall result is a failure for that UUT. You would

Customizing the Operator Interface

Operator Interfaces Created in Visual C++

accumulate the pass/fail information in a table, then use the table to make the final determination.

Related API Functions are:

- VRunSequence()
- VTestJudgement()
- VRegisterTestReport()
- Miscellaneous functions for callback registration

Displaying Messages to the User Interface

The automation interface must display whatever system status information required for the operator interface. Most of the events of interest to the automation interface are available by runtime API callback registration. Typical information desired might be the name of the current testplan, the state of the tester (running, stopped, etc.), the pass/fail status of the current or previous UUT, the name of the test currently executing, and the current or previous failure report.

Related API Functions are:

- Functions for callback registration (`VRegister...`)
- Functions for interacting with system data (`VGet...`)

Also, the full Visual C++ MFC class library is available to handle Windows interactions.

Responding to Keyboard and Mouse Commands

The automation interface must handle the keyboard and mouse input for the test system. See the example operator interface for details on ways to handle this input.

Related API Functions are:

(none)

The full Visual C++ MFC class library is available to handle this Windows interaction.

Generating Repair Information

There is no default report window or standard reporting feature. As a system integrator, you have total control over the contents, format, display, and printing of your test reports. However, you can use a simple built-in function, `StandardTestReport()`, to easily format reports.

Generating a report trace is a two-step process. Although the process may at first seem awkward, it gives you flexibility. First, your user interface needs to register three callbacks—`VRegisterTestReport()`, `VRegisterSendReportMsg()`, and `VRegisterClearReport()`—to generate the report content. Second, you route that content to its destination.

The `VRegisterTestReport()` callback gives you a handle to the test that just executed. You can then use the `VTestJudgement()` function to determine whether the test passed or failed. To trace the test, call the function `StandardTestReport(HUTEST, Cstringd)`. It returns a formatted string in the `Cstringd` reference parameter. Include “`rprtfmt.h`” and link to “`rprtfmt.lib`” to use the standard report format.

If the standard format is unsuitable or if you do not want to alter the format for all interfaces by replacing “`rprtfmt.dll`”, create your own format. The API provides several useful functions. The most important one for reporting is `VGetLimitsandRest2()`.

Related API Functions are:

- `VRegisterTestReport()`
- `VRegisterSendReportMsg()`
- `VRegisterClearReport()`
- `VTestJudgement()`
- `VGetLimitsandResult2()`

Writing Repair Tickets

Some automated lines require that the test system print a failure ticket to be attached to the UUT or its carrier after testing finishes. The automation interface must control this operation if the test system manages the printing.

You may also choose to implement a “paperless” repair ticket scheme in which you do not necessarily want to print a repair ticket on the spot if a UUT fails. For example, the automated production line, based on pass/fail

Operator Interfaces Created in Visual C++

information from the test system, may automatically route failing boards to a repair loop. There a technician would reread the UUT bar code to call up a display of the failure information.

To accommodate the paperless scheme, you can use HP TestExec SL's built-in datalogging features to output test results in a file. You can then route the results elsewhere (such as a main automation controller) for formatting and using the information in an overall repair ticket.

The sample user interface provided with HP TestExec SL includes an example of how to print failure information.

Signaling Downstream Devices

After the testplan completes, it is the responsibility of the automation interface to signal downstream automation equipment or the central computer that the test is complete and to indicate the test results. This could be by a message sent on a serial or LAN interface, depending on the specifics of the automation system.

Datalogging

One outcome of testing may be to send datalogging information to a quality management system. This capability is built into HP TestExec SL. The automation interface does not have to handle datalogging tasks unless you need a format or protocol not supported by the built-in datalogging features.

LAN Communications

Some automation systems may require that the test system communicate with a central automation controller by a LAN interface. The test system's automation interface must then provide the correct communications protocol and messaging. You can add LAN communications using existing Windows networking features. Consult the relevant Windows documentation for more information.

Dealing with Problems

Sometimes the automation line experiences problems to which the test system must respond. Examples include downstream equipment failures that force the test system to stop accepting more UUTs to test. The automation

interface must handle such exceptions as part of its interactions with other automation equipment or the central automation computer.

Other problems may arise when HP TestExec SL raises an exception testing a particular UUT. There is an error sequence testplan that tries to put the UUT and test system in a safe state when an exception occurs. The error sequence, however, also may experience an exception for a particularly bad problem. The automation interface must then decide how to continue after such exceptions occur. This may include the following:

- Notifying downstream automation equipment or the central computer of the exception.
- Deciding, based on the exception type and severity, if the test system should continue testing the next UUT.
- Deciding to exit HP TestExec SL if there is an extreme error condition.

Related API Functions are:

UtaExcRegIsError()
UtaExcRaiseUserError()
UtaExcRegClearError()
VAppExit()
VStopSequence()
VPauseSequence()

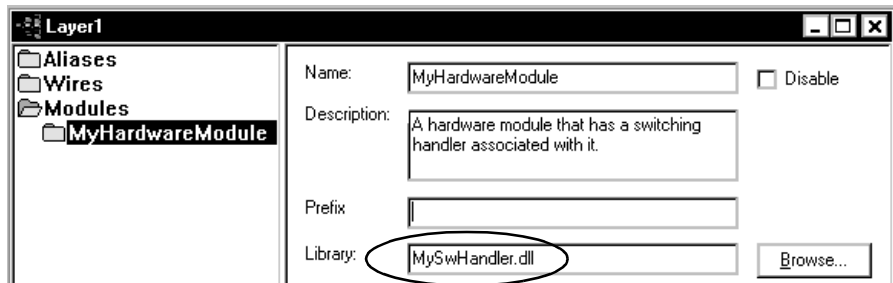
Creating a Hardware Handler

This chapter describes how to create a hardware handler, which is a software layer written in C/C++ that knows how to communicate with a hardware module and is aware of the module's internal topology.

For more information, see Chapter 3 in the *Getting Started* book, Chapter 4 in the *Using HP TestExec SL* book, and Chapter 3 in the *Reference* book.

Writing a Hardware Handler

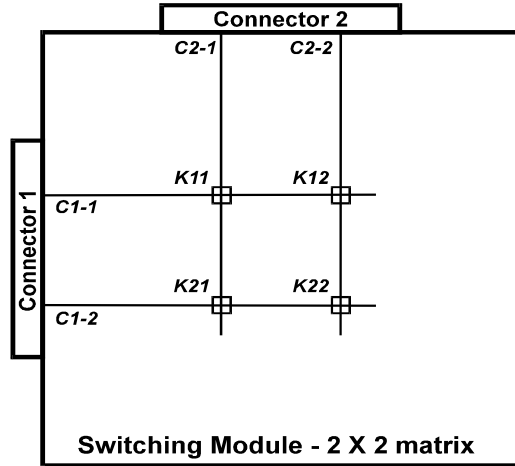
The following topics describe how to implement a hardware handler. Once you have created the hardware handler, you can specify it when using the Switching Topology Editor to define switching topology, as shown below. This makes HP TestExec SL aware of the hardware handler.



Modeling Your Hardware

As an aid when creating a hardware handler, label a diagram of the switching elements inside modules so that each connector, node, and switching element is uniquely identified.

The example below shows a simple 2x2 switching matrix with its features labeled.



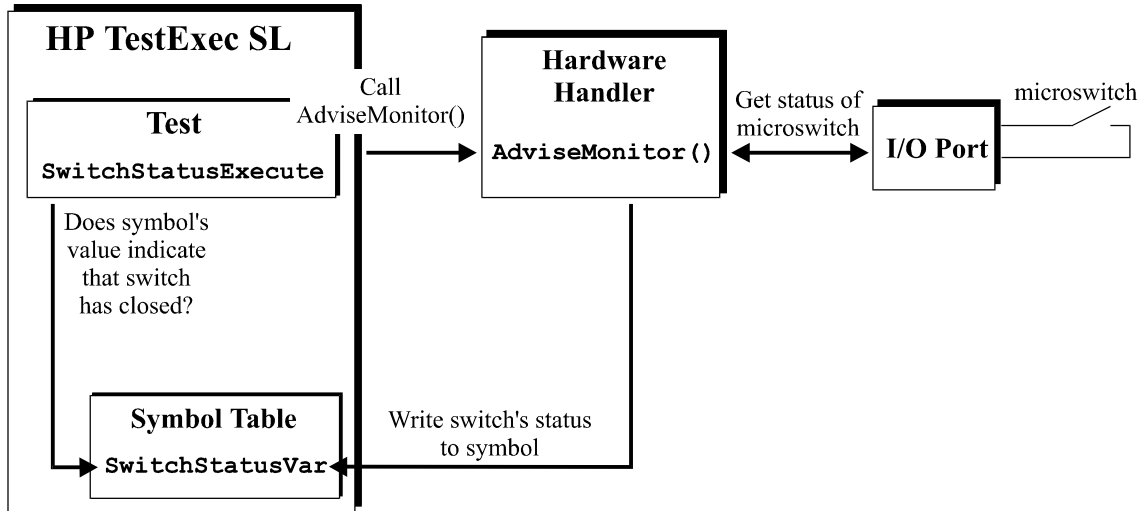
Monitoring the Status of Hardware

There may be times when you want HP TestExec SL to interact with very simple hardware, such as switches or sensors. For example, suppose your test system is connected to an automated handler that uses a microswitch to indicate when the UUT is correctly positioned for testing. HP TestExec SL

Creating a Hardware Handler

Writing a Hardware Handler

can “watch” for closure of that switch and then continue testing when the switch closes. One method for doing this is shown below.



The example works like this:

1. A test being run by HP TestExec SL contains an action named `SwitchStatusExecute` whose code (written by you) loops until the value of a symbol named `SwitchStatusVar` in a global symbol table indicates that the microswitch has closed. For example, the value of `SwitchStatusVar` might be 1 when the switch is closed and 0 when it is open.
2. Meanwhile, HP TestExec SL is periodically calling a function named `AdviseMonitor()` in the hardware handler.

For information about specifying how frequently HP TestExec SL calls the `AdviseMonitor()` function, see “Specifying the Polling Interval for Hardware Handlers” in Chapter 6 of the *Using HP TestExec SL* book.

3. When `AdviseMonitor()` is called, its implementation code (written by you) interrogates an I/O port—a serial port, perhaps—to see if the microswitch is closed or open.

4. When it receives the status of the microswitch, `AdviseMonitor()` writes that status to `SwitchStatusVar` in the symbol table.
5. When the value of `SwitchStatusVar` finally becomes “true,” the `SwitchStatusExecute` action stops looping and testing continues.

The options you have when choosing where to implement an `AdviseMonitor()` function are:

- You can put the function in the handler for a hardware module or instrument.
- You can create a minimal hardware handler whose sole purpose is to monitor the status of hardware, and put the function in it.

Besides the code that implements the `AdviseMonitor()` function, all the minimal hardware handler needs are the mandatory functions required in every hardware handler, which are `DeclareParms()`, `Init()`, `Close()`, and `Reset()`.

Creating a Project for the Hardware Handler

Note

The topics in this section describe how to use the development environment provided with Microsoft Visual C++ 6.0. If you are using another C/C++ development environment, the details will vary but the concepts will be similar.

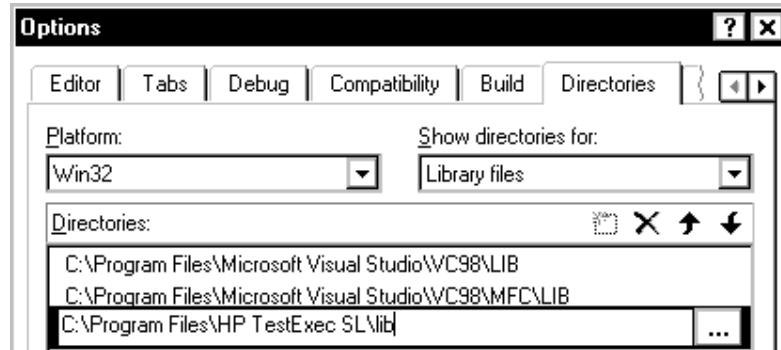
Specifying the Path for Libraries

1. Choose Tools | Options in the Visual C++ menu bar.
2. In the Options box, choose the Directories tab and specify a path for library files that includes the “lib” directory beneath the home directory

Creating a Hardware Handler

Writing a Hardware Handler

in which HP TestExec SL is installed on your system. An example is shown below.

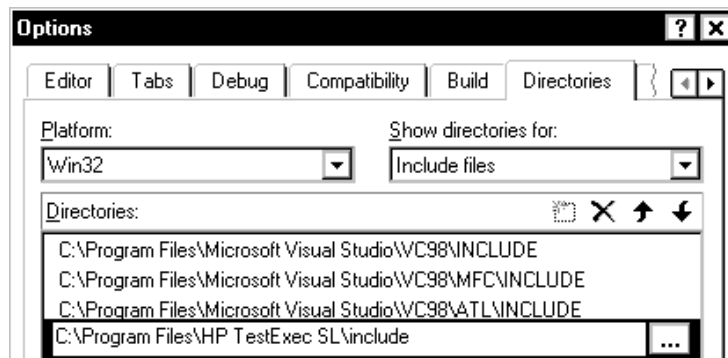


Note

Depending upon where you installed Visual C++ and HP TestExec SL on your system, your paths may vary from those shown.

Specifying the Path for Include Files

1. In the Options box, specify a path for include files that includes the “include” directory beneath the home directory in which HP TestExec SL is installed on your system. An example is shown below.



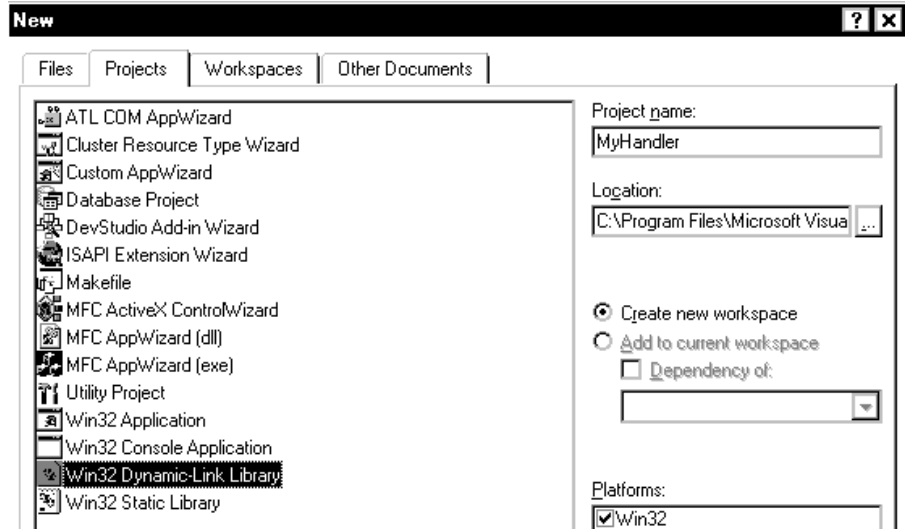
2. Click the OK button to save the path you specified.

Note

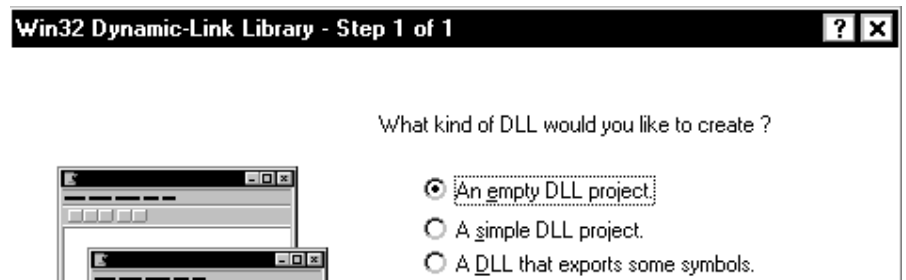
Depending upon where you installed Visual C++ and HP TestExec SL on your system, your paths may vary from those shown.

Creating a New DLL Project

1. Choose File | New in the Visual C++ menu bar.
2. Choose the Projects tab and specify Win32 Dynamic-Link Library as the type of project, as shown below.



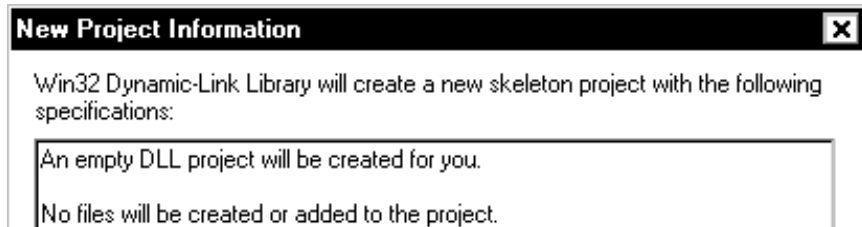
3. Type a Name for your project.
4. Specify the Location for your project.
5. Choose the OK button.
6. Choose to create an empty DLL project, as shown below.



Creating a Hardware Handler

Writing a Hardware Handler

7. Choose the Finish button.
8. Verify the information for the new project, as shown below.

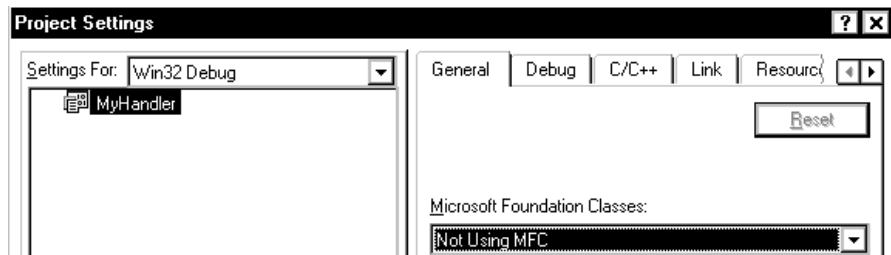


9. Choose the OK button.

Specifying the Project Settings

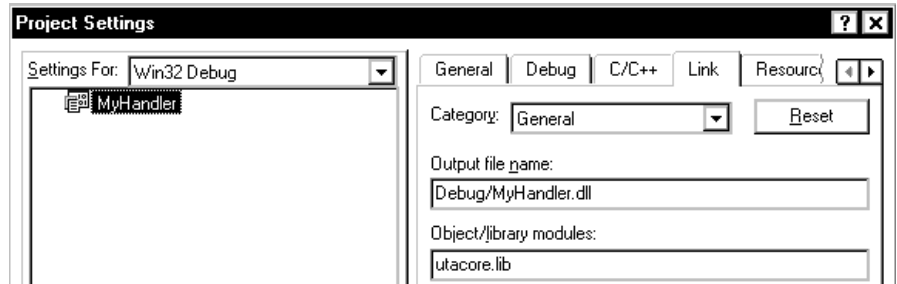
You specify the project settings once for each new project you create.

1. Choose Project | Settings in the Visual C++ menu bar.
2. If needed, choose the General tab to make its options visible.
3. In the Project Settings box, specify “Not Using MFC” for the Microsoft Foundation Classes option, as shown below.



4. Choose the Link tab to make its options visible.

5. Specify “utacore.lib” for the “Object/Library modules” option, as shown below.



Linking against “utacore.lib” lets the compiler resolve all the external references to HP TestCore definitions and functions used in your switching handler code. Because you already specified the default library path earlier, you do not need to enter the full path here.

6. Choose the OK button to save the project settings and close the Project Settings box.

Creating an Implementation File for the Hardware Handler

Note

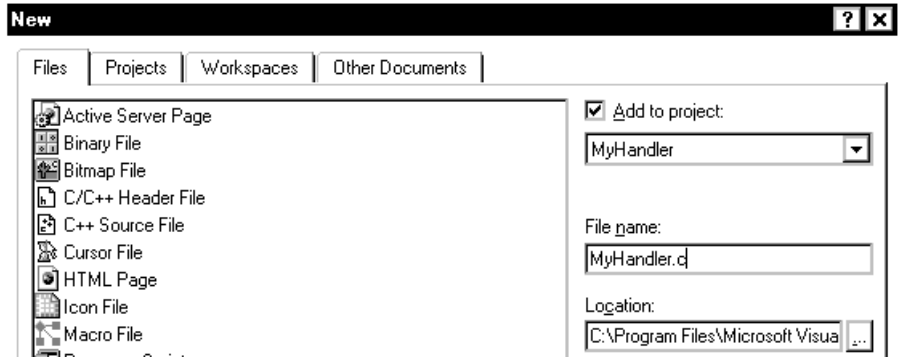
Directory “<HP TestExec SL home>\include” contains a header file named “switch_hndl.h” that declares the prototypes for the functions used in a hardware handler. If desired, you can “include” this file in your implementation file to ensure that you call the functions correctly. Or, you may even want to copy this file into your implementation file as a starting point when implementing the functions. If you use this file as a template in your implementation file, be sure to replace the UTA API macros in it with UTADLL.

1. Choose File | New in the Visual C++ menu bar.

Creating a Hardware Handler

Writing a Hardware Handler

2. On the Files tab in the New box, specify the file's type, name¹, and location, as shown below, and choose the OK button to add it to your project.



3. Type the file's contents in the editor window that appears.

Writing the Routines for Functions in the Implementation File

Note

Be sure to specify “`#include <uta.h>`” at the beginning of the implementation file for a hardware handler.

As a minimum, all hardware handlers require `DeclareParms()`, `Init()`, `Close()`, and `Reset()` functions in them. Even if these functions do nothing, they must be present or HP TestExec SL will generate an error when attempting to call them. The example shown below includes additional, switching-specific functions because it is for a switching handler, which is a common type of hardware handler used to control switching hardware. If you need to know more about a particular function, look it up in Chapter 3 in the *Reference* book.

1. Add an `Init()` function to initialize or “open” a hardware module.

Code that you write to implement this function, *which must appear in all hardware handlers*, should do whatever is needed to initialize the

1. Unless you have a specific reason for writing a hardware handler in C++, use a “.c” extension for your implementation file so it matches the examples.

hardware module. For example, you can allocate the memory for a structure to hold transient data used by a function in your hardware handler.

2. Add a `Close()` function to close a hardware module opened with the `Init()` function.

Code that you write to implement this function, *which must appear in all hardware handlers*, should do whatever is needed to close the hardware module, such as freeing or deleting any memory associated with a structure created in the `Init()` function.

3. Add a `DeclareParms()` function to declare parameters passed to the hardware handler when it is loaded for execution. (You specify the actual values of the passed parameters when you use the Switching Topology Editor to define switching topology.)

Code that you write to implement this function, *which must appear in all hardware handlers*, should call the `UtaHwModDeclareParm()` function to declare any parameters needed to distinguish one instance of the hardware module from another, such as the module's GPIB address, VXIbus slot number, etc. Also, you can use this function to pass configuration parameters, such as a parameter that chooses between 2x8 and 4x4 multiplexer configurations in a switching module.

One use for this function is to pass a parameter that identifies a specific switching module among several modules of the same type. The code you write in other functions in the switching handler, such as `SetPosition()` and `GetPosition()`, uses this parameter to address a specific module according to the I/O or driver strategy used by your module.

4. Add a `Reset` function to define what happens when the switching module is reset, and return the amount of time it will take to finish resetting, if any.

Code that you write to implement this function, *which must appear in all hardware handlers*, should do whatever is needed to reset the hardware module to whatever you want its default state to be. For example, the

Creating a Hardware Handler

Writing a Hardware Handler

code needed to control switching elements depends on what kind of I/O or driver strategy your switching module uses.

A `Reset` function for the 2x2 matrix example might look like this:

```
UTAUSECS UTADLL Reset (HUTAPB hParameterBlock, LPVOID pBindData)
{
    ...(code that opens relay K11 via I/O strategy for module)
    ...(code that opens relay K12 via I/O strategy for module)
    ...(code that opens relay K21 via I/O strategy for module)
    ...(code that opens relay K22 via I/O strategy for module)
    return TIME_TO_RESET;
}
```

5. Add a `DeclareNodes()` function to declare the nodes inside the switching module and any connectors that exist on the switching module. Also, declare the adjacencies, which are two nodes in the switching topology that can be connected by a switching element, in the switching module.

Note that the `DeclareNodes()` function uses calls to the `UtaHwModDeclareNode()` and `UtaHwModDeclareAdjacent()` APIs. The node names you assign in calls to the `UtaSwModDeclareNode()` API are the same node names that appear when the Switching Configuration Editor is used to define switching topology.

A `DeclareNodes()` function for the 2x2 matrix example might look like this:

```
void UTADLL DeclareNodes (HUTAHWMOD hModule, HUTAPBDEF hParmBlockDef)
{
    // Declare the nodes
    UtaHwModDeclareNode (hModule, "C1-1", "Connector 1, Pin 1", NULL);
    UtaHwModDeclareNode (hModule, "C1-2", "Connector 1, Pin 2", NULL);
    UtaHwModDeclareNode (hModule, "C2-1", "Connector 2, Pin 1", NULL);
    UtaHwModDeclareNode (hModule, "C2-2", "Connector 2, Pin 2", NULL);
}
```

```
// Declare the adjacencies -- i.e., pairs of adjacent nodes --  
// connected via a switching element  
UtaHwModDeclareAdjacent (hModule, "C1-1", "C2-1", 11, 1);  
UtaHwModDeclareAdjacent (hModule, "C1-1", "C2-2", 12, 1);  
UtaHwModDeclareAdjacent (hModule, "C1-2", "C2-1", 21, 1);  
UtaHwModDeclareAdjacent (hModule, "C1-2", "C2-2", 22, 1);  
}
```

6. Add a `SetPosition()` function to define the open and closed positions for switching elements that connect adjacent nodes in the switching module.

A `SetPosition()` function for the 2x2 matrix example might look like this:

```
UTAUSECS UTADLL SetPosition (HUTAPB hParameterBlock, LPVOID pBindData,  
    IDUTASWELM element, IDUTASWPOS position)  
{  
    switch (element)  
    {  
        case 11: // switching element 11 is relay K11  
            ...(code that opens/closes relay K11 based on value of  
            ..."position" via I/O strategy for module)  
            break;  
        case 12: // switching element 12 is relay K12  
            ...(code that opens/closes relay K12 based on value of  
            ..."position" via I/O strategy for module)  
            break;  
        case 21: // switching element 21 is relay K21  
            ...(code that opens/closes relay K21 based on value of  
            ..."position" via I/O strategy for module)  
            break;  
        case 22: // switching element 22 is relay K22  
            ...(code that opens/closes relay K22 based on value of  
            ..."position" via I/O strategy for module)  
            break;  
    }  
}
```

As with the `Reset()` function described earlier, this function requires that you provide the actual code needed to open and close switching elements in your specific switching module. The contents of this code

Creating a Hardware Handler

Writing a Hardware Handler

will vary with the I/O or driver strategy used by your switching module. In the case of the simple relays used in this example, your code would open the relay if “position” was passed a value of 0 and close it if the value was 1.

You also may want to include error checking routines in this function (or any of the functions, as needed). For example, here you could range check the value of “position” to ensure it is 0 or 1 before programming a switching element to a new position.

7. Add a `GetPosition()` function to return the current position of a specified switching element.

A `GetPosition()` function for the 2x2 matrix example might look like this:

```
IDUTASWPOS UTADLL GetPosition (HUTAPB hParameterBlock, LPVOID
    pBindData, IDUTASWELM element)
{
    int nPosition;
    ...(code that assigns "position" the status of switching
    ...element "element" via I/O strategy for module)
    return (nPosition);
}
```

The code you provide here interrogates the switching module, via its particular I/O or driver strategy, to determine if the switching element is opened or closed. Then it assigns `nPosition` a value of 1 if the switching element is closed, or 0 if it is opened.

8. (*optional*) Use `UtaHwModTrace()` or `UtaHwModTraceEx()` to send status messages to HP TestExec SL's Trace window during normal testplan execution.
9. (*optional*) Use `DeclareStatus()` and `GetStatus()` to send status messages to HP TestExec SL's Watch window during debugging.
10. (*optional*) Use `AdviseMonitor()` to monitor the status of hardware.

11. (optional) Use `AdviseUserDefinedMessage()` to respond to user-defined messages and control hardware based on the contents of those messages.

For an example of a hardware handler, search for “example, sample code for a hardware handler” in online help and see the sample files in directory “<HP TestExec SL home>\samples\filterdemo”.

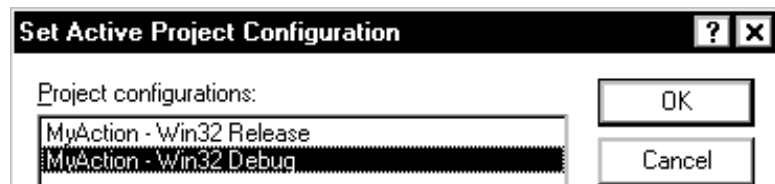
Verifying the Project’s Contents

- Choose the FileView pane in the Visual C++ workspace window to verify the contents of your project, as shown below.



Choosing Which Configuration to Build

1. Choose Build | Set Active Configuration... in the Visual C++ menu bar.
2. Specify that you wish to build a debug version of the project, as shown below.



Note

The debug version of a program contains additional code that makes it larger and slower to execute than a release version. Thus, you probably will want to build a final, release version of the DLL after you have debugged it.

Creating a Hardware Handler

Writing a Hardware Handler

3. Choose the OK button.

Building the Project

- Choose Build | Build *<project name>* in the Visual C++ menu bar to build the DLL.

Copying the DLL to Its Destination Directory

Each time you modify the DLL that contains your hardware handler, you must recopy it to directory “*<HP TestExec SL home>*\bin”.

Note

If desired, you can simplify copying DLLs for hardware handlers by creating a custom tool similar to the one described for copying DLLs for actions under “Copying the DLL to Its Destination Directory” in Chapter 3 of the *Using HP TestExec SL* book.

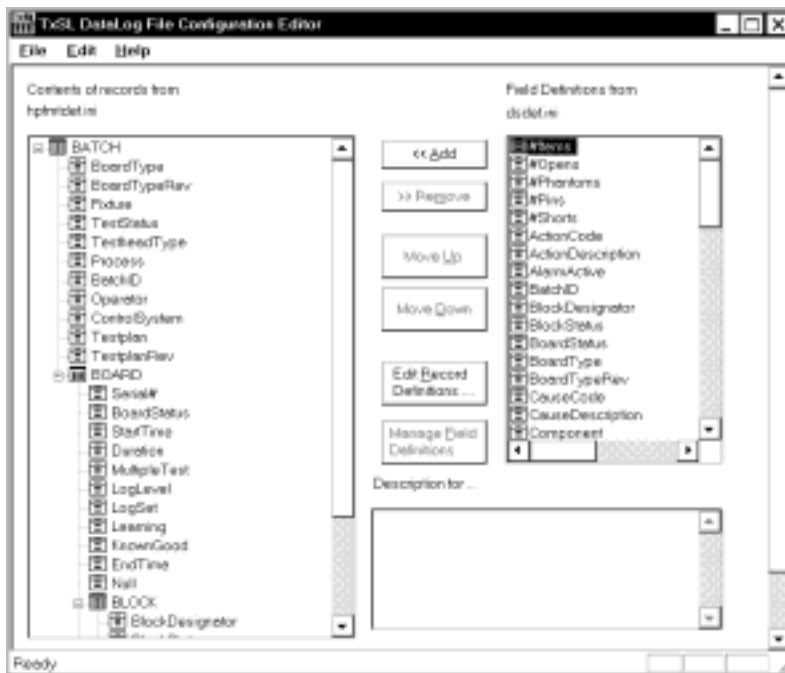
Customizing Datalogging

This chapter describes how to customize datalogging to control the data that appears in log records and how to access that data in custom applications. For general information about datalogging, see Chapter 5 in the *Using HP TestExec SL* book.

The Datalogging Configuration Editor

HP TestExec SL provides a Datalogging Configuration Editor, which is shown below, that you can use to:

- Examine or modify the default definitions of the records and fields plus the hierarchy of the fields (but not the number or hierarchy of records)
- Specify the behavior and format of datalogging files
- Associate fields in log records with symbols in symbol tables



You can run the Datalogging Configuration Editor from the Start menu in Windows. The editor's online help describes how to use it and contains detailed descriptions of the log records and the fields of data they contain.

Modifying the Records & Fields in Datalogging Files

Datalogging automatically collects data about tests when a testplan runs. The data collected includes:

- Information about a “batch” or group of UUTs
- Information about a specific UUT and the testplan used to test that UUT
- Information about an individual test done on a UUT
- Information about the operation of the limit checker used to evaluate the pass/fail results of a test done on a UUT
- Information gathered from the stream of report information generated by HP TestExec SL
- Information acquired when running a testplan in “learn” mode

Assuming that you are using the default log record format for HP TestExec SL, this information is formatted into records and fields suitable for use with a database.

For more information about the default definitions and how to modify them, see the online help for the Datalogging Configuration Editor.

Enhancing Datalogging

The default set of log records and descriptive fields provided with HP TestExec SL is intended to cover most of your testing needs. However, should you need to acquire additional information during datalogging, you can enhance the operation of the datalogging features. For example, you can log the values of symbols in symbol tables and log additional strings of report information.

Interacting with Symbol Tables

The following fields in the LogTestplan record reference predefined symbols in the System symbol table:

- FixtureID
- ModuleType
- OperatorName
- RunCount
- SerialNumber
- TestStationID
- TestStatus
- UnhandledErrorSource

As data is acquired via datalogging while testing, the values of these symbols become part of the stream of log data being generated. If you wish to examine their values, you can parse the datalogging files for them.

If the predefined symbols do not meet your needs, you can use the Datalogging Configuration Editor to define additional fields in log records and associate them with custom symbols in symbol tables of your choice. These symbols can be in the predefined symbol tables, such as SequenceLocals and TestStepLocals, *but not in external, user-defined symbol tables.*

If you create custom symbols and associate them with fields in log records, we recommend that you ensure that the scope of the fields and the symbol tables are similar. For example, fields in a LogTestplan record might reference symbols in the System or SequenceLocals symbol tables because the scope of those symbol tables is the entire testplan. However, fields in a

LogTestplan record should not reference symbols in the TestStepLocals and TestStepParms symbol tables because the scope of those symbol tables is an individual test.

For more information about the LogTestplan record and using the Datalogging Configuration Editor to define fields in log records and associate them with symbols in symbol tables, see the online help for the Datalogging Configuration Editor. For general information about symbol tables, see Chapter 5 of the *Using HP TestExec SL* book.

Knowing When a Datalogging File is Available

Suppose you use an external program to parse the datalogging files generated by HP TestExec SL. For example, your external program might manipulate log data and then insert it into a database for subsequent analysis. If you do this, it is important to know when HP TestExec SL has finished writing its datalogging file. Code that you write—in an operator interface, perhaps—can respond to a user-defined message and then parse the datalogging file only when the file is known to be valid.

If you are using the HP TestExec SL Control (described later) or equivalent functionality in the Runtime API (used to develop operator interfaces in Visual C++), it can generate a user-defined message to indicate that datalogging has finished. This message has the following characteristics:

- Its name is AfterDataLogFileWriteDone
- Its ID is 50002
- It returns a string that contains the name of the datalogging file
- It is sent when the DataLogEnabled property of the HP TestExec SL Control is set to “True” and the writing of datalogging files has not been disabled via the DataLogFileWriteEnabled message (described later).

Customizing Datalogging

Enhancing Datalogging

An example of using the `AfterDataLogFileWriteDone` message to check for a valid datalogging file looks like this:

```
'Code in operator interface written in Visual Basic
Private Sub TestExecSL1_UserDefinedMessage (ID As Integer, TextBlock _
    As String)
    Select Case ID 'Evaluate the identifier
        Case 5002 'It is the AfterDataLogFileWriteDone message
            ...code that processes the datalogging file
        Case Else 'The message is not of interest
            Exit Sub
    End Select
End Sub
```

For more information about user-defined messages and the HP TestExec SL Control, see Chapter 1 of the *Customizing HP TestExec SL* book.

Using the Results of Datalogging in Custom Applications

Disabling the Writing of Datalogging Files

If you are using the HP TestExec SL Control (described later) or similar functionality in the Runtime API (used to develop operator interfaces in Visual C++), it can send to HP TestExec SL a user-defined message that prevents datalogging files from being written although the processing of logged data continues. This message has the following characteristics:

- Its name is `DataLogFileWriteEnabled`
- Its ID is 50010
- It sends a string whose contents are “true” or “false”. If the message sent contains “true”, HP TestExec SL writes datalogging files to the hard disk; if “false”, the files will not be written but logged data will be processed if the `DataLogEnabled` property of the HP TestExec SL Control is set to “true”.
- It is sent when the `DataLogEnabled` property of the HP TestExec SL Control is set to “true” and datalogging is enabled for the testplan.

- You send it as needed, which typically is once after loading a testplan.

An example of sending a `DataLogFileWriteEnabled` message to HP TestExec SL via the HP TestExec SL Control looks like this:

```
'Code in operator interface written in Visual Basic  
'Disable the writing of datalogging files  
TestExecSL1.SendUserDefinedMessage(DataLogFileWriteEnabled, "false")
```

This is useful in custom applications where you do not need an actual datalogging file. For example, you could use a `DataLogFileWriteEnabled` message to disable the writing of datalogging files and instead use the `AfterDataLogCreateDone` message (described above) to return the results of datalogging for parsing by custom code in an operator interface.

For more information about user-defined messages and the HP TestExec SL Control, see Chapter 1 of the *Customizing HP TestExec SL* book.

Custom Parsing the Results of Datalogging

If desired, you can parse log data:

- Via a user-defined message as the testplan runs
- From a datalogging file after the testplan writes the file to disk

The main difference between these approaches lies in *when* you parse the results. The following topics describe how to do these tasks.

Parsing the Results as a Testplan Runs

If you are using the HP TestExec SL Control (described later) or similar functionality in the Runtime API, it can generate a user-defined message that contains the results of datalogging. This message has the following characteristics:

- Its name is `AfterDataLogCreateDone`
- Its ID is 50001

Customizing Datalogging

Enhancing Datalogging

- It returns a string that contains the results of datalogging¹
- It is sent when the DataLogEnabled property of the HP TestExec SL Control is set to “True” and datalogging is enabled for the testplan.

This message is useful in custom applications because you can write code—in an operator interface, perhaps—to parse the return string and then write the results to a database.

A simple example of intercepting the appropriate message from the HP TestExec SL Control and using that message to call the Datalogging Control and populate its objects so you can parse their contents looks like this:

```
Private Sub TestExecSL1_UserDefinedMessage(ByVal MessageID As Long, _
                                           ByVal Message As String)
    Select Case MessageID
        Case 5001 'ID of 5001 is the AfterDataLogCreateDone message
            'Populate the datalogging control's objects
            TxSLDataLog1.ParseDataLogMessage(Message)
            ...code that parses the contents of the control's objects
        Case Else
            'Do nothing
        End Select
    End Sub
```

For more information about user-defined messages and the HP TestExec SL Control, see Chapter 1 of the *Customizing HP TestExec SL* book.

Parsing the Results in a Datalogging File

If you are using the Datalogging Control (described later), HP TestExec SL's datalogging behavior is set to "TxSL" and its datalogging file format is set to "XML", you can readily parse data in a datalogging file. Accessing data in datalogging files is a two-step process in which you:

1. Call the Datalogging Control's ParseDataLogFile method to populate the control's internal objects with data from a datalogging file.

1. The string's contents are formatted for whichever datalogging format is specified in file "tstexsl.ini"; i.e., XML, spreadsheet, or HP 3070.

2. Use custom code that you write to access the Datalogging Control's objects to parse the data they contain.

An example is shown below.

```
Dim MyDataloggingFile As String
MyDataloggingFile = "\logdir\logfile.xml"
'Populate the datalogging control's objects
HPTxSLDataLog1.ParseDataLogFile(MyDataloggingFile)
...code that parses the contents of the control's objects
```

For more information about setting HP TestExec SL's datalogging format and behavior, see Chapter 5 of the *Using HP TestExec SL* book.

Changing the Name of the Datalogging File

The name of a datalogging file consists of a unique time stamp, a process ID, and an extension organized like this:

- 4 chars — year
- 2 chars — month
- 2 chars — day
- 2 chars — hour, based on 24-hour clock
- 2 chars — minutes
- 3 chars — milliseconds
- N* chars — process ID for HP TestExec SL
- “.xml” or “.log” — extension¹

An example of the name of a datalogging file might be:

19980708123306250224.xml

If you are using the HP TestExec SL Control (described later), you can programmatically change the name of the file to which datalogging

1. The extension is “.xml” for XML format or “.log” for HP 3070 format.

Customizing Datalogging

Enhancing Datalogging

information is written. This is done via user-defined messages whose characteristics are:

Message ID	Name
50011	UserDefinedFileName
50012	UserDefinedFileNamePrefix
50013	UserDefinedFileNameExtension

When HP TestExec SL receives one of these messages—sent by code in an operator interface, perhaps—it modifies the name of the datalogging file as requested in a string passed by the message.

When should you send this message? First, the message must be sent prior to the writing of the datalogging file to disk; i.e., you cannot use this message to change the name of an existing datalogging file. We recommend that you follow this sequence of events:

1. Determine the name of the datalogging file, perhaps by having code in your operator interface interrogate the UUT and use its serial number to create a unique name for the datalogging file.
2. Have code in your operator interface send one of the message described above that defines the name of the datalogging file.
3. Run the testplan.

Suppose for the sake of simplicity that the default name of the datalogging file that HP TestExec SL produces is “12345.log” and that you wish to change it from an operator interface written in Visual Basic that uses the HP TestExec SL Control. Executing this line of code:

```
TestExecSL1.SendUserDefinedMessage(UserDefinedFileNameSuffix, "xzy")
```

changes the name of the datalogging file to “12345xyz.log” by adding characters to the end of the file name.

Similarly, executing this line of code in Visual Basic:

```
TestExecSL1.SendUserDefinedMessage(UserDefinedFileNamePrefix, "abc")
```


adds characters to the beginning of the file name. Here, “12345.log” becomes “abc12345.log”.

If you prefer to specify the entire file name instead of prepending or appending to an existing one, executing this code:

```
TestExecSL1.SendUserDefinedMessage(UserDefinedFileName, "MyFile")
```

creates a datalogging file named “MyFile.log”.

Note

If you specify a null value for the string passed in `UserDefinedFileName`, the default file name will be used.

How do you decide which kind of message to use? See the table below.

If you need...

Then use...

unique names for datalogging files

`UserDefinedFileNamePrefix` or `UserDefinedFileNameSuffix` because they automatically maintain the uniqueness of file names by combining a unique name generated by HP TestExec SL with a prefix or suffix specified by you

non-unique names for datalogging files

`UserDefinedFileName` because it repeatedly writes a file with the specified name unless your code explicitly changes the file's name. If you use this message, be sure to write to an empty directory or ensure that your code maintains the uniqueness of datalogging file names. *If a datalogging file of the same name already exists, an error will occur and the new file will not be written.*

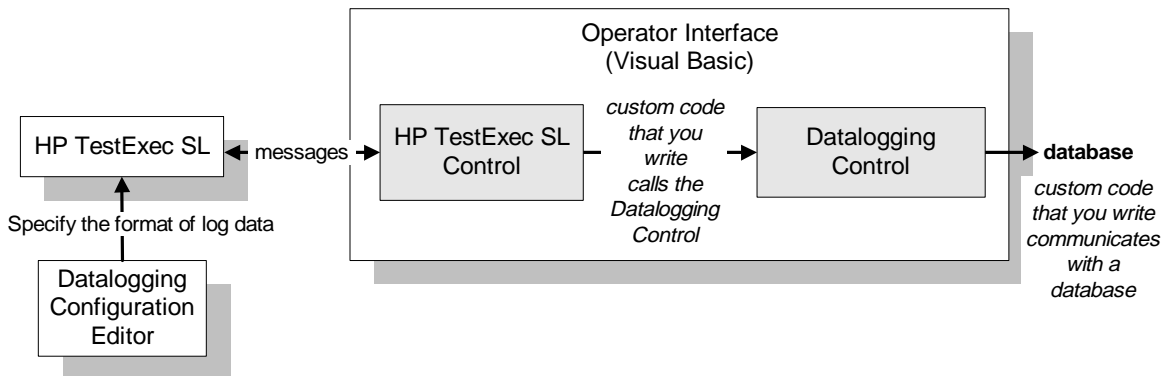
For more information about user-defined messages and the HP TestExec SL Control, see Chapter 1 of the *Customizing HP TestExec SL* book.

Using the HP TestExec SL Datalogging Control

What is the HP TestExec SL Datalogging Control?

Besides the ActiveX control used to create operator interfaces written in Visual Basic (described in Chapter 1 of the *Customizing HP TestExec SL* book), HP TestExec SL provides an ActiveX control that supports custom datalogging applications.

As shown below, various messages are passed back and forth between HP TestExec SL and the HP TestExec SL Control. Custom code that you write responds to a message received by the HP TestExec SL Control and calls the Datalogging Control as needed. This populates objects internal to the Datalogging Control with data acquired via datalogging. Accessing these objects lets you further manipulate the data, which you can send to databases or other applications.

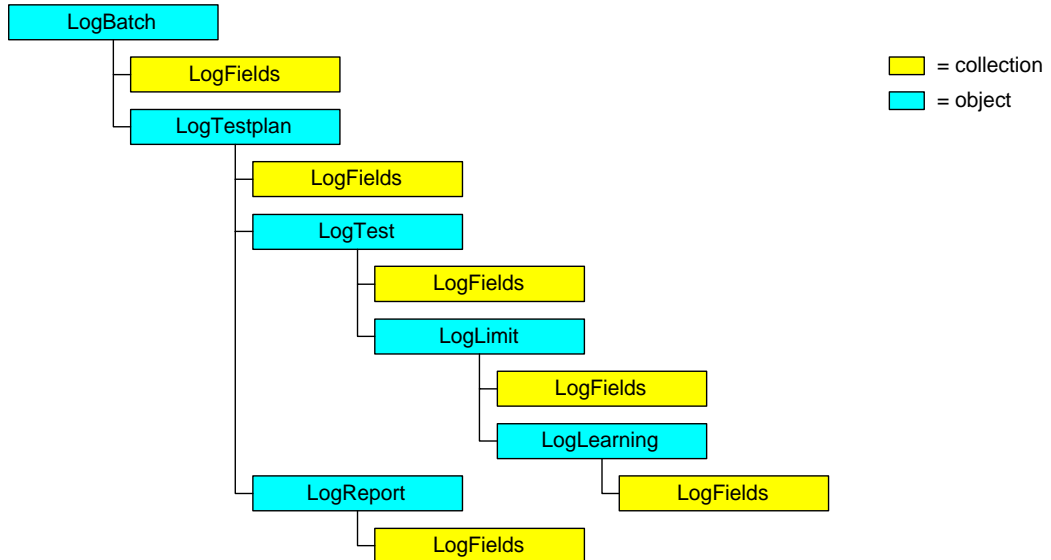


Note that the Datalogging Control has no effect on which kind of data is collected during datalogging, or the format of the data. These are determined by settings specified using the Datalogging Configuration Editor.

What's Inside the Datalogging Control?

The Datalogging Control contains collections of objects organized in the hierarchy of log records and fields used for datalogging. Seen at an overview

level, the hierarchy of the internal objects and collections of objects looks like that shown below.



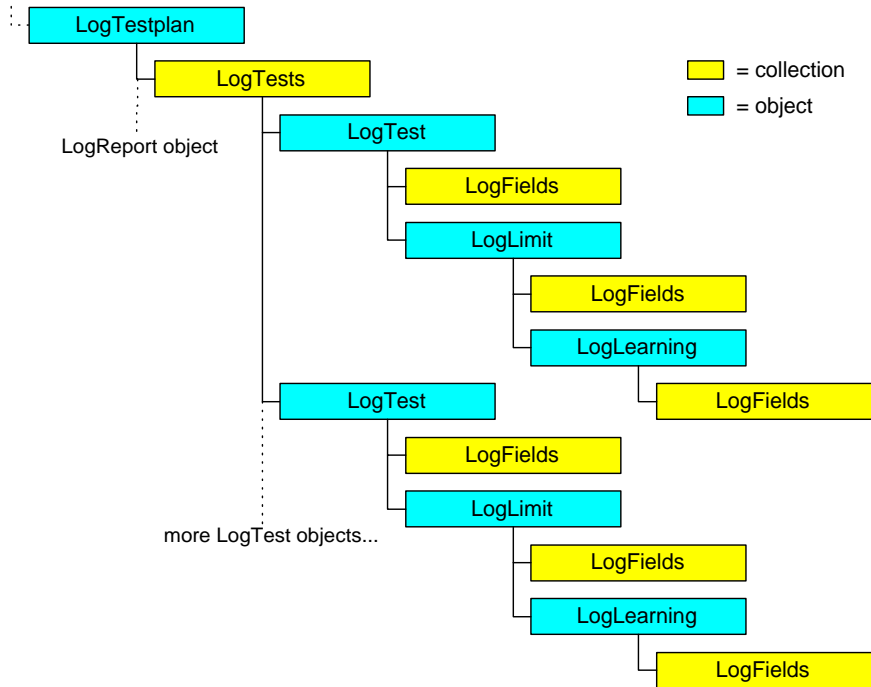
The overall scheme is that each object can contain a collection of fields and other objects that can also contain collections of fields. For example, the LogTestplan object contains:

- A LogFields collection that contains data items that describe the characteristics of a testplan
- A LogTest object that contains a LogFields collection and a LogLimit object that describe the characteristics of a test
- A LogReport object that contains a LogFields collection that contains any report data produced during datalogging

The model shown above is overly simplistic insofar as actual datalogging is complex enough to require collections of objects, as shown below. Here, the

Customizing Datalogging Using the HP TestExec SL Datalogging Control

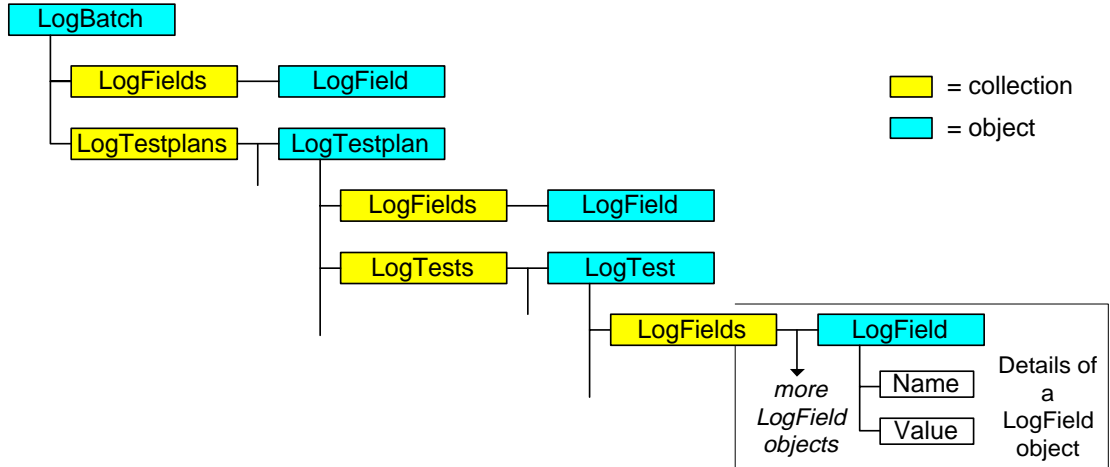
testplan contains more than one test. The LogTest objects that describe the characteristics of individual tests reside in a collection name LogTests.¹



As shown below, the actual data acquired via datalogging resides in pairs of names and values associated with LogField objects. For example, a LogTest object might contain a LogFields collection that contains a LogField object

1. Notice the naming convention used here. The name of the collection is plural—LogTests—and the name of each object in the collection—LogTest—is singular.

in which resides the name TestJudgment, whose value indicates whether the test passed or failed.



Accessing Collections & Objects in the Datalogging Control

An example of accessing a collection in the Datalogging Control and printing its contents looks like this:

```
'Print the names/values of LogField objects in the LogFields collection  
' in a LogTestplan object in the LogTestplans collection  
Dim I As Integer  
With TxSLDatalog1.LogBatch.LogTestplans(1).LogFields  
  For I = 1 to .count  
    Debug.Print "Name: " & .Item(I).Name & ", Value: " & .Item(I).Value  
  Next I  
End With
```

Customizing Datalogging

Using the HP TestExec SL Datalogging Control

The example accesses by index—i.e., `LogTestplans(1)`—the first `LogTestplan` object in the `LogTestplans` collection. With the exception of the `LogFields` collection, whose objects are accessible by name or by index, you must access objects in collections in the Datalogging Control by their indexes.¹

The `LogFields` collection gives you the option of accessing its `LogField` objects by name. This is useful when you wish to access the value of a specific data item.² An example of accessing by name a specific `LogField` object in the `LogFields` collection is shown below.

```
'Example accesses by name the value contained in the LogField object  
' named "TestName"  
Dim MyTestName As String  
MyTestName = TxSLDataLog1.LogTestplans(1).LogTests(1). _  
                    LogFields("TestName").Value
```

You can find a more detailed description of these objects and collections in the online help for the Datalogging Control. The hierarchy of objects in the Datalogging Control corresponds to the hierarchy of records and fields for logged data described in the online help for the Datalogging Configuration Editor. For example, the `LogTestplan` object in the Datalogging Control corresponds to the `LogTestplan` record in the datalogging schema, and the `LogField` object corresponds to individual fields.

Note

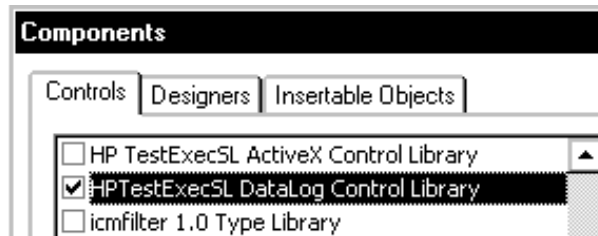
The properties of the HP TestExec SL Control's internal objects do not appear in Visual Basic's Properties window. You must browse the control's online help or use Visual Basic's Object Browser to find descriptions of its internal objects and their properties and methods.

1. Objects are added to collections in the order in which they arrive; i.e., (1) before (2). This can be useful if you wish to parse them later in the same sequence as when they arrived.
2. The names of `LogField` objects correspond to the names of fields in the hierarchy of logged data.

Adding the Datalogging Control to a Project

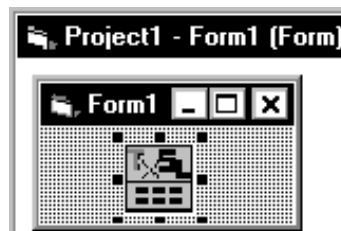
Assuming that HP TestExec SL is installed on your system, you can do the following to add the Datalogging Control to your project:

1. Open an existing or a new project in Visual Basic
2. Choose Project | Components in Visual Basic's menu bar.
3. When the Components box appears, be sure its Controls tab is selected.
4. Choose the Browse button and locate the Datalogging Control, which is in file "txslatalog.ocx" in directory "<HP TestExec SL home>\bin". When this file is selected, the HP TestExec SL control will appear in the list of controls, as shown below.



5. Choose the OK button.

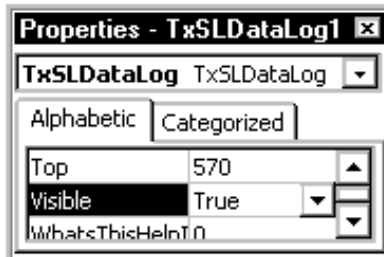
Once the Datalogging Control appears in Visual Basic's Toolbox, you can use the mouse to place it on a form as you would any other control. When copied onto a form, the control looks like this:



Customizing Datalogging Using the HP TestExec SL Datalogging Control

Note

As shown below, you probably will want to set the Datalogging Control's `Visible` property to `False` to keep the control from appearing at runtime.



Getting Online Help for the Datalogging Control

You can invoke online help for the Datalogging Control by selecting the control when it appears on a form and then pressing softkey F1.

Index

A

- action
 - for use with operator interfaces, 56
 - used to prompt system operators, 56
- AfterDataLogCreateDone message, 107
- AfterDataLogFileWriteDone message, 105
- automation interface, 12
 - creating in Visual C++, 77
 - typical scenario, 12
 - typical tasks for, 13
- automation interface created in Visual C++
 - dealing with problems that arise during testing, 82
 - displaying messages to user interface, 80
 - generating repair information, 81
 - LAN communications, 82
 - monitoring test results, 79
 - responding to keyboard & mouse commands, 80
 - signaling downstream devices, 82
 - using a bar code reader with, 79
 - using datalogging with, 82
 - writing repair tickets, 81

B

- bar code
 - changing the processing of, 63
 - parsing into a UUT type & serial number, 63
- bar code reader
 - changing how bar codes are processed, 63
 - overview of using with operator interfaces, 62
 - parsing bar codes in operator interfaces created in Visual Basic, 47
 - sample bar codes used to test bar code readers, 65
 - testing, 65
 - typical characteristics of, 63

- using to automatically load testplans in an operator interface, 68
- using with an automation interface created in Visual C++, 79

C

- concurrent testing, 41
- control
 - HP TestExec SL Control, 17
 - HP TestExec SL Datalogging Control, 112
- customization
 - customizing datalogging, 101
 - customizing operator interfaces created in Visual Basic, 15
 - customizing operator interfaces created in Visual C++, 68

D

- DataLogFileWriteEnabled message, 106
- datalogging
 - changing the name of the datalogging file, 109
 - custom parsing the results of, 107
 - disabling the writing of datalogging files, 106
 - enhancing, 104
 - interacting with symbol tables, 104
 - knowing when a datalogging file is available, 105
 - name of the datalogging file, 109
 - parsing the results as a testplan runs, 107
 - parsing the results in a datalogging file, 108
 - using the HP TestExec SL Datalogging Control, 112
- Datalogging Configuration Editor, 102
- datalogging control. See "HP TestExec SL Datalogging Control"
- disabling the writing of datalogging files, 106

E

- enhancing datalogging, [104](#)
- event
 - associated with testplans, [25](#)
 - associated with tests, [28](#)
 - in HP TestExec SL Control, [25](#)

H

- hardware handler
 - creating, [86](#)
 - interacting with an operator interface created in Visual Basic, [40](#)
 - using to monitor the status of hardware, [87](#)
- help
 - online help for the HP TestExec SL Control, [21](#)
 - online help for the HP TestExec SL Datalogging Control, [118](#)
- HP TestExec SL Control, [17](#)
 - adding to a Visual Basic project, [19](#)
 - events, [25](#)
 - getting online help for, [21](#)
 - methods, [23](#)
 - states, [23](#)
- HP TestExec SL Datalogging Control, [112](#)
 - accessing internal objects & collections in, [115](#)
 - adding to a Visual Basic project, [117](#)
 - getting online help for, [118](#)
 - overview of internal objects & collections, [113](#)

L

- language
 - adding language support for a new control to an operator interface created in Visual Basic, [53](#)
 - adding language support for a new message to an operator interface created in Visual Basic, [54](#)

- adding support for a new language to an operator interface created in Visual Basic, [50](#)
- changing for operator interface created in Visual Basic, [47](#)

M

- method
 - in HP TestExec SL Control, [23](#)

O

- operator interface, [2](#)
 - actions provided for use with, [56](#)
 - appearance of, [3](#)
 - associating testplans & UUTs with, [68](#)
 - best way to create, [3](#)
 - creating in Visual Basic, [15](#)
 - creating in Visual C++, [73](#)
 - designing for usability, [3](#)
 - reasons for customizing, [2](#)
 - sample actions provided for testing & debugging, [14](#)
 - testing & debugging, [14](#)
 - using breakpoints in Visual Basic when debugging, [43](#)
 - which programming languages are supported for, [3](#)
- operator interface created in Visual Basic, [15](#)
 - See also "HP TestExec SL control"
 - accessing a symbol table from, [65](#)
 - accessing hardware resources from, [39](#)
 - adding information to reports, [45](#)
 - adding language support for a new control, [53](#)
 - adding language support for a new message, [54](#)
 - adding support for a new language, [50](#)
 - changing the information that appears in reports, [46](#)
 - changing the language, [47](#)
 - concurrent testing, [41](#)
 - configuring, [43](#)

- controlling what appears in reports, 44
- example of minimum code needed to implement, 21
- finding items in code, 21
- hiding existing functionality of, 43
- how it interacts with HP TestExec SL, 17
- interacting with hardware handlers, 40
- parsing bar codes, 47
- samples provided by HP, 16
- skills needed to customize, 17
- user-defined message, 32
- user-defined query, 37
- user-defined response, 37
- operator interface created in Visual C++, 68
 - accessing global data from, 72
 - beginning a test cycle, 75
 - beginning when testplan name is unknown, 76
 - displaying messages, 76
 - displaying name of current test, 76
 - displaying testplan & test timing, 76
 - how it requests service, 70
 - interacting with test sequencer, 72
 - overview of internal operation, 69
 - responding to a Run button, 74

P

- parallel testing, 41

S

- state
 - for operator interface created in Visual Basic, 23
 - for operator interface created in Visual C++, 70
- symbol table
 - accessing from an operator interface created in Visual Basic, 65
 - interacting with datalogging, 104

T

- test
 - events associated with, 28
- testing more than one UUT at a time, 41
- test-level event, 28
 - enabling & disabling, 29
- testplan
 - associating with an operator interface, 68
 - events associated with, 25
 - prompting system operators from, 61
 - sample testplans used to test bar code readers, 65
- testplan-level event, 25

U

- user-defined message, 32
 - reserved by HP, 39
- user-defined query, 37
- user-defined response, 37
- UserDefinedFileName message, 110
- UserDefinedFileNameExtension message, 110
- UserDefinedFileNamePrefix message, 110
- UUT
 - associating with an operator interface, 68