MERCURY QUICKTEST PROFESSIONAL™

.NET Add-in

VERSION 8.2

Extensibility Guide

MERCURY™

Mercury QuickTest Professional .NET Add-in

Extensibility Guide

Version 8.2

MERCURY™

QuickTest Professional .NET Add-in Extensibility Guide, Version 8.2

Mercury Interactive Corporation
379 North Whisman Road
Mountain View, CA 94043
Tel: (650) 603-5200
Toll Free: (800) TEST-911
Customer Support: (877) TEST-HLP
Fax: (650) 603-5300

If you have any comments or suggestions regarding this document, please send them via e-mail to documentation@mercury.com.

QTPNETEXTGD8.2/01

Table of Contents

Welcome

Welcome to QuickTest Professional .NET Add-in Extensibility.

QuickTest Professional .NET Add-in Extensibility enables you to support testing applications using third-party and custom .NET controls that are not supported out-of-the-box by the QuickTest Professional .NET Add-in.

Who Should Read this Guide

This guide is intended for programmers, systems analysts, system designers, and technical managers.

To use this guide, you should be familiar with:

- QuickTest Professional Object Model
- QuickTest Professional .NET Add-in
- XML (basic knowledge)
- .NET Programming in C#

Using This Guide

This guide explains everything you need to know to use QuickTest Professional .NET Add-in Extensibility to extend Test Run and Test Record support for third-party and custom .NET controls. Test Record is the software module used in the session in which the actions performed on the application being tested and the application's resulting behaviors are recorded and the recording is converted to a test script. Test Run is the software module used to run this script and track the results to test if the application is performing as required.

This guide should be used together with the *QuickTest Professional .NET Add-in Extensibility API Reference* (provided in online help format). These two documents should be used in conjunction with the *QuickTest Professional User's Guide,* the *QuickTest Professional .NET Add-in Guide,* and the *QuickTest Professional Object Model Reference*. All of these guides can be accessed online by selecting **Help** > **QuickTest Professional Help** from the QuickTest main screen. The guides are also available as printed books.

This book contains the following chapters:

Chapter 1 Introducing QuickTest Professional .NET Add-in Extensibility

Explains the concepts of extending support to custom .NET controls.

Chapter 2 Installing the Custom Server C# Project Template

Explains how to install the .NET Add-in Extensibility module and how to configure your QuickTest Professional .NET Add-in project to use Extensibility.

Chapter 3 Using a .NET DLL to Extend Support for a Custom Control

Explains how to extend support for a custom control using a .NET DLL.

Chapter 4 Using an XML File to Extend Support for a Custom Control

Explains how to extend support for a custom control using an XML file.

Chapter 5 Configuring QuickTest to Use the Custom Server

Explains how to configure QuickTest to use the Custom Server and describes the configuration file format.

Chapter 6 Step-by-Step Tutorial

Provides instructions and leads you step-by-step through the process of creating custom support for a control.

Typographical Conventions

This book uses the following typographical conventions:

1, 2, 3	Bold numbers indicate steps in a procedure.
>	The greater-than sign separates menu levels (for example, **File** > **Open**).
Stone Sans	The **Stone Sans** font indicates names of interface elements (for example, the **Run** button) and other items that require emphasis.
Bold	**Bold** text indicates method or function names.
Italics	*Italic* text indicates method or function arguments and book titles. It is also used when introducing a new term.
<>	Angle brackets enclose a part of a file path or URL address that may vary from user to user (for example, <**MyProduct installation path**>\bin).
Arial	The Arial font is used for examples and text that is to be typed literally.
Arial bold	The **Arial bold** font is used in syntax descriptions for text that should be typed literally.
SMALL CAPS	The SMALL CAPS font indicates keyboard keys.
…	In a line of syntax, an ellipsis indicates that more items of the same format may be included. In a programming example, an ellipsis is used to indicate lines of a program that were intentionally omitted.
[ ]	Square brackets enclose optional arguments.
|	A vertical bar indicates that one of the options separated by the bar should be selected.

Documentation Updates

Mercury Interactive is continuously updating its product documentation with new information. You can download the latest version of this document from the Customer Support Web site (http://support.mercury.com).

To download updated documentation:

1. In the Customer Support Web site, click the **Documentation** link.
2. Under **Select Product Name**, select **QuickTest Professional**.

   Note that if **QuickTest Professional** does not appear in the list, you must add it to your customer profile. Click **My Account** to update your profile.
3. Click **Retrieve**. The Documentation page opens and lists the documentation available for the current release and for previous releases. If a document was recently updated, **Updated** appears next to the document name.
4. Click a document link to download the documentation.

1

Introducing QuickTest Professional .NET Add-in Extensibility

Welcome to QuickTest Professional .NET Add-in Extensibility.

QuickTest Professional .NET Add-in Extensibility enables you to provide high-level support for third-party and custom .NET controls that are not supported out-of-the-box by the QuickTest Professional .NET Add-in.

It is possible to record tests on .NET controls that are not supported out-of-the-box by the QuickTest Professional .NET Add-in without using the Extensibility module. However, the recorded script will reflect the low-level activities passed as Windows messages. By supporting a .NET control with the Extensibility module, this default low-level support is extended so that scripts are meaningful, understandable, and easy to modify.

This chapter describes:

- Understanding .NET Add-in Extensibility
- Understanding Coding Options—.NET DLL and XML
- Understanding Custom Server Run-Time Contexts
- Understanding Test Object Mapping

Understanding .NET Add-in Extensibility

QuickTest Professional .NET Add-in Extensibility enables you to support third-party and custom .NET controls by extending QuickTest test objects with methods representing the meaningful behaviors of those .NET controls.

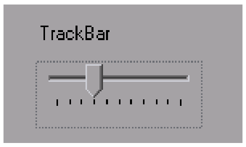
The QuickTest Professional .NET Add-in, without the Extensibility module, supports many .NET controls out-of-the-box. The .NET Add-in provides test objects that supply methods representing these controls' meaningful behaviors.

The Extensibility module enables you to implement this level of support for additional .NET controls. Using the Extensibility module, you extend the .NET Add-in interfaces by overriding existing methods and defining new ones, creating a Custom Server. When the custom control is mapped to an existing QuickTest test object, you have the full functionality of a QuickTest test object, including visibility in IntelliSense, and meaningful steps in the business component or test script.

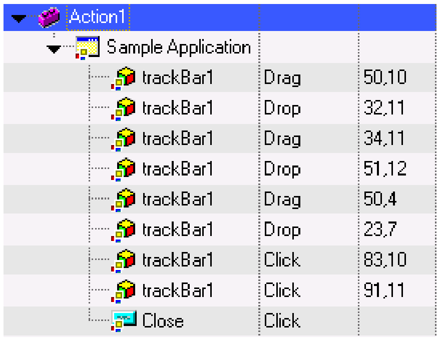
Understanding the Concept of Meaningful Behaviors

A control's meaningful behavior is the behavior that you want to test. For example, when you click on a button in a radio button group in your application, you are interested in the value of the selection, not in the **Click** event and the coordinates of the click. The meaningful behavior of the radio button group is the change in the selection.

If you record a test or business component on a custom control without extending support for the control, you record the low-level behaviors of the control. For example, the TrackBar control in the sample .NET application shown below is a control that does not have a corresponding QuickTest test object.



If you record on the TrackBar without implementing support for the control, the Keyword View looks like this:



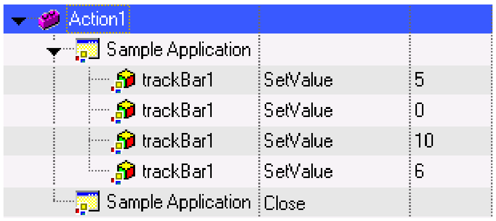
Action1		
Sample Application		
trackBar1	Drag	50,10
trackBar1	Drop	32,11
trackBar1	Drag	34,11
trackBar1	Drop	51,12
trackBar1	Drag	50,4
trackBar1	Drop	23,7
trackBar1	Click	83,10
trackBar1	Click	91,11
Close	Click	

In the Expert View, the recorded test looks like this:

```
SwfWindow("Sample Application").SwfObject("trackBar1").Drag 50,10
SwfWindow("Sample Application").SwfObject("trackBar1").Drop 32,11
SwfWindow("Sample Application").SwfObject("trackBar1").Drag 34,11
SwfWindow("Sample Application").SwfObject("trackBar1").Drop 51,12
SwfWindow("Sample Application").SwfObject("trackBar1").Drag 50,4
SwfWindow("Sample Application").SwfObject("trackBar1").Drop 23,7
SwfWindow("Sample Application").SwfObject("trackBar1").Click 83,10
SwfWindow("Sample Application").SwfObject("trackBar1").Click 91,11
SwfWindow("Sample Application").SwfButton("Close").Click
```

Note that the methods recorded are Drag, Drop and Click at specific coordinates in the control display—the low-level actions of the TrackBar control. These steps are difficult to understand and modify.

If you use .NET Add-in Extensibility to support the TrackBar control, the result is more meaningful. Below is the Keyword View of a test recorded on the TrackBar with a Custom Server:



Action1		
Sample Application		
trackBar1	SetValue	5
trackBar1	SetValue	0
trackBar1	SetValue	10
trackBar1	SetValue	6
Sample Application	Close	

In the Expert View, the recorded test looks like this:

```
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 5
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 0
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 10
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 6
SwfWindow("Sample Application").Close
```

QuickTest is now recording a SetValue operation reflecting the new slider position, instead of the low-level Drag, Drop, and Click operations recorded without the custom test object. You can understand and modify this test script more easily.

Understanding Coding Options—.NET DLL and XML

You can implement QuickTest custom support in two ways:

- **.NET DLL**—Extends support for the control using a .NET Assembly.
- **XML**—Extends support for the control using an XML file.

Guidelines for Selecting a Coding Option

Most Custom Servers are implemented as a .NET DLL. This option is generally preferred because development is supported by all the services of the program development environment, such as syntax checking, debugging, and Microsoft IntelliSense. Furthermore, a Custom Server implemented as a .NET DLL can perform part of its Test Record functions in the **QuickTest** context and part in the **Application under test** context. For more information, see “Using a .NET DLL to Extend Support for a Custom Control” on page 13, and the *QuickTest Professional .NET Add-in Extensibility API Reference*.

For information about run-time contexts, see “Understanding Custom Server Run-Time Contexts” on page 5.

The XML implementation is most practical either with relatively simple, well documented controls, or with controls that map well to an existing object but for which you need to replace the Test Record implementation, or replace or add a small number of test object Test Run methods. It is also useful when a full programming environment is not available, since it requires only a text editor.

However, when implementing a custom control with XML, you have none of the support provided by a program development environment. The XML implementation runs only in the **Application under test** context. For more information, see "Using an XML File to Extend Support for a Custom Control" on page 29.

For information about setting the coding option, see "Configuring QuickTest to Use the Custom Server" on page 35.

Understanding Custom Server Run-Time Contexts

Classes supplied by a Custom Server may be instantiated in one of two software processes, or *run-time contexts*:

- **Application under test**
- **QuickTest**

An object created in the **Application under test** context has direct access to the .NET control's events, methods, and properties. However, it cannot listen to Windows messages.

An object created in the **QuickTest** context can listen to Windows messages. However, it does not have direct access to the .NET control's events, methods, and properties.

If the Custom Server is implemented as a .NET DLL, an object created in the **QuickTest** context can create Assistant objects that run in the **Application under test** context.

Guidelines for Selecting the Custom Server Run-Time Context

The Custom Server may implement Test Record, Test Run, or both. Test Record is the software module used in the session in which the actions performed on the application being tested and the application's resulting behaviors are recorded and the recording is converted to a test script. Test Run is the software module used to run this script and track the results to test if the application is performing as required.

Test Run is nearly always implemented in the **Application under test** context. Direct access to the control makes setting values and calling the control's methods straightforward. There is no need to listen to Windows messages during a Test Run session, so the **QuickTest** context is not required. However, if your application makes heavier use of QuickTest services than services of the custom control, it may be more efficient to implement Test Run in the **QuickTest** context.

The programming for Test Record is generally simpler in the **Application under test** context. However, if it is essential to use Windows messages for recording, you must use the **QuickTest** context.

If the .NET DLL Custom Server must both listen to Windows messages and access control events and properties, use Assistant classes. The Custom Server running in the **QuickTest** context can listen to events in the **Application under test** context with Assistant class objects that run in the **Application under test** context. These objects also provide direct access to control properties.

For more information, see "Implementing Test Record for a Custom Control Using the .NET DLL" on page 19.

For more information about Assistant classes, see "Using a .NET DLL to Extend Support for a Custom Control" on page 13, and refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

For more information about setting the context, see "Configuring QuickTest to Use the Custom Server" on page 35.

Understanding Test Object Mapping

All Custom Servers are mapped to a parent QuickTest test object. When the test object is applied to the custom control, the Custom Server extends the parent test object.

When you map your Custom Server to a functionally similar QuickTest test object, you do not have to override those Test Run methods of the parent object which apply without change to your custom object. For example, most controls have a Click method. If the Click method of the parent object implements the Click method of the custom object adequately, you do not need to override the parent's method.

To cover the Test Run functionality of the custom object that does not exist in the parent, add new methods in your Custom Server. To cover functionality that has the same method name, but a different implementation, override the parent methods. The test object type that supports the custom control is the new type that consists of the Test Run members of the parent object or overrides of those members, and new members added by this Custom Server.

Note that mapping is sometimes sufficient without any programming. If the parent QuickTest test object adequately covers a control, it is sufficient to map the control to the QuickTest test object. If the QuickTest test object adequately covers Test Record, but you need to customize Test Run, do not implement Test Record.

If you do implement Test Record, the implementation replaces that of the parent object. You must implement all required Test Record functionality.

If you do not specify a mapping, QuickTest maps the custom control to the default generic test object, **SwfObject**.

When you edit a script line that references the custom test object, Microsoft IntelliSense displays the properties and methods of the custom test object in addition to those of the parent QuickTest test object.

For more information about mapping, see "Configuring QuickTest to Use the Custom Server" on page 35.

2

Installing the Custom Server C# Project Template

This chapter explains how to install the Custom Server C# Project Template for Microsoft Visual Studio .NET. Refer to the *QuickTest Professional .NET Add-in Readme* file in your installation for an updated list of supported versions of Visual Studio.

This installation provides a Custom Server project template and the wizard that runs when the template is selected to create a new project.

The Custom Server template provides a framework of blank code, some sample code, and the QuickTest project references required to build a Custom Server.

The wizard simplifies setting up a Visual Studio .NET project to create a Custom Server .NET DLL using the .NET Add-in Extensibility module. For more information, see "Using a .NET DLL to Extend Support for a Custom Control" on page 13.

This chapter describes:

- ➤ Before You Install
- ➤ Running the Installation Program
- ➤ Uninstalling the Project Template

Before You Install

Before you install the Custom Server C# Project Template, review the requirements listed below.

- You must have access to the **InstWizard.msi** file. You can access the **InstWizard.msi** file from either a computer on which the QuickTest Professional .NET Add-in is installed, or from the root folder of the QuickTest Professional .NET Add-in CD-ROM.
- Microsoft Visual Studio .NET must be installed on your computer.

Running the Installation Program

The **InstWizard.msi** file is found in the QuickTest Professional .NET Add-in installation, and on the QuickTest Professional .NET Add-in CD-ROM.

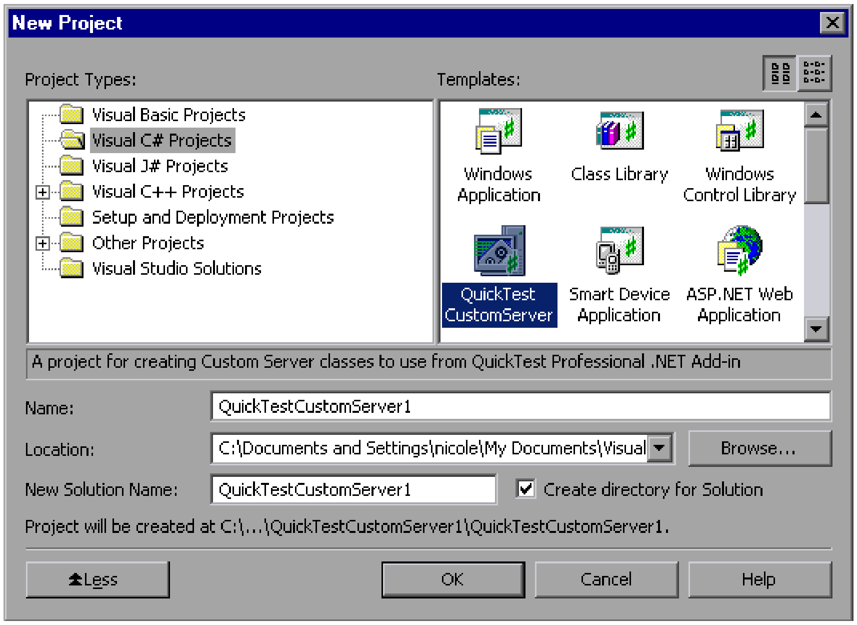
To install the .NET Add-in Custom Server C# Project Template:

1. Close all instances of Microsoft Visual Studio .NET.
2. Locate the **InstWizard.msi** file. You can find it in one of the following locations:
   - In the <**QuickTest Professional installation path**>\bin\Custom folder on a computer on which the QuickTest Professional .NET Add-in is installed.
   - In the root folder of the QuickTest Professional .NET Add-in CD-ROM.
3. Run the installation by double-clicking on the **InstWizard.msi** file. The Custom Server C# Project Template is installed on your computer.

To confirm that the installation was successful:

1. Open Microsoft Visual Studio .NET.
2. Choose **File** > **New** > **Project** to open the New Project dialog box.
3. Select **Visual C# Projects** in the **Project Types** list.

4 Confirm that the **QuickTest CustomServer** template icon appears in the **Templates** pane.



A project for creating Custom Server classes to use from QuickTest Professional .NET Add-in

Name: QuickTestCustomServer1

Location: C:\Documents and Settings\nicole\My Documents\Visual

New Solution Name: QuickTestCustomServer1

[✓] Create directory for Solution

Project will be created at C:\...\QuickTestCustomServer1\QuickTestCustomServer1.

Uninstalling the Project Template

You can uninstall the Custom Server C# Project Template from the Windows Control Panel.

To uninstall the project template:

1 Select **Start** > **Settings** > **Control Panel** > **Add/Remove Programs**. The Add/Remove Programs dialog box opens.

2 In the **Add/Remove Programs** list, select **Mercury CustomWizard**.

3 Click **Remove**.

3

Using a .NET DLL to Extend Support for a Custom Control

You can support a .NET control by creating a Custom Server implemented as a .NET DLL.

To create a .NET DLL Custom Server you need to know how to program a .NET Assembly. The illustrations and instructions in this chapter assume that you are using Microsoft Visual Studio .NET as your development environment and that you have installed the Custom Server C# Project Template. For more information, see “Installing the Custom Server C# Project Template” on page 9.

This chapter describes:

About Using a .NET DLL to Extend Support for a Custom Control

You can create a Custom Server to implement high level support for a custom .NET control. The Custom Server is a .NET DLL class library that implements interfaces for Test Record and/or Test Run, and general utilities. For more information, see “Implementing Test Record for a Custom Control Using the .NET DLL” on page 19, “Implementing Test Run for a Custom Control Using the .NET DLL” on page 23, and “API Overview” on page 26.

After creating the Custom Server, configure QuickTest to use it. For more information, see “Configuring QuickTest to Use the Custom Server” on page 35.

Creating a Custom Server

To create a Custom Server, set up a .NET project in Microsoft Visual Studio .NET, code the support for QuickTest Test Record and/or Test Run, and edit the configuration file so that QuickTest loads the Custom Server.

Setting up the .NET Project

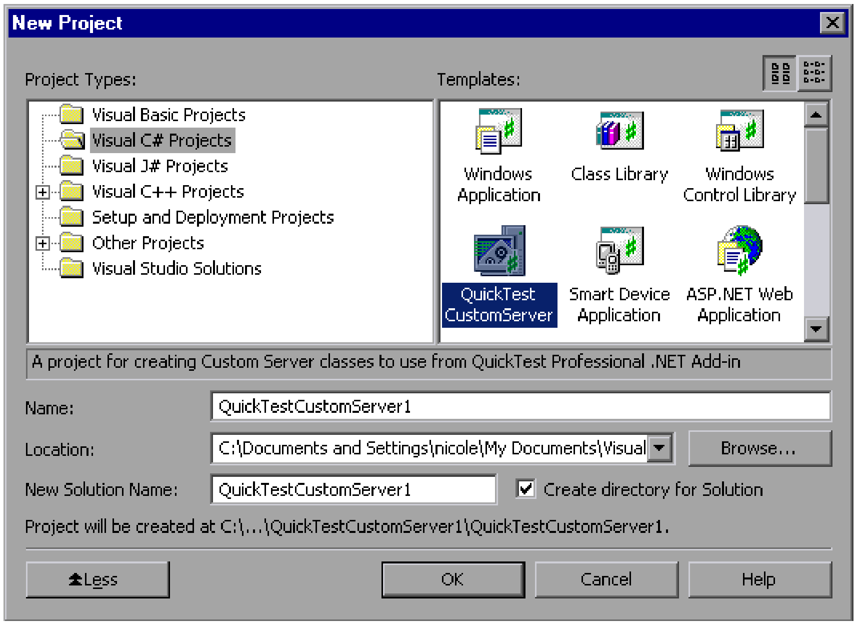
Set up a .NET project in Microsoft Visual Studio .NET using the Custom Server C# Project Template.

When you set up the .NET project, the template does the following:

- Creates an XML file with definitions of the Custom Server that you can copy into the QuickTest configuration file.
- Creates the project files necessary for the build of the .DLL file.
- Sets up a C# file with commented code that contains the definitions of methods that you can override when you implement Test Record or Test Run.
- Provides sample code that demonstrates some Test Record and Test Run implementation techniques.

To setup a new .NET project:

1. Start Microsoft Visual Studio .NET.
2. Choose **File** > **New** > **Project** to open the New Project dialog box, or press CTRL + SHIFT + N. The New Project dialog box opens.
3. Select **Visual C# Projects** in the **Project Types** list.



A project for creating Custom Server classes to use from QuickTest Professional .NET Add-in

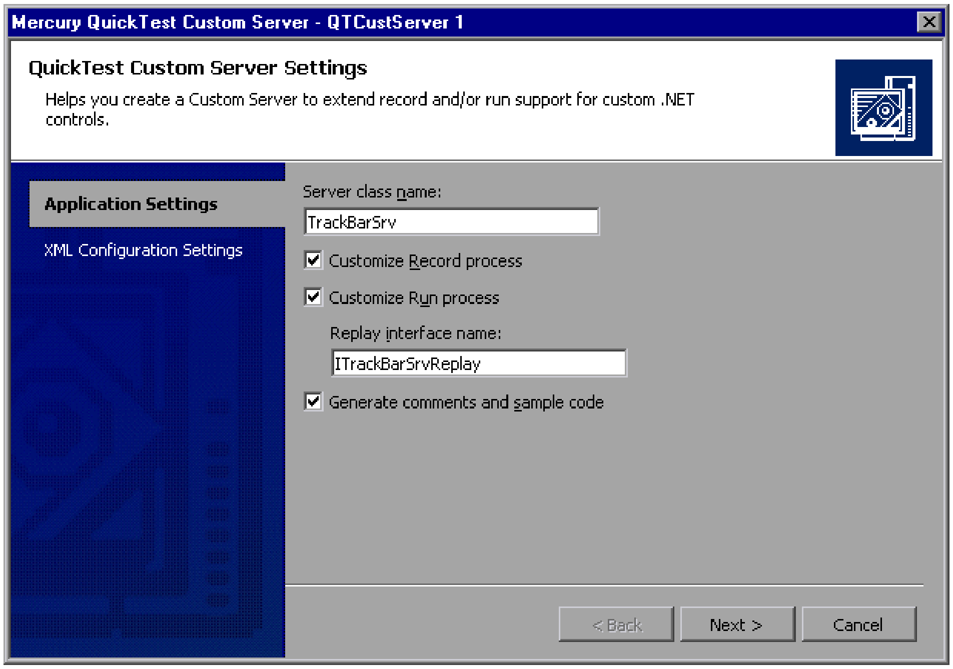
Name: QuickTestCustomServer1

Location: C:\Documents and Settings\nicole\My Documents\Visual

New Solution Name: QuickTestCustomServer1 ✓ Create directory for Solution

Project will be created at C:\...\QuickTestCustomServer1\QuickTestCustomServer1.

4 Select the **QuickTest CustomServer** template in the Templates pane. Enter the name of your new project and the location in which you want to save the project. Click **OK**. The QuickTest Custom Server Settings wizard opens.


Mercury QuickTest Custom Server - QTCustServer 1

QuickTest Custom Server Settings

Helps you create a Custom Server to extend record and/or run support for custom .NET controls.

Server class name: TrackBarSrv

☑ Customize Record process

☑ Customize Run process

Replay interface name: ITrackBarSrvReplay

☑ Generate comments and sample code

5 Make your selections in the Application Settings page of the wizard.

➤ In the **Server class name** box, provide a descriptive name for your custom server class.

➤ Check **Customize Record process** if you intend to implement the Test Record process in QuickTest.

If you check **Customize Record process**, the wizard creates a framework of code for the implementation of recording steps.

Do not select this check box if you are going to create the script manually in QuickTest, or if you are going to use the Test Record functions of the parent test object to which this control will be mapped. Note that if you implement Test Record, the implementation replaces that of the parent object. You must implement all required Test Record functionality.

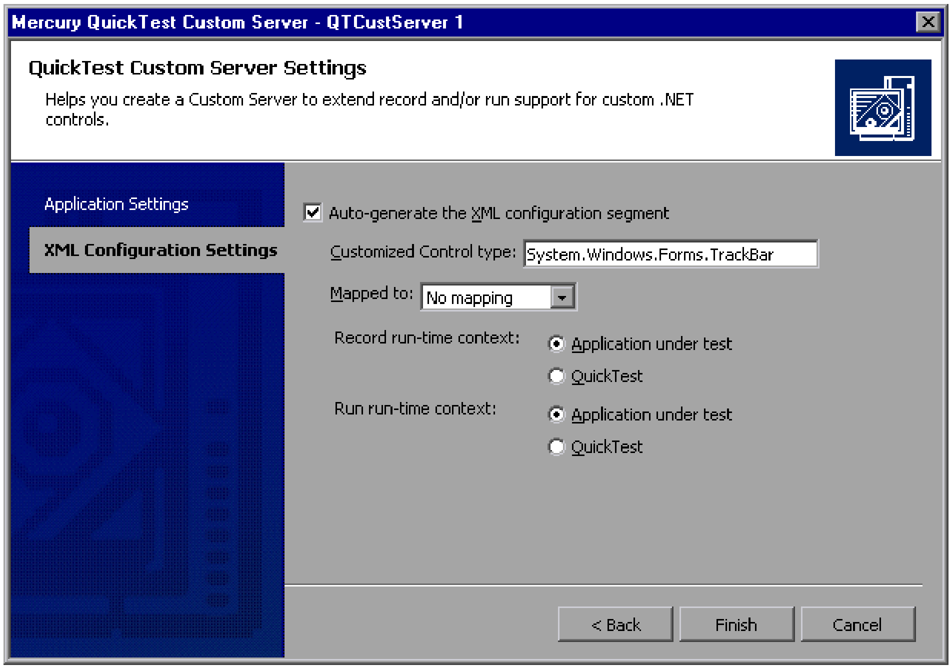
- Check **Customize Run process** if you intend to implement the Test Run functions for the custom control. Enter a name for the Replay Interface you will create in the **Replay interface name** box.

  If you check **Customize Run process**, the wizard creates a framework of code to implement Test Run support.

  Check **Customize Run process** if you are going to override any of the existing test object's methods, or extend the test object with new methods.

- Check **Generate comments and sample code** if you want the wizard to add comments and samples in the code that it generates.

6 Click **Next**. The XML Configuration Settings page of the wizard opens.


Mercury QuickTest Custom Server - QTCustServer 1

QuickTest Custom Server Settings

Helps you create a Custom Server to extend record and/or run support for custom .NET controls.

☑ Auto-generate the XML configuration segment

Customized Control type: System.Windows.Forms.TrackBar

Mapped to: No mapping

Record run-time context: [✓ Application under test] QuickTest

Run run-time context: [✓ Application under test] QuickTest

7 Make your selections in the XML Configuration Settings page of the wizard.

- Check **Auto-generate the XML configuration segment** to have the wizard create a file, **Configuration.xml,** containing an XML segment with the configuration information for QuickTest.

- ➤ In the **Customized Control type** box, enter the full type name of the control for which you are creating the Custom Server, including all wrapping namespaces, for example, System.Windows.Forms.CustomCheckBox.
- ➤ In the **Mapped to** box, select the test object to which you want to map the Custom Server. If you select **No mapping**, the Custom Server is automatically mapped to the **SwfObject** test object.

  For more information, see "Understanding Test Object Mapping" on page 7.
- ➤ Select the run-time context for Test Record and/or Test Run: **Application under test** or **QuickTest**.

  For more information, see "Understanding Custom Server Run-Time Contexts" on page 5.

8 Click **Finish**. The Wizard closes and the new project opens, ready for coding.

When you click **Finish** in the wizard, a **Configuration.xml** file is created and added to the project. When you are ready to use the Custom Server, update and modify the configuration information as required and transfer it to the QuickTest configuration file as described in "Using the XML Configuration Segment" on page 18.

Using the XML Configuration Segment

The XML segment created by the wizard is used when the Custom Server is ready for deployment. Before using it, add the information that was not available when you created the project.

To use the segment when configuring QuickTest:

1 Edit the **Configuration.xml** file in the project to ensure that the information is correct. Set the **DllName** element value to the location where you will install the Custom Server. If Test Record and/or Test Run are to be loaded in different run-time contexts, edit the **Context** value accordingly.

2 Copy the entire **<Control>...</Control>** node. Do not include the enclosing **<Controls>** tags.

3 Open the QuickTest Professional .NET Add-in configuration file, <**QuickTest Professional**>**dat****SwfConfig.xml**. Paste the **Control** node from **Configuration.xml** at the end of the file, before the closing </**Controls**> tag.

4 Save the file.

For more information, see "Configuring QuickTest to Use the Custom Server" on page 35.

Implementing Test Record for a Custom Control Using the .NET DLL

Recording a business component or test script on a control means listening to the activity of that control, translating that activity into test object method calls, and writing the method calls to the script. Listening to the activities on the control is done by listening to control events, hooking Windows messages, or both.

To implement Test Record, implement the methods in the IRecord interface created by the wizard. Add all the functionality required by your application. Your Test Record implementation does not inherit from the parent test object to which the custom control is mapped. It replaces the parent object's Test Record implementation entirely. Therefore, if you need any of the parent object's functionality, code it explicitly.

Before reading this section, make sure you are familiar with "Understanding Custom Server Run-Time Contexts" on page 5.

For more details about the interfaces, classes, enumerations, and methods in this section, refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

This section describes:

➤ Implementing the IRecord Interface

➤ Writing Test Object Methods to the Script

Implementing the IRecord Interface

To implement the IRecord interface, override the call-back methods described in this section, and add the details of your implementation in your event handlers or message handler.

Callback Method InitEventListener

CustomServerBase.InitEventListener is called by QuickTest when your Custom Server is loaded. Add your event and message handlers in this method.

1 Implement handlers for the control's events.

A typical handler captures the event and writes a method to the test script. This is an example of a simple event handler:

```
public void OnMouseDown(object sender, MouseEventArgs  e)
{
    // Get the event.
    if(e.Button != System.Windows.Forms.MouseButtons.Left)
    return;
    /*
    For more complex events, here you would get any
    other information you need from the control.
    */
    // Write the test object method to the script
    RecordFunction("MouseDown",
    RecordingMode.RECORD_SEND_LINE,
    e.X,e.Y);
}
```

For more information, see "Writing Test Object Methods to the Script" on page 23.

2 Add your event handlers in InitEventListener:

```
public override void InitEventListener()
{
    .....
    // Adding OnMouseDown handler.
    Delegate  e = new MouseEventHandler(this.OnMouseDown);
    AddHandler("MouseDown", e);
    .....
}
```

Note that if Test Record will run in the **Application under test** context, you can use the syntax:

```
SourceControl.MouseDown += e;
```

If you use this syntax, you must release the handler in ReleaseEventListener.

3 Add a Remote Event Listener.

If your Custom Server will run in the **QuickTest** context, use a remote event listener to handle events. Implement a remote listener of type EventListenerBase that handles the events, and add a call to AddRemoteEventListener in method InitEventListener.

```
public class EventsListenerAssist : EventsListenerBase
{
    // class implementation.
}
public override void InitEventListener()
{
    ...
    AddRemoteEventListener(typeof(EventsListenerAssist));
    ...
}
```

When you implement a remote event listener, you must override EventListenerBase.InitEventListener and EventListenerBase.ReleaseEventListener in addition to overriding these call-back functions in CustomServerBase. The use of these two EventListenerBase call-backs is the same as for the CustomServerBase call-backs. For details, refer to the EventsListenerBase class in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Note that when you handle events from the **QuickTest** context, the event arguments must be serialized. For details, refer to CustomServerBase.AddHandler(*String, Delegate, Type*) and the IEventArgsHelper Interface in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

To avoid the complications of remote event listeners, run your event handlers in the **Application under test** context, as described above.

Callback Method OnMessage

OnMessage is called on any window message hooked by QuickTest. If Test Record will run in the **QuickTest** context and message handling is required, implement the message handling in this method.

If Test Record will run in the **Application under test** context, do not override this function.

For details, refer to CustomServerBase.OnMessage in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Callback Method GetWndMessageFilter

If Test Record will run in the **QuickTest** context and listen to windows messages, override this method to inform QuickTest whether the Custom Server will handle only messages intended for the specific custom object window, or whether it will handle messages from child windows, as well.

For details, refer to IRecord.GetWndMessageFilter in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Callback Method ReleaseEventListener

QuickTest Professional calls this method at the end of the recording session. In ReleaseEventListener, unsubscribe from all the events to which the Custom Server was listening. For example, if you subscribed to OnClick in InitEventListener with this syntax,

```
SourceControl.Click += new EventHandler(this.OnClick);
```

you must release it:

```
public override void ReleaseEventListener()
{
    ....
    SourceControl.Click -= new EventHandler(this.OnClick);
    ....
}
```

However, if you subscribe to the event with the AddHandler method, QuickTest unsubscribes automatically.

Writing Test Object Methods to the Script

When information about activities of the control has been received, whether in the form of events, Windows messages, or a combination of both, this information must be processed as appropriate for the application and a script step written as a test object method call.

To write a script step, use the RecordFunction method of the CustomServerBase class or the EventsListenerBase, as appropriate.

Since it is sometimes impossible to know how an activity should be processed until the next activity occurs, there is a mechanism for storing a script step and deciding in the subsequent call to **RecordFunction** whether to write it to the script. For details, refer to RecordingMode Enumeration in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

In order to determine the parameter values for the test object method call, it may be necessary to retrieve information from the control that is not available in the event arguments or Windows message. If the Custom Server Test Record object is running in the **Application under test** context, use the SourceControl property of the CustomServerBase class to obtain direct access to the public members of the control. If the control is not thread-safe, use the ControlGetProperty method to retrieve control state information.

Implementing Test Run for a Custom Control Using the .NET DLL

Defining test object methods for Test Run means specifying the actions to perform when the method is encountered in the business component or test script. Typically, the implementation of a test object method performs several of the following actions:

- Sets the values of attributes of the control object
- Calls a method of the control object
- Makes mouse and keyboard simulation calls

- Reports a step outcome to QuickTest
- Reports an error to QuickTest
- Makes calls to another library (to show a message box, write custom log, and so forth)

The custom control is mapped to a parent QuickTest test object. If there is no explicit mapping, it is mapped to **SwfObject**. The test object type that supports the custom control is the new type that consists of the members of the parent object or overrides of those members, and new members added by this Custom Server.

Define custom Test Run methods if you are overriding existing methods of the parent test object, or if you are extending the parent test object by adding new methods.

Ensure that all test object methods recorded are implemented in Test Run, either by the parent test object, or by this Custom Server.

To define custom Test Run methods, define an interface and identify it to QuickTest as the Test Run interface by applying the ReplayInterface attribute to it. Only one replay interface can be implemented in a Custom Server. If your interface defines methods with the same names as existing methods of the parent object, the interface methods override the test object implementation. Methods that do not have the same name as a method of the parent object, are added as new methods.

Start a test object method implementation with a call to PrepareForReplay, specify the activities to perform, and end with a call to ReplayReportStep and/or ReplayThrowError.

For more details, refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Running Code under Application Under Test from the QuickTest Context

When the Custom Server is running in the **QuickTest** context, there is no direct access to the control, which is in a different run-time process. To access the control directly, run part of the code in the **Application under test** context.

To launch code from the **QuickTest** context that will run under the **Application under test** context, implement an assistant class that inherits from CustomAssistantBase. To create an instance of an assistant class, call CreateRemoteObject. Before using the object, attach it to the control with SetTargetControl.

Once SetTargetControl has been called, there are two ways to call methods of the assistant. If the method can run in any thread of the **Application under test** process, read and set control values and call control methods with the simple obj.Member syntax:

```
int i = oMyAssistant.Add(1,2);
```

If the method must run in the control's thread, use the InvokeAssistant method:

```
int i = (int)InvokeAssistant(oMyAssistant, "Add", 1, 2);
```

EventListenerBase is an assistant class that supports listening to control events.

API Overview

This section provides a quick reference of the most commonly used API calls. For more details, refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Test Record Methods

AddHandler	Adds an event handler as the first handler of the event.
RecordFunction	Records a line in the Test script.

Test Record Callback Methods

GetWndMessageFilter	Called by QuickTest to set the Windows Message filter.
InitEventListener	Called by QuickTest to load event handlers and start listening for events.
OnMessage	Called when window message hooked by QuickTest.
ReleaseEventListener	Stops listening for events.

Test Run Methods

DragAndDrop, KeyDown, KeyUp, MouseClick, MouseDblClick, MouseDown, MouseMove, MouseUp, PressKey, PressNKeys, SendKeys, SendString	Mouse and keyboard simulation methods.
PrepareForReplay	Prepares the control for a replay action.
ReplayReportStep	Writes an event to the test report.
ReplayThrowError	Generates an error message and changes the reported step status.

ShowError	Displays the .NET warning icon.
TestObjectInvokeMethod	Invokes one of the methods exposed by the test object's IDispatch interface.

Cross-Process Methods

AddRemoteEventListener	Creates an EventListener instance in the **Application under test** process.
CreateRemoteObject	Creates an instance of an Assistant object in the **Application under test** process.
GetEventArgs (IEventArgs)	Retrieves and deserializes the EventArgs object.
Init (IEventArgsHelper)	Initializes the Event Arguments helper class with an EventArgs object.
InvokeAssistant	Invokes a method of a CustomAssistantBase class in the control's thread.
InvokeCustomServer	Invokes the Custom Server's methods running in the **QuickTest** process from the **Application under test** process.
SetTargetControl	Attaches to the source control object by the control's window handle.

General Methods

ControlGetProperty	Retrieves a property of a control that is not thread-safe.
ControlInvokeMethod	Invokes a method of a control that is not thread-safe.
ControlSetProperty	Sets a property of a control that is not thread-safe.
GetSettingsValue	Gets a Parameter value from the settings of this control in the configuration file.
GetSettingsXML	Returns the settings of this control as entered in the configuration file.

4

Using an XML File to Extend Support for a Custom Control

You can extend support for a customized .NET control using an XML file. Using an XML file enables you to extend support without a program development environment.

This chapter describes:

- About Using an XML File to Extend Support for a Custom Control
- Understanding the Control Definition XML File
- Example of a Control Definition XML File

About Using an XML File to Extend Support for a Custom Control

You can implement custom control support without programming a .NET DLL by entering the appropriate Test Record and Test Run instructions in a Control Definition XML file. You can tell QuickTest Professional to load the instructions by pointing to this control definition file in the QuickTest configuration file, **SwfConfig.xml**.

When using this technique, you do not have the support of the .NET development environment—the object browser and the debugger. However, by enabling the implementation of custom control support without the .NET development environment, this technique enables relatively rapid implementation, even in the field.

This feature is most practical either with relatively simple, well documented controls, or with controls that map well to an existing object but for which you need to replace the Test Record definitions, or replace or add a small number of test object Test Run methods.

Understanding the Control Definition XML File

The Control Definition XML file specifies the control events to be captured during recording and used to generate steps to be written to the business component or test script. These steps are calls to methods of the custom control's test object. The file also specifies the operations QuickTest performs for each method during Test Run. You do not always need to enter both a Record and a Run element.

If the custom object is mapped to a parent test object that implements either all the required Test Record methods or all the required Test Run methods, you do not need to create the section of the definition file that defines that element.

If you create a Record element, the definitions replace the Test Record implementation of the parent object entirely. If you create a Run element, it inherits the Test Run implementation of the parent object and extends it. For more information on test object mapping options, see "Understanding Test Object Mapping" on page 7.

The structure of the Control Definition XML file is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Customization>
    <Record>
        <Events>
            <!-- There are 1 to n Event elements -->
            <Event name="controlEventName" enabled="true|false">
                <RecordedCommand name="theCommandName">
                    <!-- There are 0 to n Parameter elements -->
                    <Parameter> param</Parameter>
                </RecordedCommand>
            </Event>
        </Events>
    </Record>
    <Replay>
        <Methods>
            <!-- There are 1 to n Method elements -->
            <Method name="theCommandName">
                <Parameters>
                    <!-- There are 0 to n Parameter elements -->
                    <Parameter type="theDataType" name="param 1
                        name"></Parameter>
                </Parameters>
                <MethodBody>theCommand</MethodBody>
            </Method>
        </Methods>
    </Replay>
</Customization>
```

Control Definition File Elements

- **Customization**—The root element.
- **Record**—Information about the conversion of events to steps in a test script.
- **Events**—Collection of control events to capture for generation of test script steps.
- **Event**—Contains the information needed to convert a specific event to a step in a test. It has the following attributes:
  - **name**—The name of the control event.
  - **enabled**—The flag to activate recording for this event. Can be true or false.
- **RecordedCommand**—Defines the step to be written to the script when the event described in the parent Event element is received. Has the following attribute:
  - **name**—The test object method name to write to the script.
- **Parameter**—Each Parameter element defines a parameter to be written to the script after the name of the RecordedCommand. The parameters are written to the script in the order in which they are defined in the Control Definition XML file.

A Parameter element has two possible formats. It may contain a single line of text content that will be evaluated and then written to the script. Alternatively, it may contain a short section of code to be run in order to produce the value to be written. In this case, the lang attribute must be specified, and the final value must be assigned to the return value variable, Parameter.

Several reserved words are available for use in a Parameter element:

- **Sender**—The object that fired the event.
- **EventArgs**—The object that represents EventArgs parameter of the Event Handler.
- **Parameter**—The return value of the code.

The Parameter element has the following optional attribute:

- **lang**—If the element contains code, the lang attribute specifies the programming language. Currently, C# is supported.

- **Replay**—Information about the conversion of test object methods to the activities to be performed during the Test Run session.
- **Methods**—Collection of Method elements.
- **Method**—Defines a method added to the test object interface. It has the following attribute:
  - **name**—The test object method name.
- **Parameters**—Collection of Parameter elements.
- **Parameter**—Each Parameter element contains instructions for reading a command line parameter from the script. The order of Parameter elements must be the same as the order of the command line parameters in the script.
- These parameters are used in the MethodBody element to create the method call. Each parameter element has the following attributes:
  - **type**—The data type of the value as it will be used in the MethodBody.
  - **name**—The name by which to refer to the value in the MethodBody.
- **MethodBody**—A series of C# instructions to perform when the test object method is executed.

  The reserved word RtObject refers to the run-time object.

Example of a Control Definition XML File

The following example shows the handling of an object whose value changes at each MouseUp event. The value is in the Value property of the object. The MouseUp event handler has Button, Clicks, Delta, X, and Y event arguments.

The Record element describes the conversion of the MouseUp event to a SetValue command. The Replay mode defines the SetValue command as setting the value of the object to the recorded Value and displaying the position of the mouse pointer for debugging purposes.

```
<?xml version="1.0" encoding="UTF-8"?>
<Customization>
  <Record>
    <Events>
      <Event name="MouseUp" enabled="true">
        <RecordedCommand name="SetValue">
          <Parameter>
            Sender.Value
          </Parameter>
          <Parameter lang="C#">
            String xy;
            xy = EventArgs.X + ";" + EventArgs.Y;
            Parameter = xy;
          </Parameter>
        </RecordedCommand>
     </Event>
   </Events>
  </Record>
  <Replay>
    <Methods>
      <Method name="SetValue">
        <Parameters>
          <Parameter type="int" name="Value"/>
          <Parameter type="String" name="MousePosition"/>
        </Parameters>
        <MethodBody>
          RtObject.Value = Value;
          System.Windows.Forms.MessageBox.Show(MousePosition, "Mouse
Position at Record Time");
        </MethodBody>
      </Method>
    </Methods>
  </Replay>
</Customization>
```

5

Configuring QuickTest to Use the Custom Server

The QuickTest System Windows Forms Configuration File provides QuickTest with all the information necessary to load your Custom Server with the required configuration.

This chapter describes:

- About Configuring QuickTest to Use the Custom Server
- Understanding the QuickTest System Windows Forms Configuration File

About Configuring QuickTest to Use the Custom Server

To instruct QuickTest to load Custom Servers and to pass the required configuration, enter the information in the QuickTest System Windows Forms Configuration File. The configuration file, **SwfConfig.xml**, is located in the <**QuickTest Professional installation path**>**\dat** folder.

Each control is configured in a **Control** node in the file.

Understanding the QuickTest System Windows Forms Configuration File

The structure of the **SwfConfig.xml** file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
    <Control Type=" " MappedTo=”” >
        <CustomRecord>
            <Component>
                <Context> </Context>
                <DllName></DllName>
                <TypeName></TypeName>
            </Component>
        </CustomRecord>
        <CustomReplay>
            <Component>
                <Context></Context>
                <DllName></DllName>
                <TypeName></TypeName>
            </Component>
        </CustomReplay>
        <Settings>
            <Parameter Name=""> </Parameter>
        </Settings>
    </Control>
</Controls>
```

Configuration File Elements

- **?xml**—The XML declaration, version="1.0" encoding="UTF-8”?, is required.
- **Controls**—The root element.
- **Control**—The information required to support a custom control.

  Attributes:

  - **Type**—The custom control’s full type including wrapping namespaces, for example, System.Windows.Forms.CustomCheckBox.

  - **MappedTo** (optional)—A QuickTest test object class that has similar behaviors your Custom Server will inherit.

- **Settings**—This element is generally a collection of Parameter elements. It has two uses. For .NET DLL Custom Servers, the element is optional.

  The first use is to pass information for the internal use of your Custom Server. This use is optional. You can use the Parameters for any purpose appropriate to your application. You may also use a different structure—you are not bound to a collection of Parameters. However, if you use a different structure you must parse it yourself in code, whereas the collection of Parameters structure has straightforward support in the API.

  The second use is required when extended control support is implemented with XML, and you must use the collection of Parameters structure. The full path and name of the XML file containing the implementation of the extended control support is passed in a Parameter where the **Name** attribute is ConfigPath and the value of the element is the file path name.

- **Parameter**—A value to be passed to the Custom Server at run time.
  - **Name**—The name of the Parameter.
- **CustomRecord**—The information required for the Test Record.
- **CustomReplay**—The information required for the Test Run.

  The CustomRecord and CustomReplay nodes both contain a Component node. Not all Component sub-elements apply to both processes.

- **Component**—The Custom Server component data.
- **Context**—The Custom Server run-time context and the coding option. There are three options:
  - **AUT**—The run-time context is the **Application under test** process. The support is implemented as a .NET .DLL Custom Server.
  - **QTP**—The run-time context is the **QuickTest** process. The support is implemented as a .NET DLL Custom Server.
  - **AUT-XML**—The run-time context is the **Application under test** process. The support is implemented in an XML file.

- **DllName**—The filename of the DLL in which the user's class type is defined. Applies to the .NET DLL coding option only. There are two formats for identifying the assembly:
  - The full path and file name.
  - If the Custom Server assembly is installed in the global assembly cache (GAC), pass the type name with the standard syntax, for example:

```
    myQTCustomServer
```
or

```
    myQTCustomServer, Version=1.0.1234.0
```
or

```
    myQTCustomServer, Version=1.0.1234.0, Culture="en-US",
    PublicKeyToken=b77a5c561934e089c
```

- **TypeName**—The name of the type created by the Custom Server, including wrapping namespaces. Applies to the .NET DLL coding option only.

Example of a Configuration XML File

Following is an example of a file that configures QuickTest to recognize two controls.

Support for the **CustomMyListView.CustListView** control is implemented in a .NET DLL Custom Server. **MyListView** is mapped to the **SwfListView** test object, and runs in the **Application under test** context. The Custom Server is not installed in the GAC.

Support for the **mySmileyControls.SmileyControl2** control is implemented in an XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
  <Control
        Type="MyCompany.WinControls.MyListView “
        MappedTo="SwfListView" >

    <CustomRecord>
      <Component>
        <Context>AUT</Context>
        <DllName>C:\MyProducts\Bin\CustomMyList View.dll</DllName>
```

```
                <TypeName>CustomMyListView.CustListView</TypeName>
            </Component>
        </CustomRecord>

         <CustomReplay>
             <Component>
                <Context>AUT</Context>

                <DllName>C:\MyProducts\Bin\CustomMyList View.dll</DllName>
                <TypeName>CustomMyListView.CustListView</TypeName>
            </Component>
        </CustomReplay>

        <Settings>
            <Parameter Name="sample name">sample value</Parameter>
        </Settings>
    </Control>

    <Control Type="mySmileyControls.SmileyControl2">
        <Settings>
            <Parameter Name="ConfigPath">d:\Qtp\bin\ConfigSmiley.xml
            </Parameter>
        </Settings>

         <CustomRecord>
            <Component>
                <Context>AUT-XML</Context>

            </Component>
        </CustomRecord>

         <CustomReplay>
             <Component>
                <Context>AUT-XML</Context>

            </Component>
        </CustomReplay>
    </Control>
</Controls>
```

6

Step-by-Step Tutorial

In this tutorial, you will learn how to build a Custom Server for a Microsoft TrackBar control that enables QuickTest Professional to record and run a **SetValue** operation on the control.

This chapter describes:

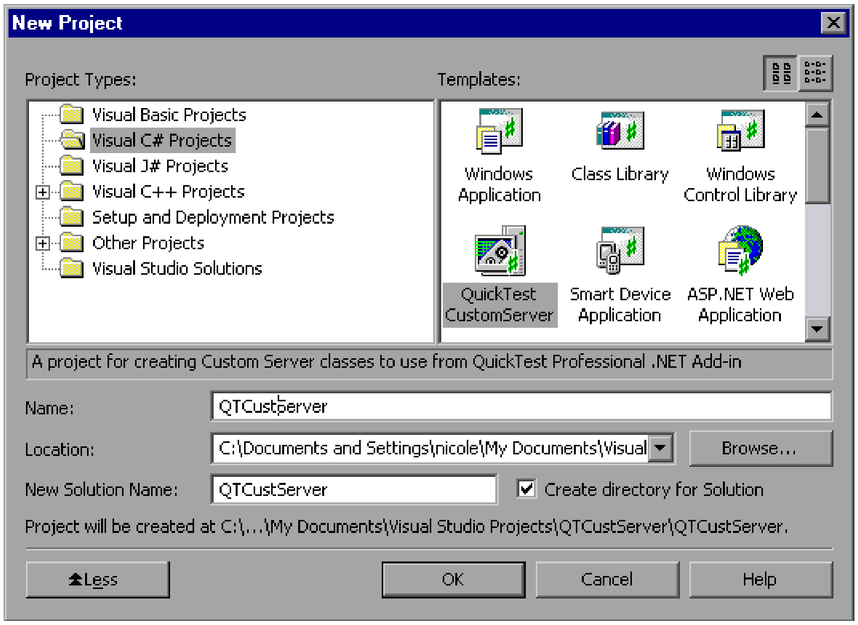
Creating a New Custom Server Project

The first step in creating support for the TrackBar control is to create a new Custom Server project.

To create a new Custom Server project:

1 Open Microsoft Visual Studio .NET.

2 Select **File** > **New** > **Project**. The New Project dialog box opens.



A project for creating Custom Server classes to use from QuickTest Professional .NET Add-in

Name: QTCustServer

Location: C:\Documents and Settings\nicole\My Documents\Visual

New Solution Name: QTCustServer

☑ Create directory for Solution

Project will be created at C:\...\My Documents\Visual Studio Projects\QTCustServer\QTCustServer.

3 Specify the following settings:

➤ Select **Visual C# Projects** in the **Project Types** list.

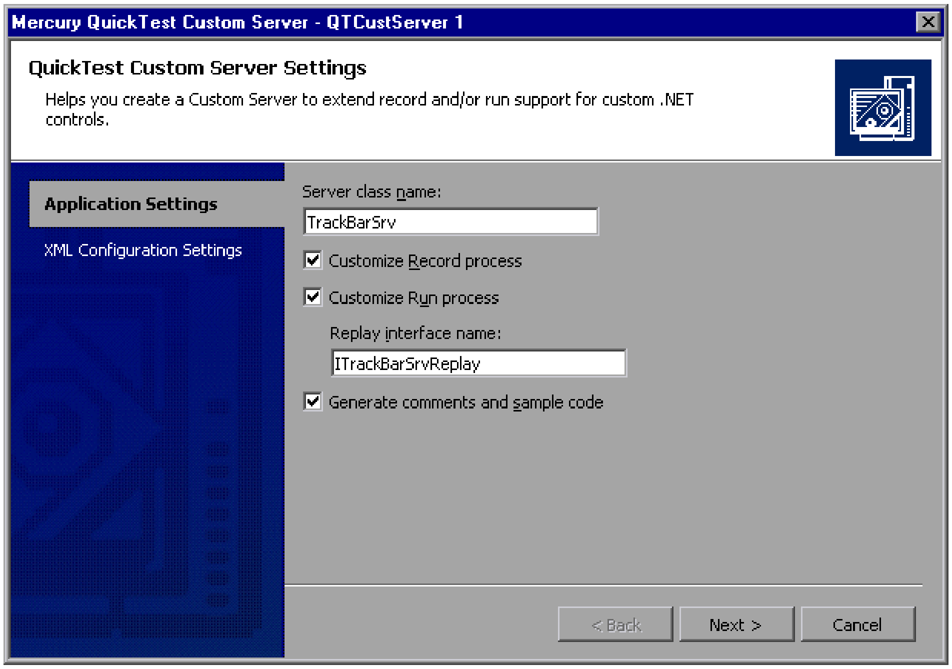
➤ Select **QuickTest CustomServer** in the **Templates** pane.

➤ In the **Name** box, specify the project name QTCustServer.

➤ In the **Location** box, specify the location in which to save your project.

➤ Accept the rest of the default settings.

4 Click the **OK** button. The QuickTest Custom Server Settings wizard opens.



Mercury QuickTest Custom Server - QTCustServer 1

QuickTest Custom Server Settings

Helps you create a Custom Server to extend record and/or run support for custom .NET controls.

Server class name: TrackBarSrv

☑ Customize Record process

☑ Customize Run process

Replay interface name: ITrackBarSrvReplay

☑ Generate comments and sample code

5 In the Application Settings page, specify the following settings:

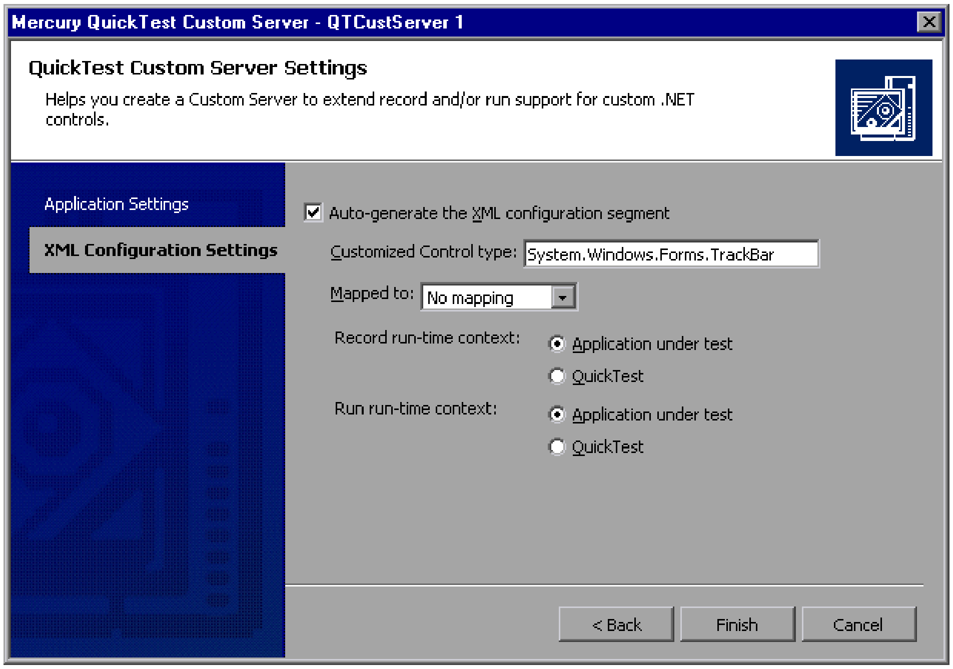
➤ In the **Server class name** box, enter TrackBarSrv.

➤ Select the **Customize Record process** check box.

➤ Select the **Customize Run process** check box.

➤ Accept the rest of the default settings.

6 Click **Next**. The XML Configuration Settings page opens.


Mercury QuickTest Custom Server - QTCustServer 1

QuickTest Custom Server Settings

Helps you create a Custom Server to extend record and/or run support for custom .NET controls.

[✓] Auto-generate the XML configuration segment

Customized Control type: System.Windows.Forms.TrackBar

Mapped to: No mapping

Record run-time context: [● Application under test] ○ QuickTest

Run run-time context: [● Application under test] ○ QuickTest

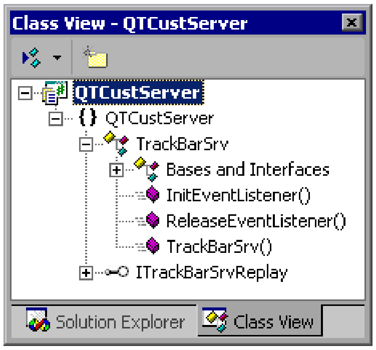
7 In the XML Configuration Settings page, specify the following settings:

➤ Make sure the **Auto-generate the XML configuration segment** check box is selected.

➤ In the **Customized Control type** box, enter System.Windows.Forms.TrackBar.

➤ Accept the rest of the default settings.

8 Click **Finish**. In the Class View window, you can see that the wizard created a **TrackBarSrv** class derived from the **CustomServerBase** class and **ITrackBarSrvReplay** interface.



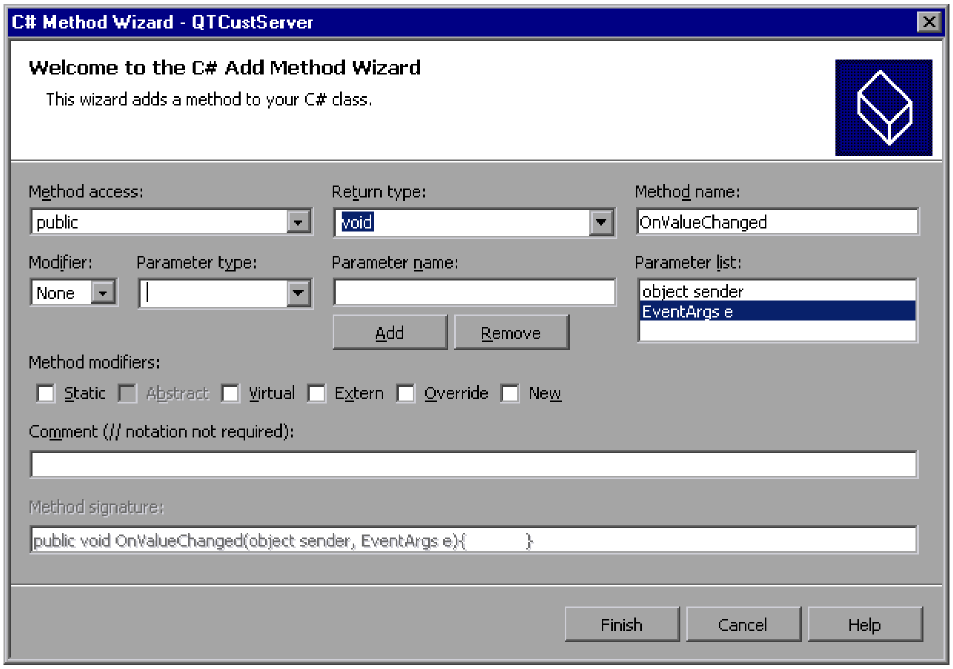
Implementing Test Record Logic

You will now implement the logic that records a **SetValue(X)** command when a **ValueChanged** event occurs, using an event handler function.

To implement the Test Record logic:

1 Right-click the **TrackBarSrv** class name in the Class View window and select **Add > Add Method**.

The C# Add Method Wizard opens.



C# Method Wizard - QTCustServer

Welcome to the C# Add Method Wizard

This wizard adds a method to your C# class.

Method access: public

Return type: void

Method name: OnValueChanged

Modifier: None

Parameter type:

Parameter name:

Parameter list: object sender, EventArgs e

Method modifiers: [ ] Static [ ] Abstract [ ] Virtual [ ] Extern [ ] Override [ ] New

Comment (// notation not required):

Method signature: public void OnValueChanged(object sender, EventArgs e){ }

2 Use the C# Add Method Wizard to add a new method with the following signature:

```
public void OnValueChanged(object sender, EventArgs e) { }
```

Note: Alternatively, you can add the new method manually without using the C# Add Method Wizard.

---

3 Add the following implementation to the function we just added:

```
public void OnValueChanged(object sender, EventArgs e)
{
 System.Windows.Forms.TrackBar trackBar =
(System.Windows.Forms.TrackBar)sender;
 // get the new value
 int newValue = trackBar.Value;
 // Record SetValue command to the test script
 RecordFunction("SetValue", RecordingMode.RECORD_SEND_LINE,
newValue);
}
```

4 Register the OnValueChanged event handler for the ValueChanged event, by adding the following code to the InitEventListener method:

```
public override void InitEventListener()
{
  Delegate e = new System.EventHandler(this.OnValueChanged);
  AddHandler("ValueChanged", e);
}
```

Implementing Test Run Logic

You will now implement a **SetValue** method for the test or business component Test Run.

To implement the Test Run logic:

1 Add the following method definition to the **ITrackBarSrvReplay** interface:

```
[ReplayInterface]
public interface ItrackBarSrvReplay
{
   void SetValue(int newValue);
}
```

2 Add the following method implementation to the **TrackBarSrv** class:

```
public void SetValue(int newValue)
{
 System.Windows.Forms.TrackBar trackBar =
(System.Windows.Forms.TrackBar)SourceControl;
 trackBar.Value = newValue;
}
```

3 Build your project.

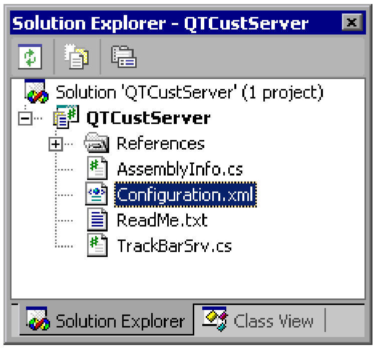
Note: You can see the full source code of the **TrackBarSrv** class in "Understanding the TrackBarSrv.cs File" on page 50.

Configuring QuickTest Professional

Now that you have created the QuickTest Custom Server, you need to configure QuickTest Professional to use this Custom Server when recording and running tests on the TrackBar control.

To configure QuickTest Professional to use the Custom Server:

1 In the Solution Explorer window, click the **Configuration.XML** file.



The following content should be displayed:

```
<!-- Merge this XML content into file "<QuickTest Professional>\dat\
    SwfConfig.xml". -->
    <Control Type="System.Windows.Forms.TrackBar">
        <CustomRecord>
            <Component>
                <Context>AUT</Context>
                <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
                    </DllName>
                <TypeName>QTCustServer.TrackBarSrv</TypeName>
            </Component>
        </CustomRecord>
        <CustomReplay>
            <Component>
                <Context>AUT</Context>
                <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
                    </DllName>
                <TypeName>QTCustServer.TrackBarSrv</TypeName>
            </Component>
        </CustomReplay>
        <!--<Settings>
             <Parameter Name="sample name">sample value</Parameter>
        </Settings> -->
    </Control>
```

2 Select the <**Control**>...</**Control**> segment and select **Edit** > **Copy** from the menu.

3 Open the **SwfConfig.xml** file located in <**QuickTest Professional installation folder**>**dat**.

4 Paste the <**Control**>...</**Control**> segment you copied from **Configuration.xml** into **SwfConfig.xml**, under the <**Controls**> tag in **SwfConfig.xml**. After you paste the segment, the **SwfConfig.xml** file should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
    <Control Type="System.Windows.Forms.TrackBar">
        <CustomRecord>
            <Component>
                <Context>AUT</Context>
                <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
                    </DllName>
                <TypeName>QTCustServer.TrackBarSrv</TypeName>
            </Component>
        </CustomRecord>
        <CustomReplay>
            <Component>
                <Context>AUT</Context>
                <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
                    </DllName>
                <TypeName>QTCustServer.TrackBarSrv</TypeName>
            </Component>
        </CustomReplay>
    </Control>
</Controls>
```

5 Make sure that the <**DllName**> elements contain the correct path to your Custom Server DLL.

6 Save the **SwfConfig.xml** file.

Testing the Custom Server

You can now check that QuickTest records and runs tests or components as expected on the custom TrackBar control.

To test the Custom Server:

1 Open QuickTest Professional with the .NET Add-in loaded.

2 Start recording on a .NET application with a System.Windows.Forms.TrackBar control.

3 Click the **TrackBar** control. QuickTest should record commands such as:

```
SwfWindow("Form1").SwfObject("trackBar1").SetValue 2
```

4 Run the test. The TrackBar control should receive the correct values.

Understanding the TrackBarSrv.cs File

Following is the full source code for the **TrackBarSrv** class.

```
using System;
using Mercury.QTP.CustomServer;

namespace QTCustServer
{
    [ReplayInterface]
    public interface ITrackBarSrvReplay
    {
        void SetValue(int newValue);
    }
    public class TrackBarSrv:
        CustomServerBase,
        ITrackBarSrvReplay
    {
        public TrackBarSrv()
        {
        }

        public override void InitEventListener()
        {
            Delegate  e = new System.EventHandler(this.OnValueChanged);
            AddHandler("ValueChanged", e);
        }

        public override void ReleaseEventListener()
        {
        }

        public void OnValueChanged(object sender, EventArgs e)
        {
            System.Windows.Forms.TrackBar trackBar =
                        (System.Windows.Forms.TrackBar)sender;
            int newValue = trackBar.Value;
            RecordFunction("SetValue",
                        RecordingMode.RECORD_SEND_LINE,
                        newValue);
        }

        public void SetValue(int newValue)
        {
            System.Windows.Forms.TrackBar trackBar =
                        (System.Windows.Forms.TrackBar)SourceControl;
            trackBar.Value = newValue;
        }
    }
}
```

Index