Error Handling

> There are two ways of producing error-free software.
>
> But only the third will work …
>
>
> **Unknown Author**

- There are bugs than can eliminate an entire project.
- There are also bugs that can kill people.
- Reliability is a major characteristic of high-quality software.
- Software should behave well in nearly every situation.
- During the development process, we try to avoid, detect and remove as many errors as possible.

We cannot assume that the delivered software is free of errors. Therefore, we have to think about mechanisms to protect the productive system from unacceptable behavior while an erroneous situation occurs.

Why Error Handling?

Reliability is a major characteristic of high-quality software. Software should behave well in nearly every situation. During the development process, we try to avoid, detect and remove as many errors as possible. Because I consider that developing **QuickTest** scripts is the same a software development, and both must be treat the same way. Nevertheless, we cannot assume that the delivered software is free of errors. Therefore, we have to think about mechanisms to protect the productive system from unacceptable behavior while erroneous situations occur.

If you are an experience automation tools developer, or automation tools user, how many times did you have to rerun a script to investigate what went wrong with the previous run? After each run, you have to make an analysis on the results. A good detailed report can make the difference.

Fault Tolerance

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong. it usually turns out to be impossible to get at or repair.

Douglas Adams

A system able to recover from errors and to restore normal operation is called **fault tolerant**. Depending on the criticality of a system fault tolerance has many facets. Especially in safety critical applications

In the domain of business information systems, the major concern is to guarantee the correctness and integrity of the data and to prevent any corruption and loss of data. In an error situation (especially design faults), it is generally not possible to correct the fault on-line and so it is probable that the same error occurs again. Furthermore, there is the danger to proceed with inconsistent or corrupted data. Thus in most cases we do not try to recover (only if sensible) from errors in order to continue normal operation, but we try to terminate (when necessary and possible) the application in a consistent way.

Error recovery in fault tolerant systems is very complex and expensive. It is used in safety critical applications where availability and timing are important requirements. In the domain of business information systems, which are often critical too but with other requirements, it is usually more effective to invest in error prevention during development and robustness of the productive system.

What to Do?

To behave well in nearly every situation the Scripts has to cope not only with the normal situations but also with a number of unexpected situations. Thus the motto should be: **Expect the Unexpected!**

During normal operation a script that test a program is in a „good state", but normally the set of good states is only a subset of the set of all technically possible states. So there will be many „bad states" and we want to avoid the script that test the application running into such a state.

With error handling we want to reach the following goals:
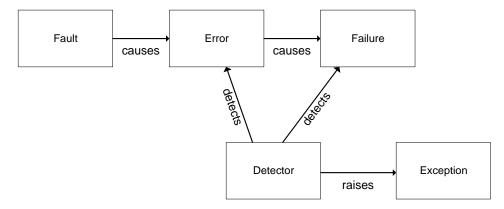
- We want to be prepared for the script goes into a bad state. In most cases, there will be no return from the bad state to a good state and the program has to be terminated in a controlled way. Of course, whenever possible the program tries to get out of trouble by itself, but normally it would be very difficult to recover from a serious error situation.
- We want to minimize the set of disastrous states by foreseeing these situations and thus converting them to expected situations.

To make the script robust against bad states we have to extend the architecture by the following error handling facilities:

- Error detection
- Error handling
- Propagation of error information
- Administration and collection of all information which will be important for the user or developer to analyze and resolve an error
- Administration of error messages which will be reported to the user/analyzer in case of errors (error reporting)

Terminology

Cause Relationship

Fault

A fault is the origin of any misbehavior. It is the adjudged or hypothesized cause of an error.

Error

This is the part of a system state that is liable to lead to a failure. With respect to a fault it is a manifestation (or in object-oriented terms an instance) of a fault in a system.

User errors are not part of the above error-definition. A user error is a mistake made by a user when operating a software system. The system is able to react to these mistakes, because it is designed to expect such situations and it is a part of the required functionality. The handling of those situations, which of course can be abnormal, should be treated in the user interface component of the system. In contrast to an error, a user error cannot result in a system crash, but like a system error, a user error normally can result in an exception raised by a system component.

Failure

A failure is a deviation of the delivered service from compliance with the specification. Whereas an error characterizes a particular state of a system, a failure is a particular event namely the transition from correct service delivery to incorrect service.

Detector

Before software can react to any error, it has to detect one first. If we look at the error handling as a separate software system, errors and failures of an application are the phenomena the error handling system observes. Thus a detector is the interface between the error and failures happening outside and their internal handling. The number, place and kind of detectors have great impact on the quality of an error handling system.

Exception

Generally, any occurrence of an abnormal condition that causes an interruption in normal control flow is called an **exception**. It is said that an **exception** is raised (thrown) when such a condition is signaled by a software unit. In response to an exception, the control is immediately given to a designated handler for the exception, which reacts to that situation (exception handler). The handler can try to recover from that exception in order to continue at a predefined location or it cleans up the environment and further escalates the exception. To raise an exception, a program first has to detect an error or failure

Error Handling Guidelines

Within every Action/Reusable Action/Function you have to think about possible exceptions and how to handle them. Especially in the uppermost layers of the script, it is important to handle all exceptions.

- We want to relieve the developer from writing similar error handling code for every method.
- We want to minimize the set of disastrous states by foreseeing these situations and thus converting them to expected situations.
- We should not bother everybody with technical error handling stuff. Some developers want to concentrate on domain code.
- Individual error handling should be possible whenever necessary. For instance, a first version contains a very simple error handling which we want to refine in later versions.
- Error handling should be consistent to keep it maintainable.
- It is important to provide some error handling for unexpected exceptions.
- Changes to application code (e.g. new error detectors) or extensions should not result in change requirements for a large number of error handlers (e.g. all error handlers that are reachable by call paths of changed class methods).

Error Handling in QuickTest Professional

QuickTest and **VBScript** give the developer some tools to handle errors and Exceptions. Every Tool or method references on the 5 topics of **Error handling**. **Fault, Error, Detector, Failure and Exception.**
I Will start with the easiest topic, handling user and/or developer faults.

Error handling in **QuickTest** is very different; usually we will end the script, with a detailed message to the user, even an additional bitmap capture.

Finding errors in your scripts. Instead, use error-handling techniques to allow your program to continue executing even though a potentially fatal error has occurred. Ordinarily, all runtime errors that are generated by the **VBScript** engine are fatal, since execution of the current script is halted when the error occurs. Error handling allows you to inform the user of the problem and either halt execution of the program or, if it is prudent, continue executing the program.

Data Validation

When function receives parameters use a data validation procedure using **VarType**, **IsNumeric, IsNull, IsDate, IsEmpty VBScript** functions. When a function receives a wrong parameter type, you will get sometimes an error description not so clear and accurate.

Error Preventing

A good method for using error handling is to try to prevent them.
When an error occurred, Report it in detail. When working with GUI objects, use the Window.**Exist** property. Every **If…Then..End** If statement has the **Else** part, the same for **Select Case**. Use **Case Else**.

There is much to be said for the old maxim, "The best way to learn is by making mistakes." Once you have made a mistake, understood what you did wrong, and rectified the error, you will — in general — have a much better understanding of the concepts involved and of what's needed to build a successful application. But to save you from having to experience this painful process of trial and error in its entirety, we'd like to share with you some of the most common errors that ourselves and other programmers we've worked with have made over the years. These types of errors are actually not unique to VBScript, or in fact to VB, but to programming in general.

Syntax Errors

Syntax errors generated by typing errors. This is a tough one. Typing errors — the misspelled function call or variable name — are always going to creep into code somewhere. They can be difficult to detect, particularly because they are typing errors; we frequently train our eyes to see what should be there, rather than what is there. When the effect of the typing error is subtle, it becomes even more difficult to detect.
The Solution : Perform a syntax check every time you change your script.

Synchronization

Wikipedia says :

Synchronization (or Sync) is a problem in timekeeping which requires the coordination of events to operate a system in unison.

QuickTest Synchronization

If you do not want **QuickTest** to perform a step or checkpoint until an object in your

application achieves a certain status, you should insert a synchronization point to instruct QuickTest to pause the test until the object property achieves the value you specify (or until a specified timeout is exceeded).

Synchronization Example

The following example will demonstrate, why synchronization is necessary for specific tasks. The following task is to retrieve the order number from the flight application. The following code does not use a synchronization point.

```vbscript
Option Explicit

Dim orderNum, pathStr
pathStr = Environment( "ProductDir" ) &  "\samples\flight\app\"
SystemUtil.Run "flight4a.exe", "", pathStr,"open"
If Dialog("Login").Exist( 10000 ) Then
   Dialog("Login").WinEdit("Agent Name:").Set "dani"
   Dialog("Login").WinEdit("Password:").Set "Mercury"
   Dialog("Login").WinButton("OK").Click
Else
   Reporter.ReportEvent micFail, "Sync timeout", "Dialog 'Login' is not available."
   ExitTest( "Sync timeout" )
End If
If Not Window("Flight Reservation").Exist( 10000 ) Then
    Reporter.ReportEvent micFail, "Sync timeout", "Window 'FR' is not available."
    ExitTest( "Sync timeout" )
End If
With Window("Flight Reservation")
   .WinButton("Button").Click
   .ActiveX("MaskEdBox").Type "010109"
   .WinComboBox("Fly From:").Select "London"
   .WinComboBox("Fly To:").Select "Los Angeles"
   .WinButton("FLIGHT").Click
   .Dialog("Flights Table").WinButton("OK").Click
   .WinEdit("Name:").Set "lionel messi"
   .WinEdit("Tickets:").SetSelection 0,1
   .WinEdit("Tickets:").Set "2"
   .WinRadioButton("First").Set
   .WinButton("Insert Order").Click
   orderNum = Window("Flight Reservation").WinEdit("Order No:").GetROProperty("text")
   MsgBox orderNum, vbInformation, "Order Number"
End With
```


We've got an empty value, that is because, the script did not wait for the Message "Insert Done..." on the progress bar.

```vbscript
Option Explicit


Dim orderNum, pathStr
pathStr = Environment( "ProductDir" ) &  "\samples\flight\app\"
SystemUtil.Run "flight4a.exe", "", pathStr,"open"
If Dialog("Login").Exist( 10000 ) Then
    Dialog("Login").WinEdit("Agent Name:").Set "dani"
    Dialog("Login").WinEdit("Password:").Set "Mercury"
    Dialog("Login").WinButton("OK").Click
Else
    Reporter.ReportEvent micFail, "Sync timeout", "Dialog 'Login' is not available."
    ExitTest( "Sync timeout" )
End If
If Not Window("Flight Reservation").Exist( 10000 ) Then
     Reporter.ReportEvent micFail, "Sync timeout", "Window 'FR' is not available."
     ExitTest( "Sync timeout" )
End If
With Window("Flight Reservation")
    .WinButton("Button").Click
    .ActiveX("MaskEdBox").Type "010109"
    .WinComboBox("Fly From:").Select "London"
    .WinComboBox("Fly To:").Select "Los Angeles"
    .WinButton("FLIGHT").Click
    .Dialog("Flights Table").WinButton("OK").Click
    .WinEdit("Name:").Set "lionel messi"
    .WinEdit("Tickets:").SetSelection 0,1
    .WinEdit("Tickets:").Set "2"
    .WinRadioButton("First").Set
    .WinButton("Insert Order").Click
    ' Synchronization point "Insert Done…".
    .ActiveX("Threed Panel Control").WaitProperty "text", "Insert Done...", 10000
    orderNum = .WinEdit("Order No:").GetROProperty("text")
    MsgBox orderNum, vbInformation, "Order Number"
End With
```


Synchronization Considerations

As we saw in the previous example, synchronization required for the task.
is a good practice to use synchronization points whenever is necessary, by combining Exist property or **WaitProperty**, or just Wait. Don't forget that QTP is much faster from any human, and 100 millisecond can be a big difference to avoid an error.

Object.Exist Property

Description

Checks whether the object currently exists in the open application.

Syntax

```
returnValue = object.Exist( [TimeOut] )
```

Arguments

Parameter	Description
timeout	Optional. An **ULong** object.

Return Value

- Read-only. A **Boolean** value.

Remarks

The length of time (in milliseconds) to search for the object before returning a True or False value.

- If a timeout value is specified, **QuickTest** waits until it finds the object or until the timeout is reached.
- If the value **0** (zero) is specified, the property returns the **True** or **False** value immediately.
- If no value is specified, the value specified in the Test Settings dialog box for the Object Synchronization Timeout is used for tests. For business components, the pre-defined value of 20 seconds is used.

Object.WaitProperty Property

Description

Waits until the specified object property achieves the specified value or exceeds the specified timeout before continuing to the next step.

Syntax

```
returnValue = object.WaitProperty (PropertyName, PropertyValue, [TimeOut])
```

Arguments

Parameter	Description
propertyName	Required. A **String** value. The name of the property whose value is checked. The available properties are listed in the Identification Properties page under the Properties section for each test object.
propertyValue	Required. A **Variant** value. The value to be achieved before continuing to the next step. You can either use a simple value or you can use a comparison object together with the value to perform more complex comparisons.
timeout	Optional. A **Long** value. The time, in milliseconds, after which QuickTest continues to the next step if the specified value is not achieved. If no value is specified, QuickTest uses the time set in the Object Synchronization Timeout option in the Run tab of the Test

| | Settings dialog box. |

Return Value

- A **Boolean** value. Returns **TRUE** if the property achieves the value, and **FALSE** if the timeout is reached before the property achieves the value. A **FALSE** return value does not indicate a failed step.

Remarks

- **micGreaterThan** - Specifies that **QuickTest** waits until the property value is greater than the specified value.
- **micLessThan** - Less than; Specifies that **QuickTest** waits until the property value is less than the specified value.
- **micGreaterThanOrEqual** - Greater than or equal to; Specifies that QuickTest waits until the property value is greater than or equal to the specified value.
- **micLessThanOrEqual**: Less than or equal to; Specifies that **QuickTest** waits until the property value is less than or equal to the specified value.
- **micNotEqual**: Not equal to; Specifies that **QuickTest** waits until the property value is not equal to the specified value.
- **micRegExpMatch**: Regular expression; Specifies that QuickTest waits until the property value achieves a regular expression match with the specified value. Regular expressions are case-sensitive and must match exactly. For example, 'E.*h' matches 'Earth' but not 'The Earth' or 'earth'.

Object.WaitProperty Review

I think that the WaitProperty method had an impressive improve since QTP 9.0. is not only "wait" for a single value, is also support a range of values and regular Expressions. But is still not perfect.

1. I experienced a few bugs in this topic (**on Web Testing**), especially when you Wait (the same is for **CheckProperty** method) for a numeric value inside a WebEdit, somehow is translated to String this especially happens when the expected values are variables. The **Empty** example in the QTP does not work.
2. it cannot handle Object run-time properties, you always get an Empty string value.

Solution :

For the first problem I found that the solution is to use conversion functions.

```
Num = 8
Object.WaitPeoperty "value", Clng( num ), 1000
Object.WaitPeoperty "value", micGreaterThan( Clng( num ) ), 1000
```

For the second problem/improve the solution is quite bit more complex.
Lets say I want to Wait the a link that the font-size changed from 22px to 10px, or the property **readystate** of an object, is complete.
OK, I can write a specific function using **object.currentstyle.fontSize** property, but I wanted to combine the capabilities of the WaitProperty method inside a one generic function, not depending on the hierarchy of the property I want to retrieve . So here is the following function

First Registering the function to the object I want to test

```
RegisterUserFunc "WebEdit", "WaitROProperty", "GenWaitROProp", False
RegisterUserFunc "Page", "WaitROProperty", "GenWaitROProp", False
RegisterUserFunc "Link", "WaitROProperty", "GenWaitROProp", False
RegisterUserFunc "WebList", "WaitROProperty", "GenWaitROProp", False
```

And the function

```vbscript
Function GenWaitROProp( ByRef obj, ByVal propName, byRef propValue, ByVal timeout )
   Dim currentValue

   GenericWaitROProperty = False
   MercuryTimers.Timer( "WaitROProperty" ).Start
   Do While MercuryTimers.Timer( "WaitROProperty" ).ElapsedTime <= timeout
      Execute "currentValue = sender.Object." & propName
      If StrComp( currentValue, propValue ) = 0 Then
         MercuryTimers.Timer( "WaitROProperty" ).Stop
         MercuryTimers.Timer( "WaitROProperty" ).Reset
         GenWaitROProp = True
         Exit Function
      Else
         MercuryTimers.Timer( "WaitROProperty" ).Pause
         Wait 0, 100
         MercuryTimers.Timer( "WaitROProperty" ).Continue
       End If
    Loop
    ' ** Wait failed resetting and exit
    MercuryTimers.Timer( "WaitROProperty" ).Stop
    MercuryTimers.Timer( "WaitROProperty" ).Reset

End Function
```
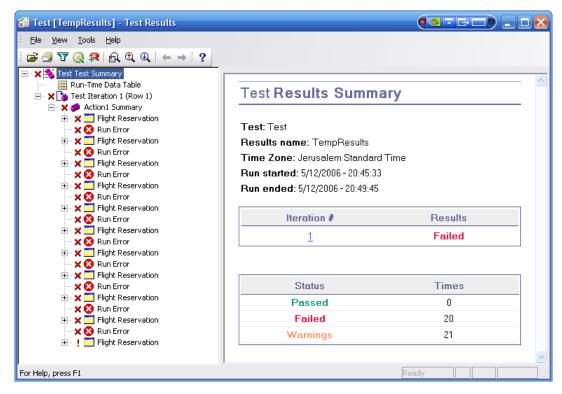
You can use this function in several ways

```vbscript
Browser("B").Page("P").WaitROPrperty "readystate", "complete", 5000
Browser("B").Page("P").Link("L").WaitROPrperty "currentstyle.fontWeight", "22px", 5000
Browser("B").Page("P").WebList("WL").WaitROProperty "rows", micGreaterThan( 1 ), 1000
```

Object Exist Property Sample
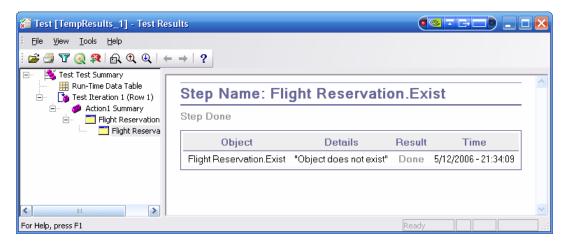
It should be very frustrate to start a new working day, by analyzing the automation results ran last night, and to find a report like in Previous figure. It's not only frustrated, it is also very sad. The errors were caused because the main window was not activated, that's all! All the other errors were directly caused by the first error. **How to prevent such a report?**

```
Option Explicit
If Window("Flight Reservation").Exist( 1000 ) Then
  Window("Flight Reservation").Activate
  Window("Flight Reservation").WinButton("NewOrder").Click
  Window("Flight Reservation").ActiveX("MaskEdBox").Type "121212"
  Window("Flight Reservation").WinComboBox("FlyFrom").Select "Denver"
  Window("Flight Reservation").WinComboBox("FlyTo").Select "London"
  Window("Flight Reservation").WinButton("FLIGHT").Click
  Window("Flight Reservation").Dialog("FlightsTable").WinButton("OK").Click
  Window("Flight Reservation").WinEdit("Name").Set "tester"
  Window("Flight Reservation").WinRadioButton("Business").Set
  Window("Flight Reservation").WinButton("InsertOrder").Click
End If
```

So, to avoid continuous errors, use the **Exist**(n) statement for a Window/Dialog on Windows add-ins Page/Frame on Web etc. Use synchronization points whenever are required, and **Sync** methods on the Web.

Still, is not enough. let's say, that the window was activated after 10 seconds. Now, you'll see the following report:

But, the object window is there!!! You will not understand what exactly happened.
The reason is that you wait 5 seconds, but the window appears after 10 seconds.
For every **If...Then...End** If statement there is a very important additional part, the **Else**.
Combination of **Else** usage and a report message, it will be no doubt what cause the unexpected result - the timeout.

```vbscript
Option Explicit
If Window("Flight Reservation").Exist( 5000 ) Then
  Window("Flight Reservation").Activate
  Window("Flight Reservation").WinButton("NewOrder").Click
  Window("Flight Reservation").ActiveX("MaskEdBox").Type "121212"
  Window("Flight Reservation").WinComboBox("FlyFrom").Select "Denver"
  Window("Flight Reservation").WinComboBox("FlyTo").Select "London"
  Window("Flight Reservation").WinButton("FLIGHT").Click
  Window("Flight Reservation").Dialog("FlightsTable").WinButton("OK").Click
  Window("Flight Reservation").WinEdit("Name").Set "tester"
  Window("Flight Reservation").WinRadioButton("Business").Set
  Window("Flight Reservation").WinButton("InsertOrder").Click
Else
  Reporter.ReportEvent micWarning, "Window unavailable", "timeout 5 sec."
End If
```

Or, by using the negative question, can save more line of code and indentation issues.
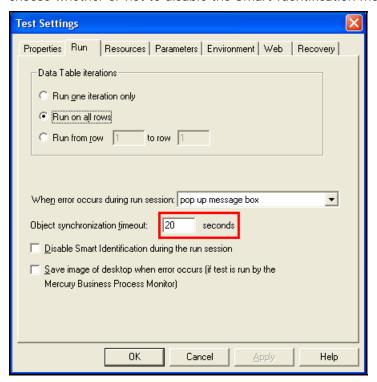
```vbscript
Option Explicit
If Not Window("Flight Reservation").Exist( 5000 ) Then
  Reporter.ReportEvent micWarning, "Window unavailable", "timeout 5 sec."
  ExitTest( "Timeout Exception" )
End If
Window("Flight Reservation").Activate
Window("Flight Reservation").WinButton("NewOrder").Click
Window("Flight Reservation").ActiveX("MaskEdBox").Type "121212"
Window("Flight Reservation").WinComboBox("FlyFrom").Select "Denver"
Window("Flight Reservation").WinComboBox("FlyTo").Select "London"
Window("Flight Reservation").WinButton("FLIGHT").Click
Window("Flight Reservation").Dialog("FlightsTable").WinButton("OK").Click
Window("Flight Reservation").WinEdit("Name").Set "tester"
Window("Flight Reservation").WinRadioButton("Business").Set
Window("Flight Reservation").WinButton("InsertOrder").Click
```

To fix the problem, just increase the timeout. The same situation with **Select Case** Statement, make a reference to an unexpected argument, parameter or value.

Default Object synchronization timeout

When you run a test, **QuickTest** performs the steps you recorded on your application or Web site.
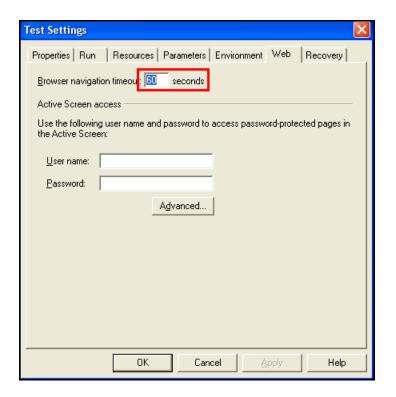
You can use the Run tab in the Test Settings dialog box (File > Settings) to choose what to do when an error occurs during the run session, set the object synchronization timeout and choose whether or not to disable the Smart Identification mechanism for the test.

Object synchronization timeout on Web

The Web tab of the Test Settings dialog box (File > Settings) provides options for recording and running tests on Web sites. You can set how long to wait for browser navigations and you can specify the Active Screen access information to use with password-protected resources in the captured Active Screen page.

Note: The Web tab is available only if the Web Add-in is installed and loaded.
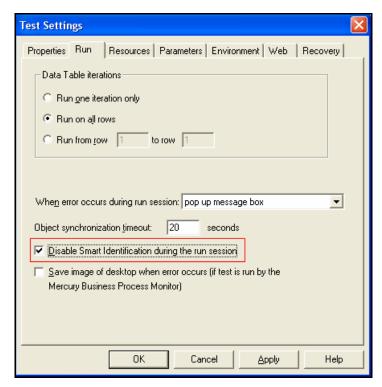
Disable/Enable Smart Identification during the run session

QTP Help : When the learned definition for an object does not enable **QuickTest** to identify an option, **QuickTest** uses the **Smart Identification** definition (if defined and enabled) to identify the object.

When **QuickTest** uses the learned description to identify an object, it searches for an object that matches all of the property values in the description. In most cases, this description is the simplest way to identify the object, and, unless the main properties of the object change, this method will work.

If **QuickTest** is unable to find any object that matches the learned object description, or if it finds more than one object that fits the description, then **QuickTest** ignores the learned description, and uses the **Smart Identification** mechanism to try to identify the object.

While the **Smart Identification** mechanism is more complex, it is more flexible. Therefore, if configured logically, a **Smart Identification** definition can probably help **QuickTest** identify an object, if it is present, even when the learned description fails.

Instructs **QuickTest** not to use the **Smart Identification** mechanism during the run session.

Note: When you select this option, the **Enable Smart Identification** check boxes in the Object Properties and Object Repository dialog boxes are disabled, although the settings are saved. When you clear this option, **the Enable Smart Identification** check boxes return to their previous on or off setting

```vbscript
Option Explicit

Dim qtApp
Set qtApp = CreateObject( "QuickTest.Application" )
If Not qtApp.Launched Then
    qtApp.Launch
    qtApp.Visible = True
End If
qtApp.Test.Settings.Run.DisableSmartIdentification = True
Set qtApp = Nothing
```

What is Smart Identification

Article: http://my330space.wordpress.com/2006/05/18/interview-question-of-qtp/

QuickTest has a unique feature called **Smart Identification**. **QuickTest** generally identifies an object by matching its test object and run time object properties. **QuickTest** may fail to recognize the dynamic objects whose properties change during run time. Hence it has an option of enabling **Smart Identification**, wherein it can identify the objects even if their properties changes during run time.

If **QuickTest** is unable to find any object that matches the recorded object description, or if it finds more than one object that fits the description, then **QuickTest** ignores the recorded description, and uses the **Smart Identification** mechanism to try to identify the object.

While the **Smart Identification** mechanism is more complex, it is more flexible, and thus, if configured logically, a **Smart Identification** definition can probably help **QuickTest** identify an object, if it is present, even when the recorded description fails.

The **Smart Identification** mechanism uses two types of properties:

1. **Base filter properties** - The most fundamental properties of a particular test object class; those whose values cannot be changed without changing the essence of the original object. For example, if a Web link's tag was changed from to any other value, you could no longer call it the same object.

2. **Optional filter properties** - Other properties that can help identify objects of a particular class as they are unlikely to change on a regular basis, but which can be ignored if they are no longer applicable.

Understanding the Smart Identification Process

QTP Help : If **QuickTest** activates the **Smart Identification** mechanism during a run session (because it was unable to identify an object based on its learned description), it follows the following process to identify the object:

- **QuickTest** "forgets" the learned test object description and creates a new object candidate list containing the objects (within the object's parent object) that match all of the properties defined in the Base Filter Properties list.

- **QuickTest** filters out any object in the object candidate list that does not match the first property listed in the Optional Filter Properties list. The remaining objects become the new object candidate list.

- **QuickTest** evaluates the new object candidate list:

- If the new object candidate list still has more than one object, QuickTest uses the new (smaller) object candidate list to repeat step 2 for the next optional filter property in the list.

  - If the new object candidate list is empty, **QuickTest** ignores this optional filter property, returns to the previous object candidate list, and repeats step 2 for the next optional filter property in the list.

  - If the object candidate list contains exactly one object, then **QuickTest** concludes that it has identified the object and performs the statement containing the object.

- **QuickTest** continues the process described in steps 2 and 3 until it either identifies one object, or runs out of optional filter properties to use.
  If, after completing the **Smart Identification** elimination process, **QuickTest** still cannot identify the object, then **QuickTest** uses the learned description plus the ordinal identifier to identify the object.
  If the combined learned description and ordinal identifier are not sufficient to identify the object, then **QuickTest** stops the run session and displays a Run Error message.

When should I use SMART Identification?

Article : http://www.sqaforums.com/showflat.php?Number=220886

- Something that people don't think about too much. But the thing is that you should disable SI while creating your test cases. So that you

- Are able to recognize the objects that are dynamic or inconsistent in their properties. When the script has been created, the SI should be enabled,

- So that the script does not fail in case of small changes. But the developer of the script should always check for the test results to verify if the SI

- Feature was used to identify an object or not. Sometimes SI needs to be disabled for

particular objects in the OR, this is advisable when you use
- **SetTOProperty** to change any of the TO properties of an object and especially ordinal identifiers like index, location and creationtime.

My conclusion, based on my experience is to **disable Smart identification**.
I have experienced many times, that Smart identification is sometimes "too smart" and my scripts does very unexpected actions on my AUT.

If you build a good repository and/or use descriptive programming. I will never need **Smart Identification**.

Actions return values

QTP is able to return value from actions.

QTP Exit Statements

ExitTest() Statement

Description

Exits the entire **QuickTest** test or Quality Center business process test, regardless of the run-time iteration settings. The pass or fail status of the test remains as it was in the step prior to the **ExitTest** statement.

Syntax

```
ExitTest [(retval)]
```

Arguments

Parameter	Description
retVal	Optional. The return value.

Example

Is always a good practice to return a value from an action or test, because the return value is directly reflected on the QTP report.
The following example handles a critical error, in which the Test set cannot continue, because the login process failed. Is obvious "No login → no test"
to see the script results, put a breakpoint after SystemUtil.Run, and simulate an error by closing the login dialog box.

```
Option Explicit

Dim orderNum, pathStr
pathStr = Environment( "ProductDir" ) &  "\samples\flight\app\"
SystemUtil.Run "flight4a.exe", "", pathStr, "open"
If Dialog("Login").Exist( 5 ) Then
   Dialog("Login").WinEdit("Agent Name:").Set "dani"
   Dialog("Login").WinEdit("Password:").Set "Mercury"
   Dialog("Login").WinButton("OK").Click
Else
   Reporter.ReportEvent micFail, "Sync timeout", "Dialog 'Login' is not available."
```

```
    ExitTest( "SystemUtil.Run Failed " )
End If
If Not Window("Flight Reservation").Exist( 10 ) Then
    Reporter.ReportEvent micFail, "Sync timeout", "Window 'FR' is not available."
    ExitTest( "Login Failed" )
End If
```


ExitAction() Statement

Description

Exits the current action, regardless of its local (action) iteration attributes. The pass or fail status of the action remains as it was in the step prior to the **ExitAction** statement.

Syntax

```
ExitAction [(retval)]
```

Arguments

Parameter	Description
retVal	Optional. The return value.

Example

In the following example I demonstrate a RETRY mechanism combining ExitAction and ExitTest. The code makes 3 retries for an action and requires two actions. 1 a reusable action and a calling action

```
[Action SendFax]
Option Explicit

If Not Window("Flight Reservation").Dialog("Fax Order").Exist( 5 ) Then
    ExitAction( micWarning )
End If
With Window( "Flight Reservation" )
    .Dialog("Fax Order").ActiveX("MaskEdBox").Type "1111111111"
    .Dialog("Fax Order").WinButton("Send").Click
    .Activate
End With
ExitAction( micPass )
```

```
[Main Action]
Option Explicit
Dim retval, nRetry : nRetry = 3
```

```
Do While nRetry > 0
   Window( "Flight Reservation" ).WinMenu("Menu").Select "File;Fax Order..."
   retval = RunAction( "SendFax", oneIteration )
   If retval = micPass Then Exit Do
   nRetry = nRetry - 1
Loop


If nRetry <= 0 Then ExitActionIteration( "Failed to send Fax" )
```

ExitActionIteration() Statement

Description

Exits the current iteration of the action. The pass or fail status of the action iteration remains as it was in the step prior to the **ExitActionIteration** statement.

Syntax

ExitActionIteration [(retval)]

Arguments

Parameter	Description
retVal	Optional. The return value.

ExitTestIteration() Statement

Description

Exits the current iteration of the QuickTest test or Quality Center business process test and proceeds to the next iteration, or exits the test run if there are no additional run-time parameter iterations. The pass or fail status of the test iteration remains as it was in the step prior to the **ExitTestIteration** statement.
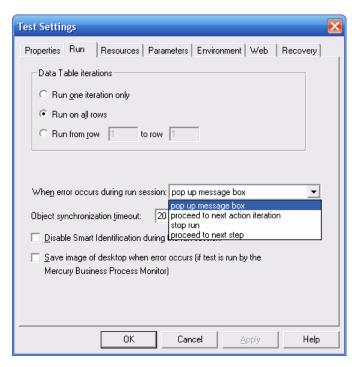
Syntax

ExitTestIteration [(retval)]

Arguments

Parameter	Description
retVal	Optional. The return value.

Response to an Error

Specifying the Response to an Error
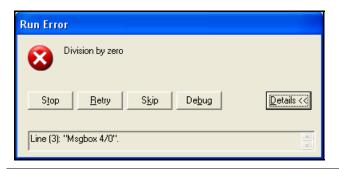
- By default, if an error occurs during the run session, **QuickTest** displays a popup message box describing the error. You must click a button on this message box to continue or end the run session.
- **popup message box**: option or you can specify a different response by choosing one of the alternative options in the list in the When error occurs during run session box:
- **Proceed to next action iteration**: **QuickTest** proceeds to the next action iteration when an error occurs.
- **stop run**: **QuickTest** stops the run session when an error occurs.
- **proceed to next step**: **QuickTest** proceeds to the next step in the test when an error occurs.

Popup message box Option

Using this option will cause the script stop, and display a popup message every time that an error occurs. When running tests, this option can terminate a test flow, because someone has to push a confirmation button. To see the error detail, you will have to open the report. When debugging and building your scripts, use this option, so **QuickTest** will let you know exactly where the error occurred and what the error detail is.

Analyzing -"popup message box"

This option is most used I think, when you're debugging and testing your scripts. Every error in the script will raise immediately a popup message, with details, there you can see the line number that causes the error, and you have an option to start debug the script, from the same point it occurred and trace the reason of the error.

Of course you can **stop** the execution immediately, or **skip** the line error.
Those options gives you full flexibility, and decide what you want. You don't want to use this option when you're testing your AUT. The following code gives you the dynamic option to choose the error handling of QTP.

```
Dim qtApp, qtTest


Set qtApp = CreateObject("QuickTest.Application")
Set qtTest = qtApp.Test
If DEBUG_MODE Then
    qtTest.Settings.Run.OnError = "Dialog"
Else
    qtTest.Settings.Run.OnError = "Stop"
End If
```

Analyzing - "proceed to next step"

I am telling you the truth, I have never use this option. And more , I have never found a good reason to use it. If my test fails in some line, the script will continue to the next command. But if the "error" line was a required operation Like "Date Of Flight" in the Flight application, is obvious that the next command will fail too. So, until today, I never found a suitable task, in which I want to use this option. Maybe you, the readers have an idea?
if you have an idea , please send me your idea, I am curious.

Analyzing - "stop run"

In my opinion this is the best option when your testing your AUT. But, you must be carefull, the test will exit at any error, and it is possible that your AUT is displaying a popup message, an error message or it just stacked, and the next following tests will fail. So, my recommendation is to create a recovery script after a test was failed. The recovery test will kill your AUT, login again, and bring the **AUT** to the desired initial condition. And, just before you "KILL" your **AUT**, make a Screenshot using Desktop.**CaptureBitmap**, and no object,**CaptureBitmap**. By doing this, you will have a visual information, of the failure, and your scripts will continue to run. So later, you can analyze you results, and find out, why the error occurs.

Analyzing - "proceed to next action iteration"

Again, my opinion for this option is like "proceed to next step". If you are not using the QTP iteration mechanism, is useless to select this option. So lets say that we are creating new orders in the Mercury "Flight Application", based on an excel Data-Driven. And suddenly when you select Date of Flight an invalid date your script will fail a popup message "Invalid Date entered ... " , and the next iteration will start, trying to Set a new date... but what about the popup? If you have a recovery scenario, it will be OK. Otherwise every time you start a new iteration you have to ask "if dialog("Error").Exist".
When you talking about "Flight Reservation", doing this is an easy task... but what about your AUT?

My conclusion about this option, is, that this option is a good candidate to be a very good option, but you need a lot of work in addition, to handle all the situations your AUT can produce. You need recovery scenarios and complex ones. So, you must ask yourself : "Is it worth it?"

Recovery Scenarios

From QTP User guide.

Unexpected events during a test run disrupt a test or may give invalid test results. For example, during a test run, an application error may occur. This error does not permit the automated test to navigate to the feature, screen, or module that needs to be tested. These unexpected errors and events are called exceptions.

When QTP encounters an exception during a run session, it displays a run error window

Introduction to a Recovery Scenario

To successfully complete a test run, you need to identify the exceptions that can occur during a test run and take appropriate action to handle the exceptions.

A recovery Scenario consists of the following components :

- **Trigger** : Specifies the exception that may occur during a run session. For example the test may be interrupted by an error popup window
- **Recovery** : Instructs QTP how to handle the exception
- **Post-Recovery** : Instruct QTP how to proceed after the recovery operations are performed.

VBScript error handling vs. Recovery Scenarios

Based on article : http://geekswithblogs.net/tmoore/archive/2006/06/26/83067.aspx

Quote " I can honestly say that they blow...and I mean goats...nickel a herd. This is one time when the tool based approach isn't terribly useful. I find the VBScript On Error/Goto 0 approach far more useful to catch and handle granular errors. I have also figured out that the recovery scenarios REALLY S-L-O-W down the test run. If you use more than a couple of them, you see a drastic degradation in run speed. We are just testing, after all, so one must ask if speed is important, but I mean it's significant enough that speed becomes an issue. There is one very handy function in recovery scenarios that I like, though: upon a trigger event firing, you can force the machine to reboot. Now, it won't restart the next test, but at least you can recover from a drastic crash. I suppose I could write something to start over after the reboot. "

For this, I am afraid the programmatic approach is still better....have to get credit for the

merit goals somewhere else.....

On a lighter note, the band has its first show this week...wish me luck!

When to use a Recovery Scenario and when to us on error resume next?

Based on article : http://www.sqaforums.com/showflat.php?Number=220886

Recovery scenarios are used when you cannot predict at what step the error can occur or when you know that error won't occur in your QTP script but could occur in the world outside QTP, again the example would be "out of paper", as this error is caused by printer device driver. "On error resume next" should be used when you know if an error is expected and don't want to raise it, you may want to have different actions depending upon the error that occurred. Use err.number & err.description to get more details about the error.

Reporting

In my concept, Reporting is the most important stage, of the automation process. I don't need to explain why reports are useful.

The ROI of Test Automation

Article from :
http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=8502

Senior managers are always interested in the return on their investments. The standard justification for choosing one type of testing over another is that the benefits exceed the costs over the lifecycle of the project or over a certain period of time. Michael Kelly describes the costs of test automation and its benefits, along with a method for determining your automation ROI. He describes areas in the automation process where improvements can dramatically increase ROI. If you are new to automation or looking for ways to improve, this presentation will help you determine if automation is right for your project and how much is the right amount.
Article pdf :
http://www.stickyminds.com/getfile.asp?ot=XML&id=8502&fn=XDD8502filelistfilename1%2Epdf

Selected quoting from his article

With the exception of my first project team out of college, in every project team since, I've had to explain either what automated testing is, why we need to do it, or what value it brings to the project. This can be difficult as each time I explain automation, I have to battle different preconceptions and experiences. Sometimes I even find myself having to develop a basic understanding of testing and software development. Wouldn't it be great if every project manager had to have a testing background or at least actual development experience? Probably not, I'm sure that would introduce all sorts of other problems. The point is it is a unique and difficult battle every time.

Educating people about automated testing requires time and mutual respect. Normally both of these are lacking on projects that are running late, are shot on budget, and have high visibility. Often automation is thought of as way to reduce the costs of testing, increase coverage, and shorten the testing cycles required for the project.

Why you need ROI analysis

In general, automated testing involves higher upfront costs for a project while providing reduced execution costs down the road. Performing return on investment (ROI) analysis on each

automation project will determine a simple approximation of cost, will help determine upfront what types of automation you want for the project, what tools will be required, and what level of skill will be required for the testing resources for the project. Not only does ROI serve as a justification for effort, but also a necessary piece of the planning process for the project.

Item	Cost	Time
Publish website and execute testing		Weekly
Develop manual tests for the site	1 tester @ $50 hr	4 days
Execute manual tests for the site	2 testers @ $50 hr	1 day
Develop automated tests for the site	1 tester @ $50 hr	10 days
Execute automated tests for the site	1 tester @ $50 hr	1 hour to review logs
Maintain manual test cases	1 tester @ $50 hr	1 day every two weeks
Maintain automated test scripts	1 tester @ $50 hr	1 day every week
Hardware for test execution	2 computers @ $1,000 ea	
Test software and licenses	2 licenses @ $2,000 ea	

Why am I talking about ROI, under this topic?
because I think that reporting (good or bad) affects directly on the Execution measurement and the maintenance.

Picture this : just before I am going home, I want to execute a set of automation scripts.
I click run , look that everything starts ok, and go home. Almost **zero** cost.
the next day, I need to analyze the results, and find that some tests were failed.
I am opening the report , I am checking, analyze, review and no idea why my set failed.
my boss ask me Why the set failed? You have a report? My answer would **be "sorry, I need to rerun to find out what went wrong".**

So , where is the ROI? It will be only a word.
but if I had put more efforts on the report my answer would be

1. problem A because communication error
2. problem B because wrong input data
3. problem C is a bug on the script (sorry I an a human too)

What I have to do now, is only fix the errors, or reproduce the fault on the AUT. Bottom line the second answer is more professional, is more … automation developer.

In this topic I will try to give you tips about reporting as much as I know.

Report Object

The object used for sending information to the test results.

ReportEvent Method

Description

Reports an event to the test results.

Syntax

```
Reporter.ReportEvent EventStatus, ReportStepName, Details [, Reporter]
```

Arguments

Parameter	Description
EventStaus	Status of the report step: ■ 0 or **micPass**

	Causes the status of this step to be passed and sends the specified message to the report.
	■ **1** or **micFail** Causes the status of this step to be failed and sends the specified message to the report. When this step runs, the test fails. ■ **2** or **micDone** Sends a message to the report without affecting the pass/fail status of the test. ■ **3** or **micWarning** Sends a warning message to the report, but does not cause the test to stop running, and does not affect the pass/fail status of the test. ■ **4** or **micInfo** Sends an information message to the report, does not cause the test to stop running, and does not affect the pass/fail status of the test.
ReportStepName	Name of the intended step in the report (object name).
Details	Description of the report event. The string will be displayed in the step details frame in the report.

Example

```
Reporter.ReportEvent micDone, "micDone", "micDone status message."
Reporter.ReportEvent micFail, "micFail", "micFail status message."
Reporter.ReportEvent micPass, "micPass", "micPass status message."
Reporter.ReportEvent micWarning, "micWarning", "micWarning status message."
Reporter.ReportEvent micInfo, "micInfo", "micInfo status message."
```
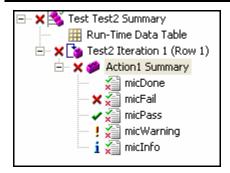

Filter Property

Description

Retrieves or sets the current mode for displaying events in the Test Results. You can use this property to completely disable or enable reporting of steps following the statement, or you can indicate that you only want subsequent failed or failed and warning steps to be included in the report.

Syntax

```
CurrentMode = Reporter.Filter
Reporter.Filter = NewMode
```

Arguments

Value	Description
rfEnableAll	Default. All reported events are displayed in the Test Results.
rfEnableErrorsAndWarnings	Only event with a warning or fail status are displayed in the Test Results.
rfEnableErrorsOnly	Only events with a fail status are displayed in the Test Results.
rfDisableAll	No events are displayed in the Test Results.

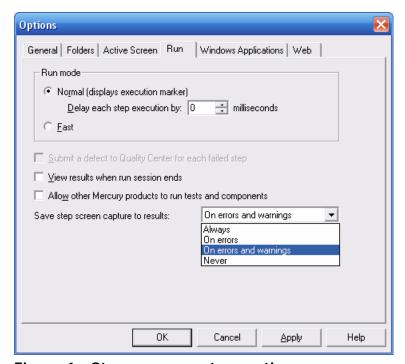## Save image of desktop when error occurs

Step Screen Capture

Figure 1 - Step screen capture option

- Determines when **QuickTest** captures and saves images of the application during the test run to display them in the test results.

The options are:

- Always: Captures images of each step, whether the step fails or passes.
- On errors: Captures images only if the step fails.
- On errors and warnings: Captures images only if the step returns a failed or warning status.
- Never: Never captures images.
- By default, **QuickTest** saves screen captures only on errors. You can instruct **QuickTest** to save screen captures in other cases, as well, by selecting Always or On errors and warnings in the Save step screen capture to results in the Run tab of the Options dialog box.
- Also can be programmatically changed as the following code:

```
Setting( "SnapshotReportMode" ) = CaptureValue
```

Value	Description
0	Always captures images.
1	Captures images only if an error occurs on the page.
2	Captures images if an error or warning occurs on the page.
3	Never captures images.

- It is recommended to use option 2 (on errors and warnings) to take advantage of the **QuickTest** Reporting mechanism.
- Option 0 (always) it is no recommended, especially on large tests, the captured bitmaps are big files and it takes time to load them on the reporter.
- If you have a problematic test, it is recommended to use this option for a while.
- By default, **QuickTest** saves screen captures only on errors. You can instruct **QuickTest** to save screen captures in other cases, as well, by selecting Always or On errors and warnings in the Save step screen capture to results in the Run tab of the Options dialog box.
- Also can be programmatically changed as the following code:

CaptureBitmap Method

Description

The **Number** property Returns or sets a numeric value specifying an error. Number is the **Err** object's default property.

Syntax

```
object.CaptureBitmap FullFileName, OverrideExisting

Desktop.CaptureBitmap
```

Arguments

Parameter	Description
FullFileName	Required. A String value. The full path of the .**png** or .bmp image to save. If you specify a relative path, the path is added to the test results folder.

	Optional. A Boolean value that Indicates whether the captured image
overrideexisting	should be overwritten if the image file already exists in the test results folder.

Remarks

- The object is captured as it appears in the application when the method is performed. If only part of the object is visible, only the visible part is captured.

RegisterUserFunction - ReportImage

Logging

In computerized data logging, a computer program may automatically record events in a certain scope in order to provide an audit trail that can be used to diagnose problems. Examples of physical systems which have logging subsystems include process control systems, and the black box recorders installed in aircraft.

Many operating systems and multitudinous computer programs include some form of logging subsystem. Some operating systems provide a syslog service (described in RFC 3164), which allows the filtering and recording of log messages to be performed by a separate dedicated subsystem, rather than placing the onus on each application to provide its own ad hoc logging system. In many cases, the logs are esoteric and hard to understand; they need to be subjected to log analysis in order to make sense of them. Other servers use a splunk to parse log files in order to facilitate troubleshooting; this approach may yield correlations between seemingly-unrelated events on different servers. Other enterprise class solutions such as those from LogLogic collect log data files in volume and make them available for reporting and real-time analysis.

Audit Trail

An audit trail or audit log is a chronological sequence of audit records, each of which contains evidence directly pertaining to and resulting from the execution of a business process or system function. Audit records typically result from activities such as transactions or communications by individual people, systems, accounts or other entities.

Debugging

You've designed your solution and written the code. You start to run your script with high hopes and excitement, only to be faced with a big ugly gray box telling you that the **VBScript** engine doesn't like what you've done.
So where do you start?

When confronted with a problem, you first need to know the type of error you're looking for. Bugs come in two main flavors:

- Syntax errors
  you may have spelled something incorrectly or made some other typographical or syntactical error. When this happens, usually the program won't run at all.
- Logical errors
  although syntactically correct, your program either doesn't function as you expect or it generates an error message.

Bugs appear at different times, too:

- At compile time

If a compile-time error is encountered, an error message appears as the page is loading. This usually is the result of a syntax error.

- At runtime
  The script loads OK, but the program runs with unexpected results or fails when executing a particular function or subroutine. This can be the result of a syntax error that goes undetected at compile.

Syntax Errors

Ordinarily, objects containing script are compiled, as they are loaded, and are then immediately executed. Errors can occur at either stage of the process. Although the distinction between compile-time and runtime errors is rapidly losing its importance, it is sometimes helpful to know that the entire script compiled successfully and that the error was encountered at a particular point in the script.

Syntax Check

Just a second, before saving or run your script click the Check Syntax in the toolbar or Ctrl+F7, or Tools → Check Syntax.

Note : a bug that still occurs in **QuickTest** version 9.0 is the follows:

```vbscript
Private Const HEXA_VALUE = &H1FF
Const OCT_VALUE = &O774
nHexaNum = &HABC
```

- All of the statements are valid **VBScript** hexadecimal and octal values.
  But the **Check Syntax** feature interprets them as a syntax error.

Type Mismatches Errors

Type mismatches by everyone's favorite data type, the variant. Type mismatches occur when the **VBScript** engine is expecting data of one variant type(like a string), but is actually passed another data type (like an integer.) Type mismatch errors are fairly uncommon in **VBScript/QTP**, since most of the time the variant data type itself takes care of converting data from one type to another. That tends, though, to make type mismatch errors all the more frustrating.
Also, any data read from The **DataTable** is Always **String** Subtype
The Solution : Before a calculation, string print, conditions are sure which data-sub type you are using. Use the **VBScript** conversion and data verification functions (**IsNull**, **IsEmpty**, **CInt**, **CDate**, **VarType** and more) whenever exists a minimal doubt.

Logical Errors

Logical errors are caused by code that is syntactically correct — that is to say, the code itself is legal — but the logic used for the task at hand is flawed in some way. There are two categories of logical errors. One category of errors produces the wrong program results; the other category of errors is more serious, and generates an error message that brings the program to a halt.

Logical errors that affect program results

This type of logical error can be quite hard to track down, because your program will execute from start to finish without failing, only to produce an incorrect result. There are an

infinite number of reasons why this kind of problem can occur, but the cause can be as simple as adding two numbers together when you meant to subtract them. Because this is syntactically correct (how does the scripting engine know that you wanted "-" instead of "+"?), the script executes perfectly.

Logical errors that generate error messages

The fact that an error message is generated helps you pinpoint where an error has occurred. However, there are times when the syntax of the code that generates the error is not the problem.

Out of Range Errors

Subscript Out Of Range is an error that occurs frequently when using arrays. It actually doesn't take much to eliminate this error for good. All you have to do is check the variable value you're about to use to access the array element against the value of the **UBound** function, which lets you know exactly what the maximum subscript of an array is.

Object required Errors

The next most common error is the object required errors. if you try to activate a method, or assign a property value, you'll kill your script stone dead. While it's very easy to generate an object required error in a script, it's also not at all difficult to prevent it.
The Solution : Perform an object reference comparison: **If obj Is Nothing**

On Error Statement

There are two main elements to error handling in **VBScript**. The first is the On Error statement, which informs the **VBScript** engine of your intention to handle errors yourself, rather than to allow the **VBScript** engine to display a typically uninformative error message and halt the program. This is done by inserting a statement like the following at the start of a procedure:

- On Error Goto 0
- On Error Resume Next

On Error statement Enables or disables error handling within a procedure. If you don't use an On Error statement in your procedure, or if you have explicitly switched off error handling, the **VBScript** runtime engine handles the error automatically. First, it displays a dialog containing the standard text of the error message, something many users are likely to find incomprehensible. Second, it terminates the application, so any error that occurs in the procedure produces a fatal runtime error.

- When a runtime error occurs in the routine in which the **On Error Resume Next** statement occurs, program execution continues with the program line following the line that generated the error. This means that, if you want to handle an error, this line following the line that generated the error should call or include an inline error-handling routine.

- When a runtime error occurs in any routine called by the routine in which the **On Error Resume Next** statement occurs, or by its subroutines, program execution returns to the statement immediately after the subroutine call in the routine containing the **On Error Resume Next** statement.

- You disable error handling by using the **On Error Goto 0** statement.

Figure 2 demonstrates an error message generated when trying to open a file that not exists.

Figure 2 - Error Message

If you don't use an **On Error Resume Next** statement anywhere in your code, any run-time error that occurs can cause an error message to be displayed and code execution stopped. However, the host running the code determines the exact behavior. The host can sometimes opt to handle such errors differently. In some cases, the script debugger may be invoked at the point of the error. In still other cases, there may be no apparent indication that any error occurred because the host does not to notify the user. Again, this is purely a function of how the host handles any errors that occur.

On Error Resume Next tells the **VBScript** engine that, should an error occur, you want it to continue executing the program starting with the line of code that directly follows the line in which the error occurred.

A particular **On Error** statement is valid until another **On Error** statement in the line of execution is encountered, or an **On Error Goto 0** statement (which turns off error handling) is executed. This means that if Function A contains an On Error statement, and Function **A** calls Function **B**, but Function B does not contain an On Error statement, the error handling from Function **A** is still valid. Therefore, if an error occurs in Function **B**, it is the On Error statement in Function **A** that handles the error; in other words, when an error is encountered in Function **B**, program flow will immediately jump to the line of code that followed the call to Function **B** in Function A. When Function **A** completes execution, the On Error statement it contains also goes out of scope. This means that, if the routine that called Function **A** did not include an On Error statement, no error handling is in place.

On Error Resume Next Pros and Cons

Lets view the following code sample:
Notice the mistake we made in line 4: instead of typing b = 2, i typed b 2:

```
On Error Resume Next
Dim a
a = 1
b  2
MsgBox a + b
```

The answer you will get is 1! We've managed to turn our fancy new computer into a machine that can't even add 2 and 1!

b 2 is not a valid statement, and when **QTP** encounters this line, an error occurs.
However**, On Error Resume Next** suppresses this error.
QTP then attempts to run the next line of code, something that, in this case, leads
to problems. Because of that, b is not assigned the value 2. In addition, because no value
has been assigned to b, it automatically assumes the value of 0.

Because you know that this answer is wrong, (I hope) and because this script is so short, you probably know that the script needs debugging In this case, then, debugging the script is easy. However, suppose this script was several hundred lines long, and suppose this was a much more complicated mathematical equation. In that case, you never might not know that the answer was wrong; even if you did, you might have trouble tracking down the problem.

Using **On Error Goto 0** statement and re-running now the script will cause the follow message to be displayed:

This time there's no question about whether the script worked or not, and we don't have to puzzle over whether 2 + 1 actually does equal 1. We know for sure that we have problems.

A good article about **On Error** Statements can be found in
http://www.microsoft.com/technet/community/columns/scripts/sgwho.mspx
Article Name: **What's Wrong with My Script?**

Err Object

The **Err** object is part of the **VBScript** language and contains information about the last error to occur. By checking the properties of the **Err** object after a particular piece of code has executed, you can determine whether an error has occurred and, if so, which one. You can then decide what to do about the error — you can, for instance, continue execution regardless of the error, or you can halt execution of the program. The main point is that error handling using **On Error** and the **Err** object puts you in control of errors, rather than allowing an error to take control of the program (and bring it to a grinding halt).

Err Object Properties and Methods

To see how the **Err** object works and how you can use it within an error-handling regimen within your program, let's begin by taking a look at its properties and methods.
Like all object properties, the properties of the **Err** object can be accessed by using the name of the object, **Err**, the dot (or period) delimiter, and the property name. The two methods of the **Err** object allow you to raise or clear an error, while simultaneously changing the values of one or more **Err** object properties.

Err.Number Property

Description

The **Number** property returns or sets a numeric value specifying an error. Number is the **Err** object's default property.

Remarks

- The **Number** property is a Long value that contains an error code value between -2,147,483,648 and 2,147,483,647. (The possibility of a negative error code value seems incongruous but results from the fact that error codes are unsigned long integers, a data type not supported by **VBScript**.)

- **VBScript** itself provides error code values that range from 0 to 65,535. **COM** components, however, often provide values outside of this range. If the value of **Err.Number** is 0, no error has occurred. A line of code like the following, then, can be used to determine if an error has occurred:

```
If Err.Number <> 0 Then
```

Although the properties of the **Err** object provide information on the last error to occur in a script, they do not do so permanently. All the **Err** object properties, including the Number property, are set either to zero or to zero-length strings after an **End Sub**, **End Function**, **Exit Sub**, or **Exit Function** statement.

Tips

- When returning a user-defined error from an Automation object, set **Err.Number** by adding the number you selected as an error code to the constant **vbObjectError**.

Err.Description Property

Description

The **Description** property returns or sets a descriptive string associated with an error.

Remarks

- The **Description** property contains a string that describes the last error that occurred. You can use the **Description** property to build your own message box alerting the user to an error.
- Use this property to alert the user to an error that you can't or don't want to handle. When generating a user-defined error, assign a short description of your error to this property. If **Description** isn't filled in, and the value of **Number** corresponds to a **VBScript** run-time error, the descriptive string associated with the error is returned.
- When a runtime error occurs, the Description property is automatically assigned the standard description of the error.
- If there is no error (that is, if the value of **Err.Number** is 0), the value of the Description property is an empty string.
- For user-defined errors (that is, for errors that you define in your own scripts), you must assign a string expression to the **Description** property or the error won't have an accompanying textual message.
- You can override the standard description by assigning your own description to the **Err** object for both **VBScript** errors and user-defined errors.

Tips

- If your code instantiates an **ActiveX** server, its error codes should be increased by the value of the **VBScript** intrinsic constant **vbObjectError**. When control returns to the local application after an error has been raised by the **OLE** server, the application can determine that the error originated in the **OLE** server and extract the error number with a line of code like:

```vbscript
Dim nError
If (( Err.Number And &HFF00 ) And vbObjectError ) Then
    nError = Err.Number Xor vbObjectError
End If
```

Err.Source Property

Description

The **Source** property returns or sets the name of the object or application that originally generated the error.

Remarks

- The **Source** property contains a string that indicates the class name of the object or

application that generated the error. You can use the **Source** property to provide users with additional information about an error—in particular, about where an error occurred.

- The value of the **Source** property for all errors generated within scripted code is simply "Microsoft VBScript runtime error.

- You can assign a value to the Source property in your own error-handling routines to indicate the name of the function or procedure in which an error occurred.

- If the error occurs in an ActiveX component instantiated by your application, the **Source** property usually contains the class name or the programmatic identifier of the component that raised the error.

Tips

- Use **Source** to provide your users with information when your code is unable to handle an error generated in an accessed object.

- When generating an error from code, Source is your application's programmatic ID.

Err.HelpContext Property

Description

The **HelpContext** property is a read/write property that either sets or returns a long integer value containing the **context ID** of the appropriate topic within a Help file.

Remarks

- The **HelpContext** property can be set either directly or by supplying the fifth parameter (the helpcontext parameter) to the **Err.Raise** method.

- **HelpContext IDs** are decided upon when writing and creating a Windows help file. Once the Help or **HTML** help file has been compiled, the IDs can't be changed. Each **ID** points to a separate Help topic.

Tips

- You can display a topic from a help file by supplying values to the **Err.HelpFile** and **Err.HelpContext** properties, using the **MsgBox** function with the **vbMsgBoxHelpButton** constant and passing **Err.HelpContext** as the **HelpContext** argument

- If you supply a **HelpContext ID** that can't be found in a Windows Help file, the contents page for the Help file should be displayed. However, what actually happens is that a Windows Help error is generated, and a message box is displayed that informs the user to contact their vendor. If you supply a **HelpContextID** that cannot be found in an **HTML** Help file, **VBScript** displays an error message indicating that the Help file is either invalid or corrupted.

Err.HelpFile Property

Description

The **HelpFile** property is a read/write string property that contains the fully qualified path of a Windows Help or HTML Help file.

Remarks

- The **HelpFile** property can be set either directly or by supplying the fourth parameter (the helpfile parameter) to the **Err.Raise** method.

Tips

- Some objects you may use within your application have their own help files, which you can access using **HelpFile** to display highly focused help to your users.
- Remember that once the program encounters an **On Error** statement, all the properties of the **Err** object are reset; this includes HelpFile. You must therefore set the **Err.HelpFile** property each time your application needs to access the help file.

Err.Raise Method

Description

The **Raise** method generates a run-time error.

Syntax

```
object.Raise(number, source, description, helpfile, helpcontext)
```

Arguments

Parameter	Description
number	A Long integer subtype that identifies the nature of the error. **VBScript** errors (both **VBScript**-defined and user-defined errors) are in the range 0–65535.
source	A string expression naming the object or application that originally generated the error. When setting this property for an Automation object, use the form project class. If nothing is specified, the programmatic ID of the current **VBScript** project is used.
description	A string expression describing the error. If unspecified, the value in number is examined. If it can be mapped to a **VBScript** run-time error code, a string provided by **VBScript** is used as description. If there is no **VBScript** error corresponding to number, a generic error message is used.
helpfile	The fully qualified path to the Help file in which help on this error can be found. If unspecified, **VBScript** uses the fully qualified drive, path, and file name of the **VBScript** Help file.
helpcontext	The context ID identifying a topic within *helpfile* that provides help for the error. If omitted, the **VBScript** Help file context ID for the error corresponding to the number property is used, if it exists.

Remarks

- To use the **Raise** method, you must specify an error number.
- At first glance, generating an error within your script may seem like a very odd thing to want to do! However, there are times, particularly when you are creating large, complex scripts, that you need to test the effect a particular error will have on your script.
- The easiest way to do this is to generate the error by using the **Err**.**Raise** method and providing the error code to the *errornumber* parameter, then sit back and note how your error-handling routine copes with the error, what the consequences of the error are, and what side effects the error has.
- If the value of the error code is non-zero, an Alert box opens that displays the error code and its corresponding description.
- All the arguments are optional except number. If you use Raise, however, without specifying some arguments, and the property settings of the Err object contain values that have not been cleared, those values become the values for your error.

Tips

- When setting the *number* property to your own error code in an Automation object, you add your error code number to the constant **vbObjectError**. For example, to generate the error number 1050, assign **vbObjectError** + 1050 to the *number* property.

- The **Raise** method doesn't reinitialize the **Err** object prior to assigning the values you pass in as arguments. This can mean that if you raise an error against an **Err** object that hasn't been cleared since the last error, any properties you don't specify values for still contain the values from the last error.

- As well as using **Raise** in a runtime scenario, you can put it to good use in the development stages of your program to test the viability of your error-handling routines under various circumstances.

- The fact that **Err.Number** accepts only numbers in the range 0-65536 may appear to be strange at first because the data type of the **Error Number** parameter in the **Raise** event is a Long; however, deep in the recesses of the **Err** object, the error code must be declared as an unsigned integer, which is a data type not supported by VBScript.

- When you raise an error in a scripted environment, it may not make sense to supply arguments to the helpfile and helpcontext parameters. in IE applications, the help file itself may not be available on the host computer.

Example

```
Public Function Sum( ByVal nNum2, ByVal nNum2)
    If IsNumeric(nNum1) = False Or IsNumeric(nNum2) = False Then
        On Error Resume Next
        Err.Raise vbObjectError + 100, "Sum Function", _
                "One or more parameters are invalid."
        Exit Function
    End If
    Sum = nNum1 + nNum2
End Function
```

Err.Clear Method

Description

The **Clear** method clears all property settings of the **Err** object.

Syntax

```
object.Clear
```

Remarks

- The **Clear** method clears the information that the **Err** object is storing about the previous error; it takes no parameters. It sets the values of **Err.Number** to 0 and the **Err** object's **Source** and **Description** properties to a null string (**vbNullString**).

- You need to clear the **Err** object only if you need to reference its properties for another error within the same subroutine or before another **On Error Resume Next** statement within the same subroutine.

- Clear method automatically whenever any of the following statements is executed:
  - On Error Resume Next
  - Exit Sub
  - Exit Function