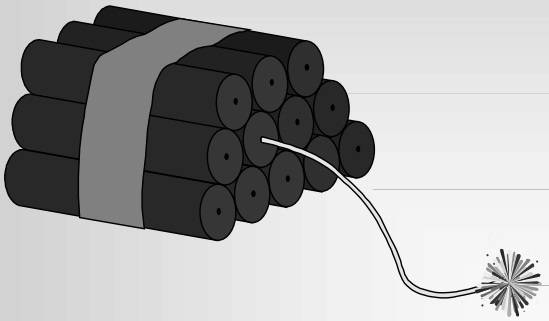
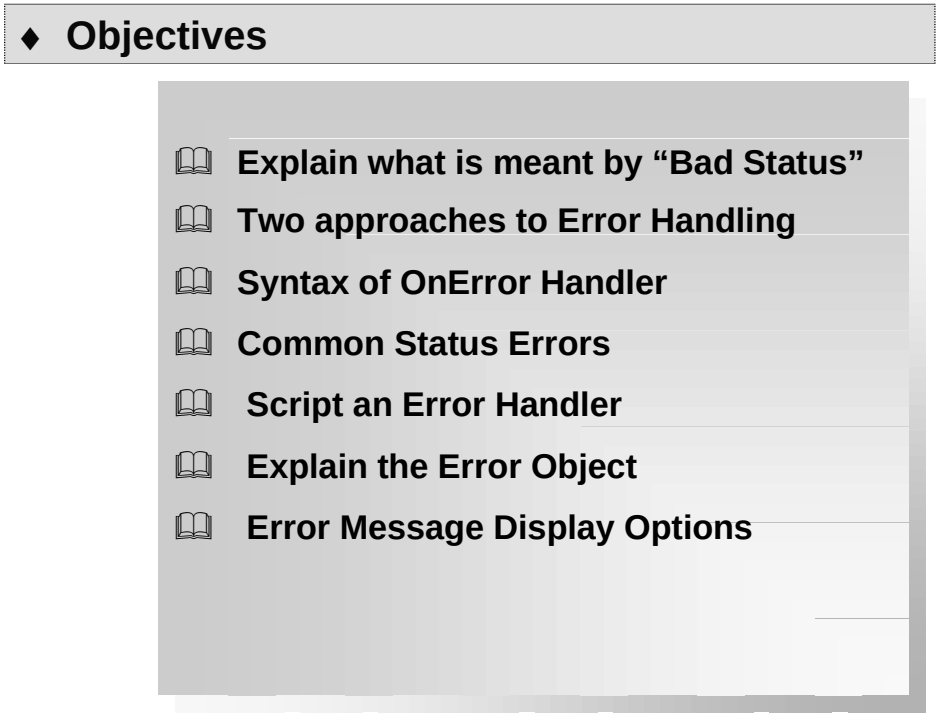
Bad Status Error Handling





Main Idea

The topics covered in this module will help you understand what bad status handling represents and why it should be used when building GUS displays.

Objective

At the end of this module, you will be able to script performant error handlers. To accomplish that goal, the following objectives are covered:

- Explain what is meant by “bad status” (i.e, represents a trappable error).
- Describe the two approaches to error handling:
    - Direct access to parameter status.
    - Bad status as a trappable run time error.
- Explain the format, syntax, and use of the Basic Script OnError handler.
- List the common status errors (i.e, HOPC_(text)_ERROR).
- Script a simple error handler using a single variable, small number of error types, and simple resolution to the error.
- Script a complex error handler using multiple variables, larger number of error types, and complex resolution to the error.
- Explain the Error (Err) object and how it can be used in an Error handler.
- Describe the options in displaying error messages to the operator in terms of:
    - Using a message box to display the trapped error or user defined message.
    - Changing the text object to display symbols (!!, ##, ??, ++, etc) that represent an error instance.
    - Using the Prompt actor to display a message in the Status Bar message area.

What Bad Status Represents

- **What is Bad Status Error Handling?**
- **What is Bad Status?**
- **Why use Bad Status Error Handling?**

What is Bad Status Error Handling?

Runtime errors can stop the execution of your display's scripts. These errors could leave the operator with no clear idea of what went wrong, or what corrective action could be taken. A way to anticipate these errors is to include bad status error handling in your display scripts.

What is Bad Status?

"Status" is a property of a point or tagname. For example, in the following code fragment:

If LCN.FIC100.PV > 100 then ...

PV is a parameter of the point FIC100. The PVparameter also has properties. Status is a property of a parameter value obtained from the LCN. Status indicates two things:

1. Whether the data access is successful
2. Whether the data obtained is reliable or valid

If you wanted to check the status of a data access, you could write the left hand of a script statement similar to this:

If LCN.FIC100.PV.**status** = {some status constant} then

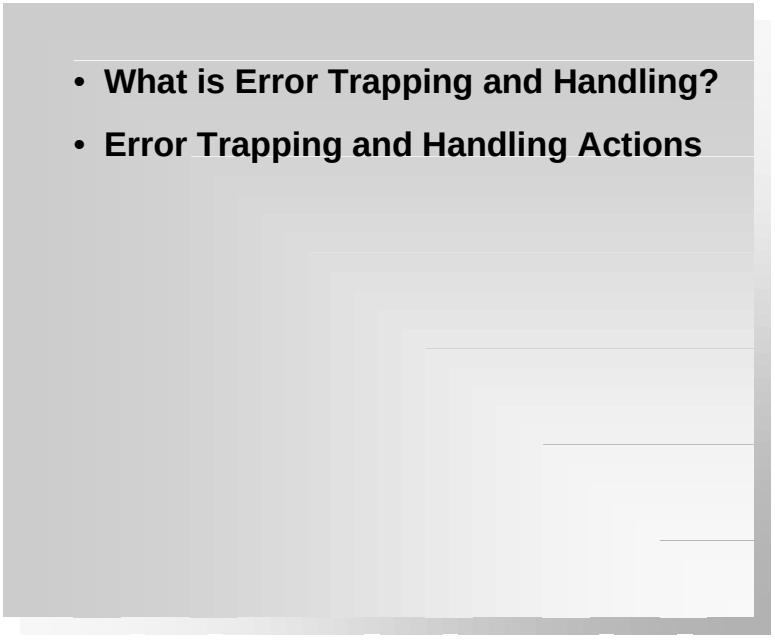
If you were attempting to access the PV parameter and the status indicated that the PV is not valid or compromised (e.g., network communication problem, bad transmitter), the status of the PV parameter is considered "**bad**."

Why use Bad Status Error Handling?

Two reasons for using bad status error handling are:

1. Solid error handling routines make operators more effective. Error handling means that your script can do something, such as take an alternate action or, if necessary, gracefully close the display.
2. Effective error handling can mean increased display performance. Error handling is just good programming practice common to scripting languages.

Error Trapping and Handling



What is Error Trapping and Handling?

Two scripting concepts to become familiar with are error trapping and error handling. *Error trapping* means that your GUS script can intercept and identify an error, then pass program control to script statements that take care of (i.e., "handle") the error. The user-written script statements that take care of the error are considered an embedded script routine, or simply an *error handler*. So, if a program error occurs in one of your display scripts, the program execution branches directly to the handler.

What Can You do with Error Trapping and Handling?

With bad status error trapping and handling you can:

- Give script execution control to an error handler
- Find out what bad status error occurred
- Inform an operator of what possible corrective action to take
- If possible, correct the problem in script
- Resume execution somewhere else in your script
- Ignore the problem
- Provide several script execution paths

Options for Bad Status Handling

- **Two Options**
  1. Directly read parameter status
     Example: IF LCN.FIC100.PV.Status
  2. Use an error handler
     Example: OnError GoTo Trap
- **Error Handlers Are the Better Choice**

Two Options for Handling Bad Status

In your display's script(s), you have two options for dealing with parameters that have bad status. You can

- Directly read the parameter status (e.g., If LCN.FIC100.PV.Status)
- Use an error handler (e.g., OnError GOTO)

Error Handlers are the Better Choice

You can use either approach depending upon your display's needs. These approaches, along with their tradeoffs, are described next. However, most users prefer using Error handlers because they represent a more performant option.

Option 1: Read Parameter Status

- **Access Status Property**
- **Honeywell Provides Common Status Errors**
  Example: HOPC_STORE_ERROR
- **Slow Option in Non-OnDataChange Scripts**

Option 1: Directly Read the Parameter Status

Bad status can be handled by directly accessing the status property of a parameter. For example, to obtain the status of FIC100.PV, code in your script FIC100.PV.STATUS.

Reference: The data-type of the status property is a long integer. Refer to Common Status Errors in the Built-in constant section for status values to compare with actual status values.

The following code fragment shows an example of access to parameter status in a script associated with a text object:

```
if lcn.FIC100.pv.status = HOPC_NO_ERROR then
          me.text = lcn.FIC100.pv
else
          me.text = "??????"
end if
```

Note: Directly reading the parameter status is a slower option than an OnError handler, especially when it is used in non-OnDataChange scripts.

Options 2: Trap Bad Status with Handler

- **Use an OnError Handler**
  - Traps bad status as runtime error
  - Handler is simply labeled code
- **Bad Status Always Raises Trappable Error**

Option 2: Use an Error Handler to trap Bad Status as a Runtime Error

The GUS scripting language provides a way for handling runtime error conditions through the use of error trapping and error handlers. An error handler is set up by using the On Error statement. This statement activates and deactivates an on error handler. The error handler itself is simply a labeled block of code in the body of the subroutine in which the on error statement occurs.

Reference: For more details, refer to On Error (statement) in the Basic Script Reference or on-line Help.

The following example shows an OnDataChange subroutine containing an error handler:

```
Sub OnDataChange()
       On Error Goto Error_Handler
       me.text = dispdb.ent01.pv
Exit Sub
       Error_Handler:
       select case err.number
              case HOPC_COMMUNICATION_ERROR
                     me.text = "====="
              case HOPC_CONFIGURATION_ERROR
                     me.text = "#####"
              case HOPC_VALUE_ERROR
                     me.text = "-----"
              case HOPC_STORE_ERROR
                     me.text = "+++++"
              case else
                     Msgbox Err.Description + " Error Number is " +
                                          CStr(Err.Number)
       end select
End Sub
```

Summary: Bad status always raises a trappable error

A reference to the value property of a parameter with bad status will always raise a trappable runtime error. You can avoid the error by first checking the status property before accessing the value property, or you can program for the occurrence of the trappable error by placing an error handler in the subroutine.

Error handling is Faster Option

Note: Using an error handler is a faster option rather than directly reading the parameter status especially in non-OnDataChange scripts.

How to Build Error Handler

- **Overview**
  - OnError statement
  - Exit Sub statement before handler
  - Handler code typically near end of procedure
  - Can use Err.number
  - Can use decision structure (If..then, select case)
  - Exit handler
- **Example Syntax**

Error Handling: Quick Overview of How to Build

Error handling in general requires several parts:

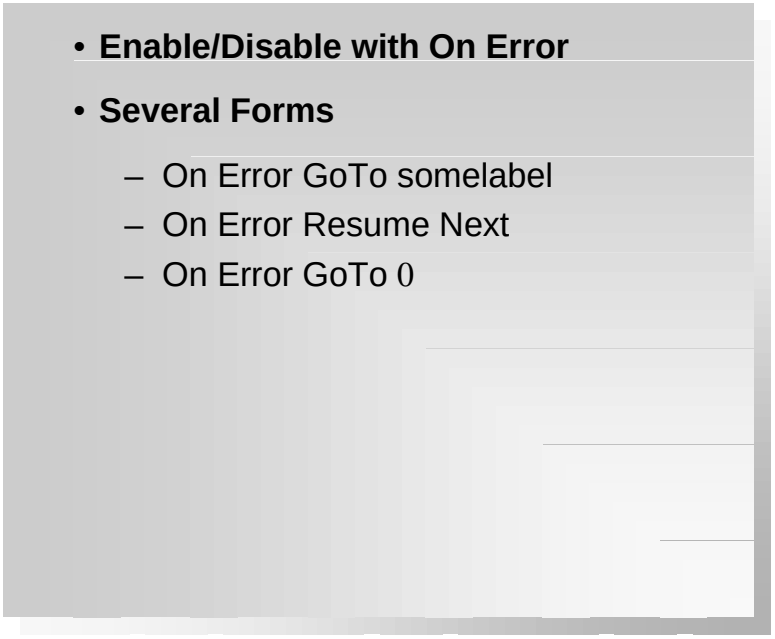
- An On Error statement as one of your first statements to enable error trapping and branch to handler (e.g., GoTo *label*).
- Exit sub statement before the destination label.
- The handler itself typically coded near or at the end of a procedure.
- The handler itself can use the error object's Err.Number property.
- A decision structure can be within the error handler (If..then, Select Case) to display the error message and any other necessary information.
- A way to leave the handler and, if necessary, reset the error (e.g., Resume statement).

These topics are described in the following slides.

Example Syntax

```
Sub OnDataChange()
            On Error Goto Error_Handler
            {user code entered here}

Exit Sub    'Always provide Exit before trap

            Error_Handler:
            select case err.number
                        case
                        case
                        case
            end select
End Sub
```

Enabling/Disabling Error Handling



Enabling/Disabling Error Handlers

To enable/disable error trapping, use the On Error instruction. There are several ways (i.e., forms) to use an On Error instruction.

On Error GoTo somelabel	Somelabel refers to a label in the same procedure where the Error Handler begins. This is a common approach in error handling where you take control of what should happen next.
On Error Resume Next	Ignores the error handler and continues program execution uninterrupted. Useful for debugging along with Err function. The Err function is used to find out if error occurred and what the error was.
On Error GoTo 0	GoTo 0 actually turns off error trapping. Use "Goto" 0 to turn off error handling that had been previously turned on. Subsequent errors are handled by GUS Basic. In other words, if an error occurs after an OnError GoTo 0 statement the error halts script execution. (Note that this is the default behavior if no error handling is enabled in the script.)

Note: GoTo Line# is not supported in GUS Basic Script.

Exit or Return From Error Handler

- **Include a Way to Leave Handler**
- **Several Forms**
  - Resume
  - Resume 0
  - Resume Next
  - Resume Label
  - GoTo Label
  - End…Exit...

Exit or Return from an Error Handler

Generally an error handler also includes a way to leave it. There are several ways to leave an error handler and re-enter your normal code.

Resume	Resume returns to the same line that caused the error and attempts to continue execution. In desktop operations, Resume normally is used to give the user another chance to fix the problem and try the task again (e.g., bad floppy). In process operations, you have to anticipate that the problem may not be immediately fixed. Recommendation: Do not use resume for bad status handling because there is a chance that you will find yourself in an infinite loop.
Resume 0	Same as Resume
Resume Next	Returns back to the line immediately following the line that caused the error.
Resume *label*	Returns to the line indicated by the label.
GoTo *label*	Branches to the label within the procedure.
End Sub, Exit Sub, End Function, Exit Function	End terminates execution of current script; exit sub exits current subroutine.

Behavior Summary and Review

- **Summary of On Error Behavior**
  - No error trapping occurs until an OnError executes
  - Error handling is local to procedure
  - Can have more than one error handler
- **Questions**

Summary of On Error behavior

No error trapping can occur until an On Error GoTo label instruction executes.

An error handler is local to a subroutine (i.e., cannot branch to an error handler outside of the procedure.)

You can have more than one error handler in a subroutine. Error handlers can be nested.

Questions

Question: Assume that a procedure makes a call to another procedure after an error handler is enabled with an On Error statement. What will happen if program control is in another procedure and an error occurs in the first procedure?_____
_____

Question: If you get stuck in an error loop and cannot seem to get out, what keystrokes let you out?
_____

Question: If you wanted to prevent an error infinite loop, what could you program into the error handler to limit the retries?_____
_____

Resetting the Error State

- **Several Ways to Reset an Error**
  - OnError or Resume
  - ERR.NUMBER = -1
  - ERR.CLEAR method
  - Exit…End...

Resetting the Error state

Error handlers have a life local to the procedure in which they are defined. There are several ways to reset the error state. After you reset the error state, your procedure can then trap newly generated errors. The error is reset when any of the following conditions occurs:

- An On Error or Resume statement is encountered.
- When Err.Number is set to -1.
- When the Err.Clear method is called.
- When an Exit Sub, Exit Function, End Function, End Sub is encountered.

The properties Err.Clear and Err.Number refer to an error object, the topic of our next discussion.

Error Object

- **The Error Object is**
  - Created when OnError handler enabled
  - Used to identify error
  - An object with properties and methods
- **Additional Details: Err Object, Reference Manual**

The Error Object

After an On Error handler is enabled, behind the scenes an error object, Err, is created. The Err object is useful because you cannot fix the error if you do not know what the error is. The Err object itself has properties that are set when an error is generated. The Err object also provides methods that, for example, allow you to clear the error. The methods and properties are the following:

- Err.Number (property)
- Err (statement)
- Err.Clear (method)
- Err.Description (property)
- Err.HelpContext (property)
- Err.HelpFile (property)
- Err.LastDLLError (property)
- Err.Raise (method)
- Err.Source (property)

The discussion that follows describes several commonly used properties and methods - - Err.Number, Err.Description, Err.Clear, Err.Raise. Additional details about the Err object can be found in the Scripting Reference Manual.

Classify and Identify Error

- **Two Considerations**
  - Classify error
  - Identify process variable in error
- **Classifying an Error**
  - Err.number
  - Err.description

Scripting a Bad Status Error Handler

When scripting a bad status error handler, two considerations are:

- Classification of the error
- Identification of the process variable that is in error

Classification of an error

Classification of an error can have at least two levels

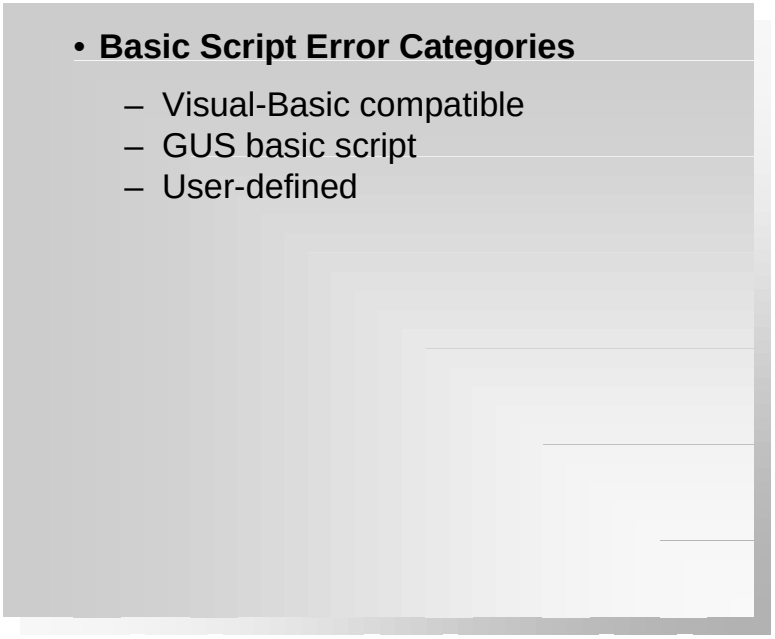
- Err.number
- Err.description

As you saw from the earlier example, the Error object, Err, has a property that identifies the error number. The err.number property can become part of your decision making code (e.g., select case err.number).

The Error numbers for Data Access problems are the following:

ERR.NUMBER	Description	Sample cause
5	Invalid Procedure Call	Invalid date input from user
6	Overflow	calculated value outside range for variable
11	Division by Zero	denominator in calculation = 0
13	Type Mismatch	Variable input from user doesn't match intended data type
1051	COMMUNICATION	Data owner not accessible
1052	CONFIGURATION	Invalid Entity or Variable input from user
1053	VALUE_ERROR	Enumeration.Internal specifies invalid member number
1054	STORE_ERROR	Data Owner in the process of initializing
1056	DYNAMIC_INDEX	Array Index not resolved
1057	NO_ERROR_NO_DATA	Enumeration.Internal specifies invalid set number
1058	varied by cause	Store block by LCN validation/security handling
-214352566	OLE Automation Error	Integer Overflow (int01 = 123456)

Categories of Errors



Note: There are more errors than shown here. BasicScript errors fall into three categories:

1. Visual-Basic compatible errors: These errors, numbered between 0 and 799, are numbered and named according to the errors supported by Visual Basic.
2. GUS BasicScript errors: These errors, numbered from 800 to 999, are unique to BasicScript.
3. User-defined errors: These errors, equal to or greater than 1,000, are available for use by Honeywell extensions or by the script itself.

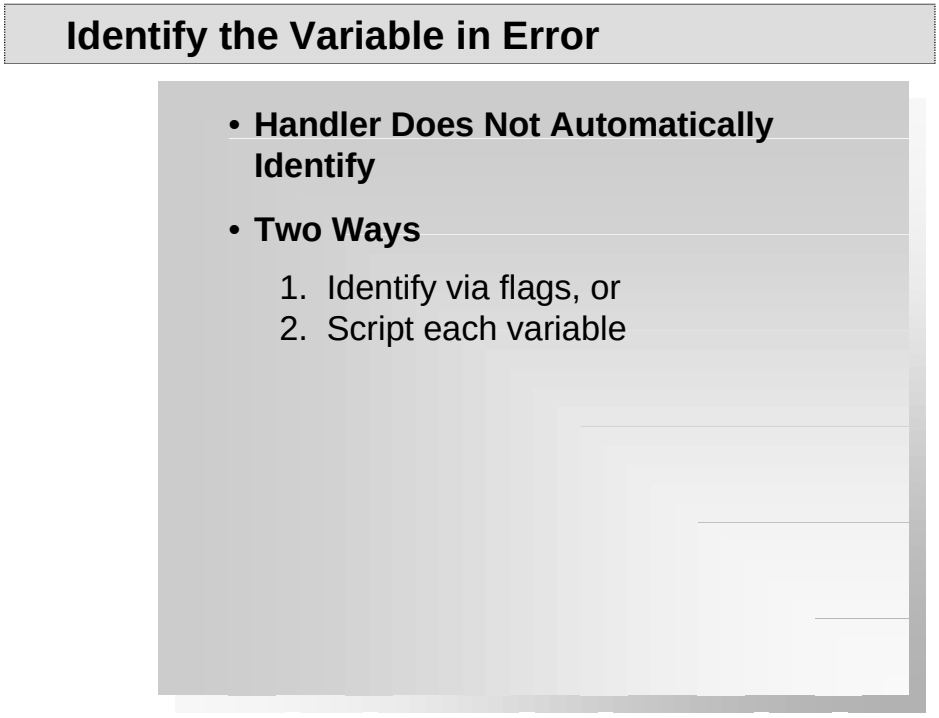
(As implied by item 3, you can define your own errors. That topic will be covered later with the Err.Raise method.)

The error number can be displayed to the operator in a message box. For example, the statement:

Msgbox “Error’” & **Err.number** & “’has occurred”

The above example is only going to show the number of the error. To accomplish a second level of error classification would mean including the textual error description and a user defined flag. For example, the following statement now provides a textual description of the error:

Msgbox “Error’” & **Err.number** & “’has occurred” & **Err.description**



Identify the Variable for an Error Handler

A bad status error handler does not automatically identify the variable in error. Two ways to accomplish this are to

- Identify the variable via user defined flags in script, or
- Script an error handler for each variable

Displaying Status Error Message

- **Display Status in Several Ways**
  - Message box
  - Defined area
  - Prompt actor

Displaying the Bad Status Error Message to the Operator

Messages to the operator can be displayed in several ways:

- Message box (MSGBOX)
- Defined message area in the display (text object)
- Use the Prompt actor to display text in status bar message area

Warning: The message box, MSGBOX, is a modal dialog. Its usage should be avoided in OnData Change scripts because it will hold up thread execution until it is cleared.

Approaches to Messaging

- **Approaches Depend Upon Script**
  - In OnDataChange scripts, keep thread and display running. Use non-modal forms
  - In User-Event scripts, alert the operator
  - OnError often first line of code

Typical Approaches to Bad Status Error Message to the Operator

Bad status error handling frequently appears in OnDataChange and user event (i.e., LButtonUp, LButtonClick) scripts. There are several approaches to bad status error handling, but the following "rules of the road" are recommended:

- The intent of bad status error handling in OnDataChange scripts is to keep the script execution thread running and the display from closing. So, you will often include bad status handling in these scripts, even if it is represents nothing more than a branch to subroutine exit. Also, if a message is sent to an operator, it is through the use of some non-modal form. In other words, modal dialog boxes (msgbox, askbox, input box, etc) are avoided in bad status error handling code within OnDataChange scripts because they hold up thread execution until an operator responds to the dialog box. Instead, visual forms (e.g., object changes color), text strings (me.text = "some string") are used.

- The intent of bad status error handling in user event scripts is to alert the operator of an incorrect entry or to request confirmation of an operator entry. If a bad status message is sent to an operator, it can be through the use of some modal form. In other words, a modal dialog box (msgbox, askbox, input box, etc) can be used here; the most typical dialog used is the msgbox. Visual forms (e.g., object changes color), text strings (me.text = "some string"), are also used. Non-modal viewports can be used in place of the message box if screen real estate permits.

- When bad status error handling is coded into OnDataChange or user event scripts, it frequently appears as the first line of code following the sub statement.

Error Handling Examples (OnDataChange)

- **Examples from OnDataChange Scripts**
  - Show error visually with color change ME.FILLCOLOR=TDC_BLUE
  - Show error visually with fill change ME.FILLPATTERN=HS_DIAGCROSS

Examples of Bad Status Error Message in OnDataChange

Bad status error handling examples that appear in OnDataChange scripts follow. The first examples show errors visually.

Example 1

```
Sub OnDataChange
On error goto Trap
‘user specific code appears here
Trap:
             Me.LineColor = TDC_MAGENTA
End Sub

Sub OnDataChange
On error goto Trap
‘user specific code appears here
Trap:
             Me.FillColor = TDC_BLUE    'Bar Fill color is Blue
             Me.LineColor = TDC_BLUE    'Bar Line color is blue
End Sub
```

Example 2

```
Sub OnDataChange()
             on error goto badstatus
‘user specific code appears here
exit sub
             badstatus:
                          me.fillpercent=100.0
                          me.filldirection=fill_LR
                          me.fillpattern=hs_diagcross
                          me.fillcolor=makecolor(191,0,126)
                          me.linecolor=makecolor(191,0,126)
End Sub
```

OnDataChange: Keep Display Open

- **Use an Error Handler to Keep Display Open**
  - Branch to end of sub
    ODC_Error:
    End Sub
  - Intent: Keep display open

The next example branches to the end of the subroutine. (Its intent is to keep the display from closing when an error occurs.)

```
Sub OnDataChange()
 On Error Goto ODC_Error

'user specific code appears here

ODC_Error:
End Sub
```

Error Status Text Messages

- **Provide text messages for operator**
- **Examples**

  E_Msg.text = “Error” & Err & “-” & Error$

  text55.visible = true (text 55 displays message) me.text = “---”

The following examples branch to the error handler and provide the operator messages via text objects.

Example 1

```
Sub OnDataChange()
             On Error Goto ODC_Error
‘user specific code appears here
Exit Sub
             ODC_Error:
                          E_Msg.visible = true
                          E_Msg.text = "Error "  & Err & " - " & Error$
End Sub
```

Example 2

```
Sub OnDataChange()
on error goto badstatus
text55.visible = false
‘user specific code appears here
Exit Sub
badstatus:
             text55.visible = true
end Sub
```

‘Text55 text defined as: COMMUNICATION ERROR CHECK SYSTEM STATUS ‘DISPLAY. CALLUP xxxDISPLAY AGAIN

Example 3

```
Sub OnDataChange()
             dim almstat as string
             On Error Goto ODC_error
‘user specific code appears here
             Exit Sub
             ODC_error:
                          me.text = "---”
                          me.fillcolor = makecolor(226,0,255)
                          me.textcolor = makecolor(0,0,0)
End Sub
```

User Event Error Examples

- **User Event Errors Can Appear Textually**
  - MsgBox is modal
  - Message also uses Err Object, function

The following examples display an error textually in a message box.

Example 1

```
Sub OnLButtonUp()
             On Error Goto B1_error
‘user specific code appears here
Exit Sub
B1_error:
             INFO.fillcolor = tdc_red ‘change button fillcolor
             MsgBox "Error '" & Err & " - " & Error$ & "'"
End Sub
```

Example 2

```
Sub OnLButtonUp()
             ON ERROR GOTO BADSTATUS
‘user specific code appears here
Exit Sub
             BADSTATUS:
                          BEEP
                          MSGBOX ERROR$,ebExclamation,"Error"
                          Text55.fillcolor = makecolor(153,153,153)
                          ‘make any other color changes
 End Sub
```

Example 3

```
Sub OnLButtonUp()
             on error goto errorhandler
‘user specific code appears here
exit sub
errorhandler:
             beep
             msgbox Err.Description, ebinformation, "your string"
End Sub
```

User Event Error Examples, continued

- **Show Errors as Text Object Text**
  E_msg.text=“Error” &Err& “-” & Error$
- **Use Non-modal Viewport**
  Viewport.Open

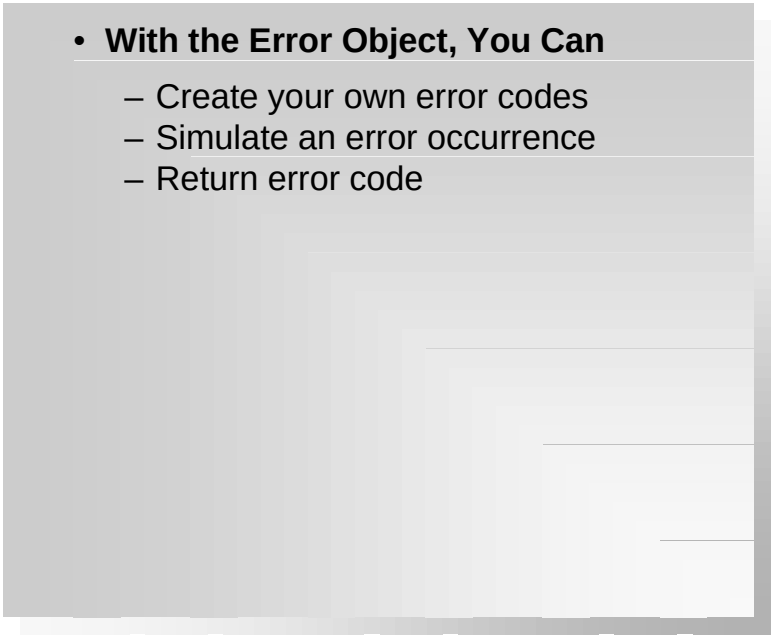
The first example shows an error displayed textually in a text object.

```
Sub OnLButtonUp()
    On Error Goto Info_Error

‘user specific code appears here

Exit Sub
    Info_Error:
              E_Msg.visible = true
              E_Msg.text = "Error "  & Err & " - " & Error$
End Sub
```

An example of the viewport follows.

```
Sub OnLButtonUp()
    On Error Goto INFO_error
‘user specific code appears here

Exit Sub
    INFO_error:
              Viewport.Open "Target Information",0,0,600,200
              Print  "\pe(point).\p(param)  " & "  " & text2.text
              ‘more user specific code appears here
End Sub
```

Err Object Options



What You Can Do with the Err object

It might get a little confusing here because there is the Err statement, Err function, and Err object's default property of Number. They all look like this: Err.

With the Err object and related functions and statements, you can

- create your own error codes defined only in your script.
- simulate an error occurrence, which is useful if you want to try to test your script for special conditions.
- return the code of the error that enabled the error handler

Error Statement, Function, Method Forms

- **Several Error Forms are Provided**
  - Err (object)
  - Err (function)
  - Err (statement)
  - Error (statement)
  - Error (function)
  - Err.Raise
  - Err.Clear

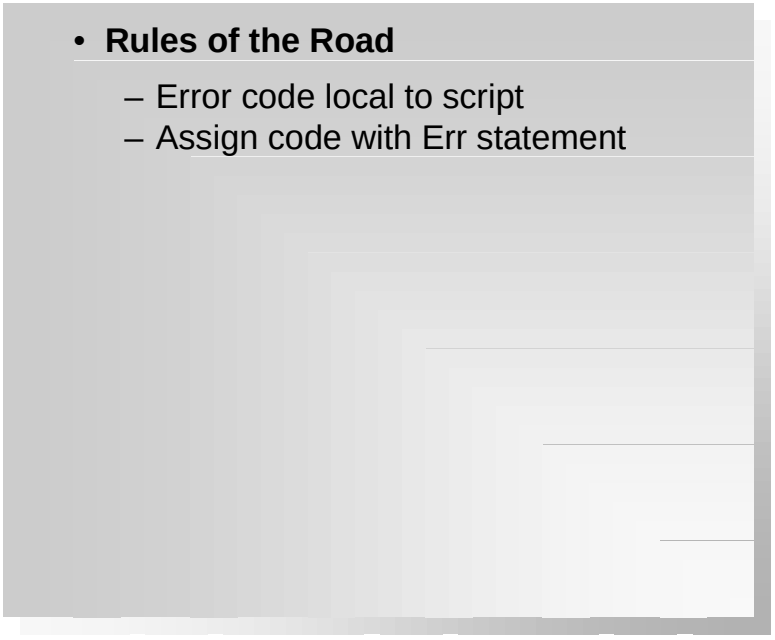
Forms of Error Statements, Functions, and Methods

Before discussing how to raise an error, first review several forms of error statements, functions, and methods.

Script form	Comment
Err (object)	An object with properties (such as description, number, and source) and methods (such as raise and clear).
Err (function)	Returns an error code
Err (statement)	Sets a value for an error code returned by the Err function to a specified integer
Error (statement)	Simulates or forces an error. Basic script recommends use of Error.Raise.
Error, Error$ (function)	Returns a string variant (Error)or string (Error$) that represents the text of an error message. Similar to Err.Description.
Err.Raise (method of Err object)	Generates a runtime error. Preferred over Error statement because several properties can be set in one statement.
Err.Clear (method of Err object)	Clears the properties of the Err object.

Note: The ErrL function is useful only for debugging.

Creating an Error Code



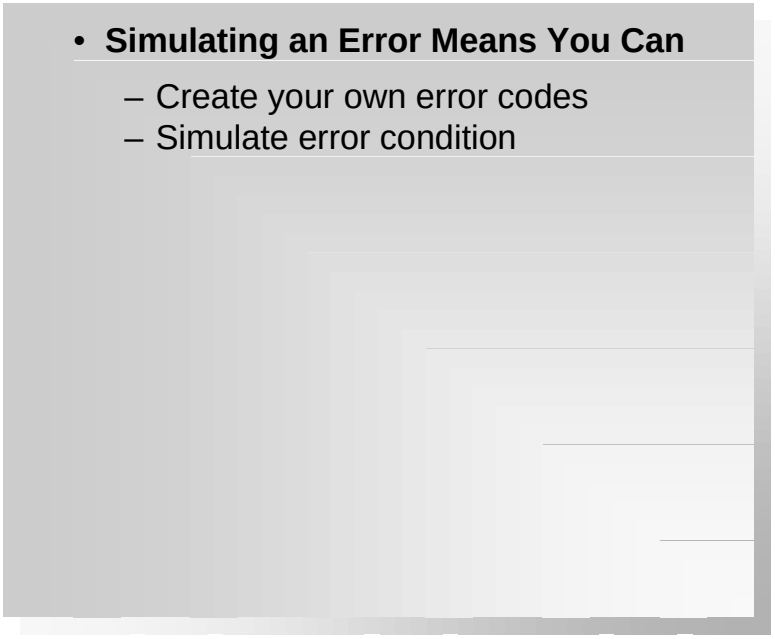
Err Statement Creation of Error Code

The Err statement can be used to create an error code available only for that script's execution. In effect, you are setting or assigning a value that can be returned by the err function. If you review the list of error codes in the Basic Script reference, you will see that there are gaps in the error codes. Numbers that are not used are available for your use to define an error code that is unique to your display's script. For example, error code 99 is not predefined. You could have a timed entry in a non-critical display for an operator. If the operator does not respond in 5 seconds, you can assign error code 99 in the following manner:

Err = 99 'timed elapsed for operator entry

Then, in a subsequent error handler, use the Err function to determine which error occurred.

Simulate an Error



Error Statement, Err.Raise Method Simulation of Error

You can also use the Error statement or Err.Raise method to simulate an error occurrence. This is useful for two different reasons:

1. You can create your own user defined error codes (just like you can with the Err statement). Just use error codes not already pre-defined. When you branch to your error handler, use the Err function to test or trap the error code. User defined error codes provide a way for you to test for special conditions.
2. Additionally, the Error statement and Err.Raise method allow you to simulate an error condition. If the error code that you assign is pre-defined by Basic Script, then it is as if the actual runtime error had occurred. The simulation of error codes is useful to test your display's script and see whether or not your error handlers work properly.

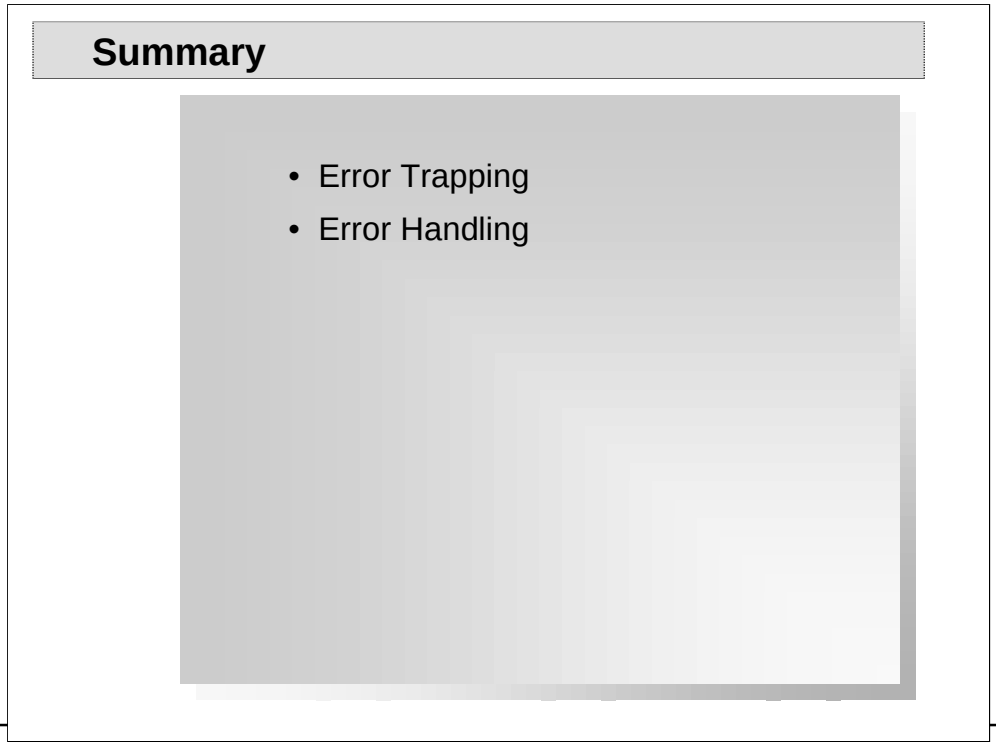
Err.Raise Method Simulation Example

As mentioned earlier, the use of Err.Raise is preferred because you can simulate the error, set the error code, description in one statement. One approach of Err.Raise is to check the validity of the operator entry for a non-process application. Note that while you can do this, it is

For example:

```
Sub OnLButtonClick()
        Dim x As Variant
        On Error Goto TRAP
        x = InputBox("Enter a target amount:")
        If Not IsNumeric (x) Then          'test if number
            Err.Raise 99,,"The target must be a number entry"
        End If
        'put code here
        Exit Sub
TRAP:
        MsgBox Err.Description
End Sub
```

Attention: The input box and message box are modal in this example.



Summary: When an error occurs within a subroutine, BasicScript checks for an On Error handler within the currently executing subroutine or function. Once an **error trap** has gained control, appropriate action should be taken.

BasicScript supports nested error handlers. If a runtime error occurs and no On Error handler is defined within the currently executing procedure, then BasicScript returns, if applicable, to the calling procedure and looks there for an **error handler** to execute. This process repeats until a procedure is found that contains an error handler or until there are no more procedures. Once an error handler has control, it should address the error condition and resume execution. If an error is not trapped or if an error occurs within the error handler, then BasicScript displays an error message, halting execution of the script. (Reminder: Error handlers have a life local to the procedure in which they are defined.)

Error handling requires an **On Error** statement to trap the error and branch to a handler; an **Exit Sub** statement before the error label allowing program execution to branch to the handler; code written in the error handle that addresses the error condition; and a way to leave the handler (e.g. resume script execution or end the subroutine).