Kofax Transformation Modules Script Logging Framework

Application Note

Date	June 25, 2012
Applies To	Kofax Transformation Modules (KTM) 5.0 and 5.5
Summary	This application note provides a framework of script functions to use for logging information from Kofax Transformation Modules (KTM) script.
Revision	1.0

Kofax Transformation Modules (KTM) Script Logging Framework Overview

The extensive customization possible through Kofax Transformation Modules (KTM) scripting can lead to projects with a large amount of custom code. For purposes of maintenance of the project and being able to clearly distinguish KTM behavior from custom script behavior, it is essential to write organized script code. Along with clear comments and other best practices, implementing some type of logging helps diagnose problems and can aid in maintaining the project. This application note provides a framework of functions recommended for logging from KTM script and explanations of the benefits of these functions.

Log File Location

When choosing a log file location, consider that it must be accessible from all of the machines and user accounts running each of the KTM modules, both interactive and automatic. Because of this, the best choices are likely to be locations to which KTM modules already writes its own log files. There are two locations where KTM modules write log files:

1. [Batch Image Folder]\Log
   This log location is located alongside the images for the batch, where we will use the naming convention [Batch ID]_KTM_Script_Batch.log. Interactive KTM modules and some Capture modules already write their logs to this location. When the batch is released, these logs are deleted. This has the benefit of allowing us to log a generous amount of information here and not worry that logs will take too much space or become too large to analyze. But for this reason, we want to make sure that any errors or other notable events are *also* logged to a permanent location. Additionally, there is no Batch Image Folder during design in Project Builder.

2. C:\ProgramData\Kofax\Capture\Local\Logs

   -or-

   C:\Documents and Settings\All Users\Application Data\Kofax\Capture\Local\Logs

   KTM Server writes daily logs to this location and so we will do the same using the naming convention YYYMMDD_KTM_Script_Log.log. This folder exists on each machine and we will use it to log errors and other information important enough that we want it to persist even if a batch releases. During design in Project Builder, we will log both local *and batch log* messages to YYYMMDD_KTM_Script_Design.log

Additionally while running during design, we will use Debug.Print to log any messages to the Immediate Window of the script editor.

Logging Metadata

Every message logged with these logging functions includes useful information. Consider the following example:

```
7/15/2011 5:11:00 PM (61859.86) -- MachineName -- 0000000A -- WindowsUser, Capture User Name
(CaptureUserID) – F1\D1\P1 (1.xdc,1.tif) -- Server 1 -- [Project|Logging_BatchOpen#226]
Message
```

Much like a news article, we want our log to answer “Who, What, When, Where?” If a problem occurs, we will be able to use this information to help determine the “How?” and the “Why?”

1. Who: WindowsUser, Capture User Name (CaptureUserID)
   These properties are only exposed in KTM 5.5, so they will not be logged in KTM 5.0. If User Profiles is not enabled, then only WindowsUser will be logged.
2. What: Message
   This is the message that we are logging.
3. When: Date Time (Seconds/Hundredths of Seconds)
   Date and time are standard in any log, but seconds/hundredths allow for quickly seeing duration between logged events.
4. Where:
   a. MachineName
      The name of the system where the code is running. Because all of the statements sent to the local log will be on the same machine, the machine name is only logged in the batch log.
   b. 0000000A
      The hex ID of the batch being processed. Because a batch log will always be about the same batch, the batch ID is only logged to the local log. The local log may show multiple batches processing concurrently, so it is important that the batch ID be present in each line.
   c. F1\D1\P1 (1.xdc,1.tif)
      This is the Folder/Document/Page identification. The example above indicates that we are dealing with the first page in the first document in the first folder, with 1.xdc storing the document and 1.tif storing the page. The root folder is always present and always stored in Folder.xfd, so that information is omitted.
   d. Server 1
      The module and instance.
   e. [Project|Logging_BatchOpen#226]
      This indicates the class, function, and line of script from which the message is being logged. The class can be an important piece of context because some script events can be defined at multiple levels in a class hierarchy.

The primary benefit of using a logging framework that includes this metadata by default is that it frees us from having to manually include these common pieces of information. While troubleshooting a For Loop without this framework we might log a message like “Function X, Document #, Page #, Beginning of Loop”, whereas with the framework, we could just log “Beginning of Loop” and know that the other data and more will already be included in the log.

Logging Functions

The functions of the logging framework are provided in full at the end of this document and are ready to be copy-pasted into a project. The following sections briefly describe each function and in some cases explain the rationale for how they are written. The only requirement is to follow the instructions in the Required Functions section. The section on Useful Functions explains the functions that we intend to use to log messages throughout our project. The Supporting Functions are used by the Useful Functions and are generally not used directly.

Required Functions

The only absolute requirement is to call Logging_InitializeBatch as described below, but it is also strongly recommended to use Logging_BatchOpen and Logging_BatchClose as described in this section.

Logging_InitializeBatch

The only absolute requirement to using this framework of logging functions is that the function Logging_InitializeBatch is called from the Application_InitializeBatch event. This will initialize all of the batch-specific information including the location that we will use for our batch log.

It is imperative that it is called specifically from the Application_InitializeBatch event rather than one of the other early events. Application_InitializeScript does not have access to batch-specific information and Batch_Open is only fired once per batch even if the batch was opened by multiple extraction processes which each needed to be initialized. The required initialization should look like this:

```
Private Sub Application_InitializeBatch(ByVal pXRootFolder As CASCADELib.CscXFolder)
   Logging_InitializeBatch(pXRootFolder)
End Sub
```

This should come before any other code in the Application_InitializeBatch event, because any logging which is done before this initialization, including any logging in the Application_InitializeScript event, will not have the benefit of batch-specific information. This means that, while the local log will work normally, the batch log location will not have been initialized and will fall back to the system's temp directory. This will not cause any problems, but is just a limitation to keep in mind.

Logging_BatchOpen

Logging_BatchOpen should be called at the beginning of the Batch_Open event like so:

```
Private Sub Batch_Open(ByVal pXRootFolder As CASCADELib.CscXFolder)
   Logging_BatchOpen(pXRootFolder)
End Sub
```

This will log a message like the following to both the batch and local logs:

```
Batch 789/00000315: Batch Class "Routing" (published 7/18/2011 9:51:57 AM, project saved
7/18/2011 9:51:45 AM)
```

This clearly establishes not only the batch and batch class, but also the dates that the batch class and KTM Project used by the batch were last modified. This can prevent confusion if a batch being analyzed was created with a previous version of a batch class. Seeing different dates written to the same batch log can also establish if the "Update Batch Class" feature was used on the batch.

Logging_BatchClose

Logging_BatchClose should be called at the end of the Batch_Close event like so:

```
Private Sub Batch_Close(ByVal pXRootFolder As CASCADELib.CscXFolder, ByVal CloseMode As
CASCADELib.CscBatchCloseMode)
   Logging_BatchClose(pXRootFolder,CloseMode)
End Sub
```

If the batch is closing in Error mode, then the batch will be checked for rejected documents with the function Logging_RejectedDocs to log details of any rejected documents. See that function’s description for more details.

If the batch is closing in Error mode, Suspend mode, or Final (“normal”) mode, then the batch will be checked by Logging_Routing to log details of documents or folders that will be routed. See that function’s description for more details.

If the batch is closing in Child mode, created from routing, this function will log the tag that the routing group used.

Useful Functions

There are three functions that encompass the majority of the functionality of this framework and that we should use throughout our code. ScriptLog is the main logging function and benefits from the metadata discussed in earlier in this document. ErrorLog is used to log Err information and stack trace directly from the Err object. Finally, MsgBoxLog is a wrapper and drop-in replacement for the MsgBox function which not only logs the message, but also logs the user's response and suppresses the dialog in non-interactive contexts.

ScriptLog

ScriptLog is the function we will use to log messages with the benefits of all the metadata described in the Logging Metadata section. Here is the function definition:

```
Public Sub ScriptLog(ByVal msg As String, Optional AddToLocalLog As Boolean=False, _
   Optional ByVal pXDoc As CscXDocument=Nothing, Optional ByVal pXFolder As CscXFolder=Nothing, _
   Optional PageNum As Integer=0, Optional ExtraDepth As Integer=0)
```

msg - The only required parameter is the message that you would like to log.

AddToLocalLog - The way this logging framework is constructed is that *all* messages are logged to the batch log and then this optional parameter specifies if a particular message should also be logged to the local log. Thus, you can think of the batch log as a verbose log and the local log as the error/important message log. For clarity, it is recommended to specify this parameter using the defined constants LOCAL_LOG or BATCH_LOG instead of literal True/False values.

pXDoc
pXFolder
PageNum – These optional parameters are passed to Logging_IdentifyFolderDocPage so we can log identifying information about the Folder/Doc/Page. See the description of that function in the Supporting Functions section for more details. Many functions or loops within functions in KTM script operate on a particular Folder, Document, or Page. By providing that context in our log, it will be great assistance in allowing us to trace messages or errors to specific documents or files.

ExtraDepth – This changes the context of the function on the stack from which we are logging. This is only useful if we are writing other logging related functions. For example, our ErrorLog function adds extra depth so that in the log, instead of saying it is logging a message from "ErrorLog", it will say it is logging from the function that called ErrorLog. If this parameter is to be used, it is recommended for the sake of clarity to use the defined constant IGNORE_CURRENT_FUNCTION instead of a literal integer.

ErrorLog

ErrorLog allows us to log errors in a more organized fashion than simply logging the string from Err.Description. It also logs a stack trace and shows a message box when used from interactive modules. Here is the function definition:

```
Public Sub ErrorLog(ByVal E As ErrObject, Optional ByVal ExtraInfo As String="", _
   Optional ByVal pXDoc As CscXDocument=Nothing, Optional ByVal pXFolder As CscXFolder=Nothing, _
   Optional PageNum As Integer=0, Optional ForceError As Boolean=False, _
   Optional SuppressMsgBox As Boolean=False)
```

E – This parameter should always be called by passing the "Err" object which already exists as part of WinWrap's error handling. We will let the ErrorLog function itself check if there is actually an error to log, which reduces the amount of code that goes into error handlers that are set in every function. For example, without having to manually check if there is an error or placing an Exit Sub above the error handler, this block of code would still only log an error if one occurs:

```
Public Sub TestErrorHandler()
   On Error GoTo catch
   'Code that might error
   catch:
   ErrorLog(Err) 'Only logs if there is an error
End Sub
```

From the Err object we will log the error number and description. Notably, the Err object does not tell us what line on which the error occurred, so it is important to be aware that the top line of our stack trace just references the line that calls the ErrorLog function.

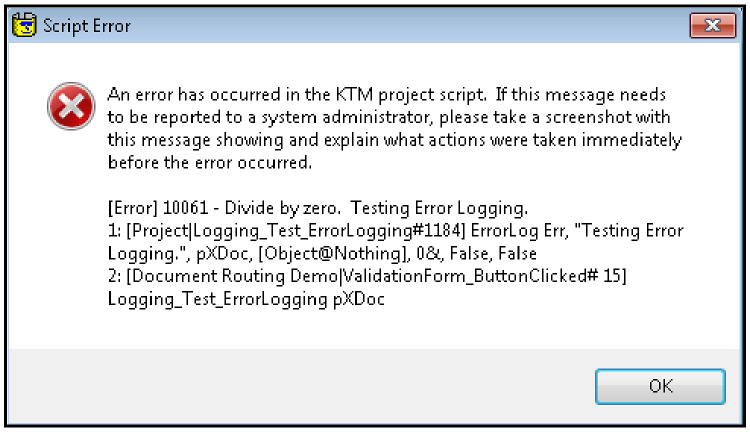
ExtraInfo – This parameter allows you to pass information in addition to the information already built into the Err object. When including extra information via this parameter, be mindful of the metadata which will already be included so that you do not spend effort including duplicate information. This function is useful in combination with the ForceError parameter.

pXDoc
pXFolder
PageNum – These optional parameters are passed to ScriptLog and on to Logging_IdentifyFolderDocPage so we can log identifying information about the Folder/Doc/Page. See the description of those functions for more details.

ForceError – The normal operation of this function is to only log an error if one is reflected in the Err object, but we can force it to log an error with this parameter. If an error is forced using this parameter then we should make sure to put something informative in the ExtraInfo parameter. For clarity it is recommended to use the defined constants FORCE_ERROR or ONLY_ON_ERROR instead of literal True/False values.

SuppressMsgBox – When the ErrorLog function is used from interactive modules that are not thin clients, it will show a message box with the error, stack trace, and instructions about reporting the error to an

Administrator if needed. The constant USER_ERROR_MSG defines the instructions to the user and defaults to the message seen in the screenshot below. Though only an English message is currently defined, the basic structure in the function is already built to accommodate localized messages. If we do not want our error to show a message box in interactive modules, we can pass the constant SUPPRESS_MSGBOX.



An error has occurred in the KTM project script. If this message needs to be reported to a system administrator, please take a screenshot with this message showing and explain what actions were taken immediately before the error occurred.

```
[Error] 10061 - Divide by zero. Testing Error Logging.
1: [Project|Logging_Test_ErrorLogging#1184] ErrorLog Err, "Testing Error
Logging.", pXDoc, [Object@Nothing], 0&, False, False
2: [Document Routing Demo|ValidationForm_ButtonClicked# 15]
Logging_Test_ErrorLogging pXDoc
```

MsgBoxLog

MsgBoxLog is a wrapper and drop-in replacement for the MsgBox function which not only logs the message, but also logs the user's response and suppresses the dialog in non-interactive contexts. Here is the function definition:

```
Public Function MsgBoxLog(ByVal Message As String, Optional ByVal MsgType As VbMsgBoxStyle, _
   Optional ByVal Title As String, Optional ByVal pXDoc As CscXDocument=Nothing, _
   Optional ByVal pXFolder As CscXFolder=Nothing,
   Optional PageNum As Integer=0) As VbMsgBoxResult
```

Message
MsgType
Title
MsgBoxLog – Because this is a drop-in replacement for WinWrap's MsgBox function, these three parameters, and the return value of the function, are exactly the same.

pXDoc
pXFolder
PageNum – These optional parameters are passed to ScriptLog and on to Logging_IdentifyFolderDocPage so we can identify information about the Folder/Doc/Page. See the description of those functions for more details.

Trying to display a MsgBox during KTM Server or while running a thin client may halt the module without a user able to click the message. As such, this function will suppress the dialog and return a vbOK result instead of waiting for user input. Notably, if the thin client is enabled for a particular module, but we open a batch in the rich client, this will cause the MsgBox to be suppressed. See the explanation of the supporting function, Logging_ExecutionModeString, for details.

An example of what might be logged from this function is as follows:

```
MsgBox: User clicked Yes for message "This message, MsgBox style, and user choice will be
logged." (vbOkOnly, vbCritical, vbDefaultButton1, vbApplicationModal)
```

Supporting Functions

These supporting functions are used by the primary functions in the framework and are not generally called directly, but familiarity with how they work may still be useful.

Logging_CaptureLocalLogs

Logging_CaptureLocalLogs determines the local logging path, creates it if needed, and then caches the path for future calls. If there is any problem, we will fall back to the system's temp path.

Logging_ExecutionModeString

Logging_ExecutionModeString is used to return a string describing what module the script is being run from. There is not currently a way to tell if a script is executing in a rich or thin client. Because of this, we check if the thin client is enabled in the project for a particular module. Then if code is running from that module, we have to assume that it is running in the thin client. This appends "(TC)" to the module name and sets the global Boolean variable THIN_CLIENT to True so that other functions can adjust behavior accordingly.

Logging_IdentifyFolderDocPage

Logging_IdentifyFolderDocPage will identify a Folder/Document/Page in terms of both structure and files. For example "F1\D1\P1 (1.xdc,1.tif)" indicates that we are dealing with the first page in the first document in the first folder, with 1.xdc storing the document and 1.tif storing the page. The root folder is always present and always stored in Folder.xfd, so that information is omitted. The parameter combinations we can use with this function are just a folder, just a document (from which we determine the folder), or a document and a page number. If either the document or the folder are not being included, the parameter can be passed with the keyword "Nothing". If the page is not being included, the parameter can be passed as 0. The page number is one-based as opposed to many PageIndex properties in KTM which are zero-based.

Logging_RejectedDocs

Logging_RejectedDocs is a recursive function that goes into a folder and checks its documents and pages to see if any have been rejected. This function is called from Logging_BatchClose when a batch is closed in error and if any rejected documents are found it will log a message like this:

```
[Error]   The following have been rejected:
D1\P1 (1.xdc\1.tif): Test page rejection note.
D2 (2.xdc): Test document rejection note.
```

Logging_RoutingFolder

Logging_RoutingFolder is a recursive function that goes into folders and checks for folders or documents that have XValues set such that they will be routed. Routing folders was introduced in KTM 5.5. Only first level folders under the root folder can be routed, and doing so implies routing all of its contents. The information collected by this function is used by Logging_Routing.

Logging_Routing

Logging_Routing uses Logging_RoutingFolder to gather information about documents and folders being routed. Using this information, it will log information useful in understanding what will be routed and where. This accounts for features in KTM 5.0 such as routing specific documents or the parent batch to a separate queue. It also accounts for features introduced in KTM 5.5 such as routing folders, routing to a new batch class, or naming the newly created batch. Notably, if all documents in the batch are being routed, it logs a message to the local log noting that this batch will be deleted because all of its contents have been routed. This function is called from Logging_BatchClose and if any routed folders or documents are found it will log a message like this:

```
Routing group (Test, Queue=KTM.Validation3): D1 (1.xdc)
```

Logging_SheetClass

Logging_SheetClass works on the return value of WinWrap's CallersLine function which, according to WinWrap documentation, has the format "[macroname|subname#linenum] linetext". The problem is that the "macroname" or "sheet" of script is just a number when we run CallersLine in KTM. This isn't very useful to us, so this function matches the number to the Class/FolderClass and returns the name. A notable aspect of doing this is that we can distinguish between events with the same name defined in different classes.

Logging_StackLine

Logging_StackLine combines the class name from Logging_SheetClass with the function name and line number information from WinWrap's CallersLine function.

Logging_StackTrace

Logging_StackTrace returns a stack trace by calling Logging_StackLine with successively deeper levels of WinWrap's CallersLine function. It is called by ErrorLog to generate stack traces where errors occur.

Logging Framework Code

The code in this section is ready to copy-paste into a project. It should then be initialized as described in the Required Functions section and used according to the Useful Functions section. For an easy way to test features of the framework, see the Testing the Logging Framework section.

```
'CONFIGURABLE LOGGING CONSTANTS

'This message will be prepended to the msgbox shown to a user if there is a script error
Public Const USER_ERROR_MSG As String = "An error has occurred in the KTM project script.  " & _
   "If this message needs to be reported to a system administrator, please take a " & _
   "screenshot with this message showing and explain what actions were taken immediately " & _
   "before the error occurred."
Public Const LOG_FILENAME As String = "_KTM_Script_Log.log"
Public Const BATCH_LOG_FILENAME As String = "_KTM_Script_Batch.log"
Public Const DESIGN_LOG_FILENAME As String = "_KTM_Script_Design.log"

'for readability messages are logged on a newline after metadata, set this to true to log to a
single line
Public Const LOG_SINGLE_LINE As Boolean = False


'NONCONFIGURABLE LOGGING CONSTANTS
Public Const LOCAL_LOG As Boolean = True
Public Const BATCH_LOG As Boolean = False
Public Const IGNORE_CURRENT_FUNCTION As Integer = 1
Public Const FORCE_ERROR As Boolean = True
Public Const ONLY_ON_ERROR As Boolean = False
Public Const SUPPRESS_MSGBOX As Boolean = True
```

```
'GLOBAL LOGGING VARIABLES
'This will be changed to true if it looks like we are in a Thin Client module
Public THIN_CLIENT As Boolean

Public BATCH_IMAGE_LOGS As String
Public CAPTURE_LOCAL_LOGS As String

Public BATCH_CLASS As String
Public BATCH_NAME As String
Public BATCH_ID As Long
Public BATCH_ID_HEX As String

'to support KTM 5.5 features
Public BATCH_USERID As String
Public BATCH_USERNAME As String
Public BATCH_WINDOWSUSERNAME As String
Public BATCH_USERSTRING As String 'combination of the previous three




'========  LOGGING CODE =======

'Initialize Capture/runtime info.  Call from Application_InitializeBatch
Public Sub Logging_InitializeBatch(ByVal pXRootFolder As CscXFolder)
   On Error GoTo CouldNotCreate

   'assume the batch log folder does not exist
   Dim FolderExists As Boolean
   FolderExists = False

   'these items are only set by Capture at runtime.  if any are set,
   ' then we are at runtime and they are all set
   If pXRootFolder.XValues.ItemExists("AC_BATCH_CLASS_NAME") Then
      'Set batchname, batchid, batch class
      BATCH_CLASS = pXRootFolder.XValues.ItemByName("AC_BATCH_CLASS_NAME").Value
      BATCH_NAME = pXRootFolder.XValues.ItemByName("AC_BATCH_NAME").Value
      BATCH_ID = CLng(pXRootFolder.XValues.ItemByName("AC_EXTERNAL_BATCHID").Value)
      BATCH_ID_HEX = Hex(BATCH_ID)

      'pad hex ID
      BATCH_ID_HEX = Right("00000000", 8 - Len(BATCH_ID_HEX)) & BATCH_ID_HEX

      'These items are only present in KTM 5.5
      If pXRootFolder.XValues.ItemExists("AC_BATCH_WINDOWSUSERNAME") Then
         BATCH_WINDOWSUSERNAME=pXRootFolder.XValues.ItemByName("AC_BATCH_WINDOWSUSERNAME").Value
         BATCH_USERID = pXRootFolder.XValues.ItemByName("AC_BATCH_USERID").Value
         BATCH_USERNAME = pXRootFolder.XValues.ItemByName("AC_BATCH_USERNAME").Value

         'if user profiles are off these will all be the same
         If BATCH_WINDOWSUSERNAME = BATCH_USERID And BATCH_USERID = BATCH_USERNAME Then
            'user profiles is off so only use the windows user
            BATCH_USERSTRING = BATCH_WINDOWSUSERNAME
         Else
            'user profiles is on, so use all
            BATCH_USERSTRING = BATCH_WINDOWSUSERNAME & ", " & BATCH_USERNAME & _
               " (" & BATCH_USERID & ") -- "
         End If
      End If


      'set the batch logging path
      BATCH_IMAGE_LOGS = pXRootFolder.XValues.ItemByName("AC_IMAGE_DIRECTORY").Value & _
         "\" & BATCH_ID_HEX & "\Log\"
      On Error GoTo FolderNotExist
      'try to write to the intended log file
      Open BATCH_IMAGE_LOGS & BATCH_ID_HEX & BATCH_LOG_FILENAME For Append As #1
      Close #1

      'if it does not error, then the folder exists
      FolderExists = True

      'if writing to the file causes an error then FolderExists is still false
FolderNotExist:
      On Error GoTo CouldNotCreate

      If Not FolderExists Then
         'try to create the folder
         MkDir(BATCH_IMAGE_LOGS)

         'if it does not error, then the folder exists
         FolderExists = True
      End If

      'if creating the folder causes an error then FolderExists is still false
CouldNotCreate:
      Err.Clear()
      On Error GoTo catch

      'if the folder still doesn't exist after trying to create, just use image path
      If Not FolderExists Then
         'we prefer to log to the "Log" folder along with the interactive modules,
         '  but if there is a problem, use the image path itself
         BATCH_IMAGE_LOGS = pXRootFolder.XValues.ItemByName("AC_IMAGE_DIRECTORY").Value & _
            "\" & BATCH_ID_HEX & "\"
      End If
   End If

   catch:
End Sub

'log initial information about the batch.  Call from Batch_Open
Public Sub Logging_BatchOpen(ByVal pXRootFolder As CscXFolder)
   On Error GoTo catch

   'the project file is copied on publish and retains its original modified date
   Dim ProjectLastSave As Date
   ProjectLastSave = FileDateTime(Project.FileName)

   'We can only get the batch class publish date if "Copy project during publish" is used
   '  otherwise we will just get the project path
   Dim BatchClassPublishOrProjectPath As String

   'if the "Copy project during publish" is checked, it will be located within PubTypes\Custom
   If InStr(1, Project.FileName, "PubTypes\Custom") > 0 Then
      'with "Copy project during publish" the folder containing the project is created
      '   (thus dated) while publishing
      Dim ProjectFolder As String
      ProjectFolder = Mid(Project.FileName, 1, InStrRev(Project.FileName, "\") - 1)

      BatchClassPublishOrProjectPath = "published " & CStr(FileDateTime(ProjectFolder))
   Else
      'without "copy project during publish" the folder could have any date
      '  and the project could be anywhere, so just get the project path
      BatchClassPublishOrProjectPath = Project.FileName
   End If

   'log basics like batch name, class, id, machine name, project save date
   ScriptLog("Opening Batch """ & BATCH_NAME & """ (" & BATCH_ID & "/" & _
      BATCH_ID_HEX & ") -- " & BATCH_USERSTRING & Environ("ComputerName") & vbNewLine & _
      "Batch " & BATCH_ID & "/" & BATCH_ID_HEX & ": Batch Class """ & BATCH_CLASS & _
      """ (" & BatchClassPublishOrProjectPath & ", project saved " & CStr(ProjectLastSave) _
      & ")", LOCAL_LOG)

   Exit Sub

   'if there is an error, log it and try to keep going
   catch:
   ErrorLog(Err, "", Nothing, pXRootFolder, 0, ONLY_ON_ERROR, SUPPRESS_MSGBOX)
   Resume Next
End Sub



'Find and return the Capture\Local\Logs directory
'   caching the result to global variable CAPTURE_LOCAL_LOGS
Public Function Logging_CaptureLocalLogs() As String

   'If CAPTURE_LOCAL_LOGS is already set, just return it
   If CAPTURE_LOCAL_LOGS <> "" Then
      Logging_CaptureLocalLogs = CAPTURE_LOCAL_LOGS
      Exit Function

   Else
      'Environment variable AllUsersProfile defaults to
      '  C:\Documents and Settings\All Users in XP/2003
      '  C:\ProgramData in Vista/2008/7
      '  But we actually want (notice the additional "Application Data" for XP):
      '  C:\Documents and Settings\All Users\Application Data\Kofax\Capture\Local\Logs
      '  C:\ProgramData\Kofax\Capture\Local\Logs

      Dim FileName As String
      FileName = Format(Now(), "yyyymmdd") & LOG_FILENAME

      Dim VistaPath As String
      VistaPath = Environ("AllUsersProfile") & "\Kofax\Capture\Local\Logs\"

      'XP Path should also work in Vista/2008/7 because there are
      '   junction folders for compatibility
      '  But we would rather get the real path to eliminate confusion, so check VistaPath first
      Dim XPPath As String
      XPPath = Environ("AllUsersProfile") & "\Application Data\Kofax\Capture\Local\Logs\"

      'If the VistaPath does not exist, writing to it will error, goto label NotVista,
      '   then check the XPPath
      On Error GoTo NotVista
      Open VistaPath & FileName For Append As #1
      Close #1
      'no error means it is the Vista path
      CAPTURE_LOCAL_LOGS = VistaPath
      Logging_CaptureLocalLogs = CAPTURE_LOCAL_LOGS
      Exit Function

NotVista:
      CAPTURE_LOCAL_LOGS = XPPath
      Logging_CaptureLocalLogs = CAPTURE_LOCAL_LOGS

      On Error GoTo CreateFolder
      Open XPPath & FileName For Append As #1
      Close #1
      'No error means it is the XP path
      Exit Function

CreateFolder:
      'Neither folder exists, try to create the XPPath

      'assume the batch log folder does not exist
      Dim FolderExists As Boolean
      FolderExists = False

      On Error GoTo CouldNotCreate

      If Not FolderExists Then
         'try to create the folder
         MkDir(CAPTURE_LOCAL_LOGS)

         'if it does not error, then the folder exists
         FolderExists = True
      End If

      'if creating the folder causes an error then FolderExists is still false
CouldNotCreate:
      Err.Clear()
      On Error GoTo catch

      'if the folder still doesn't exist after trying to create, just use image path
      If Not FolderExists Then
         'if there is a problem getting the Capture local logs folder,
         '   log to the system temp folder
         CAPTURE_LOCAL_LOGS = Environ("Temp") & "\"
         Logging_CaptureLocalLogs = CAPTURE_LOCAL_LOGS
      End If
   End If

   catch:
End Function

'primary logging function
Public Sub ScriptLog(ByVal msg As String, Optional ByVal AddToLocalLog As Boolean = False, _
   Optional ByVal pXDoc As CscXDocument = Nothing, _
   Optional ByVal pXFolder As CscXFolder = Nothing, _
   Optional ByVal PageNum As Integer = 0, _
   Optional ByVal ExtraDepth As Integer = 0)

   On Error GoTo catch

   'for readability messages are logged on a newline after metadata,
   '   but this is a matter of preference
   If Not LOG_SINGLE_LINE Then
      msg = Replace(vbNewLine & msg, vbNewLine, vbNewLine & vbTab)
   End If

   'refer to the WinWrap documentation for "CallersLine Function" for
   '   an explaination of the Depth parameter
   Dim Caller As String
   Caller = CallersLine(ExtraDepth)

   'In addition to the date/time, Timer makes it easy to see the number
   '  of seconds/hundredths of seconds between events
   '  output of Timer is also padded for readability in the log
   Dim DateString As String
   DateString = Now & " (" & Format(Timer, "00000.00") & ") -- "

   'if we have an folder/xdoc/page (optional) we can log which document for context
   Dim WhichDoc As String
   WhichDoc = Logging_IdentifyFolderDocPage(pXFolder, pXDoc, PageNum)

   If WhichDoc <> "" Then
      WhichDoc = WhichDoc & " -- "
   End If

   'current module, class/function/line logged from
   Dim ModuleAndFunction As String
   ModuleAndFunction = Logging_ExecutionModeString() & " -- " & Logging_StackLine(Caller) & " "
   'check if we are in design or runtime
   If Project.ScriptExecutionMode = CscScriptModeServerDesign Or _
      Project.ScriptExecutionMode = CscScriptModeValidationDesign Or _
      Project.ScriptExecutionMode = CscScriptModeVerificationDesign Then

      'if we are in project builder, there is no "image directory" so just write to local logs
      '  Because Application_InitializeBatch is only called manually in PB,
      '  use Logging_CaptureLocalLogs() directly to ensure the path is set
      Open Logging_CaptureLocalLogs() & Format(Now(), "yyyymmdd") & DESIGN_LOG_FILENAME _
         For Append As #1
         Print #1, DateString & WhichDoc & ModuleAndFunction & msg
      Close #1

      'also print a shorter message to the intermediate pane of the script window
      Debug.Print(Format(Timer, "00000.00") & " " & WhichDoc & ModuleAndFunction & msg)
   Else
      'In case logging is attempted without or before initialization
      If BATCH_IMAGE_LOGS = "" Then
         BATCH_IMAGE_LOGS = Environ("Temp") & "\"
         BATCH_ID_HEX = "Unknown"
      End If

      'During runtime, always log to the batch log
      Open BATCH_IMAGE_LOGS & BATCH_ID_HEX & BATCH_LOG_FILENAME For Append As #1
      'batches are processed from various machines, so each line in the batch log
      '   should specify the machine
         Print #1, DateString & Environ("ComputerName") & " -- "  & BATCH_USERSTRING & _
            WhichDoc & ModuleAndFunction & msg
      Close #1

      'And add to local log if specified
      If AddToLocalLog Then
         Open Logging_CaptureLocalLogs() & Format(Now(), "yyyymmdd") & LOG_FILENAME _
            For Append As #1
         'the machine may be processing different batches/modules/users concurrently,
         '  so each line in the local log should have a batch id, userstring
            Print #1, DateString & BATCH_ID_HEX & " -- "  & BATCH_USERSTRING & _
               WhichDoc & ModuleAndFunction & msg
         Close #1
      End If
   End If

   catch:
End Sub


'Primary error logging function.  First parameter should always be "Err".
Public Sub ErrorLog(ByVal E As ErrObject, Optional ByVal ExtraInfo As String = "", _
   Optional ByVal pXDoc As CscXDocument = Nothing, _
   Optional ByVal pXFolder As CscXFolder = Nothing, _
   Optional ByVal PageNum As Integer = 0, _
   Optional ByVal ForceError As Boolean = False, _
   Optional ByVal SuppressMsgBox As Boolean = False)

   'checking if there is an error here means it does not need to be
   '   checked before the function is called
   If E = 0 And ForceError = False Then
      Exit Sub
   End If

   'ErrorMessage will be displayed To user In interactive modules
   Dim ErrorMessage As String
   ErrorMessage = "[Error] "

   If E > 0 Then
      ErrorMessage = ErrorMessage & E.Number & " - " & E.Description
   End If
   If ExtraInfo <> "" Then
      ErrorMessage = ErrorMessage & "   " & ExtraInfo
   End If


   'when the error handler is set it clears the error, so we must finish with the e param first
   E.Clear()
   On Error GoTo catch


   'get stack trace
   Dim Stack As String

   '1 extra depth to ignore this current function in the stack
   Stack = Logging_StackTrace(IGNORE_CURRENT_FUNCTION)

   'Add stack trace to the error message
   ErrorMessage = ErrorMessage & vbNewLine & Stack

   'log the error and stacktrace
   ScriptLog(ErrorMessage, LOCAL_LOG, pXDoc, pXFolder, PageNum, IGNORE_CURRENT_FUNCTION)


   'Display to user if not in Server or thin client
   If Project.ScriptExecutionMode <> CscScriptModeServer And Not THIN_CLIENT And _
      Not SuppressMsgBox Then
      'if the message needs to be localized, other languages can be added
      '   as seen in the script help topic:
      'Script Samples | Displaying Translated Error Messages For a Script Validation Method
      Dim LocalizedMessage As String

      Select Case Application.UILanguage
         Case "en-US"  'American English
            LocalizedMessage = USER_ERROR_MSG
         Case Else
            LocalizedMessage = USER_ERROR_MSG
      End Select

      'include info about where we are
      Dim WhichDoc As String
      WhichDoc = Logging_IdentifyFolderDocPage(pXFolder, pXDoc, PageNum)

      'include batch info if it exists
      If BATCH_ID_HEX <> "" Then
         WhichDoc = BATCH_NAME & " (" & BATCH_ID_HEX & "), Batch Class: " & BATCH_CLASS _
            & vbNewLine & WhichDoc
      End If

      MsgBox(LocalizedMessage & vbNewLine & vbNewLine & WhichDoc & vbNewLine & vbNewLine & _
         ErrorMessage, vbCritical, "Script Error")
   End If

   catch:
End Sub


'returns a stacktrace from where ever it is called
Public Function Logging_StackTrace(Optional ByVal ExtraDepth As Integer = 0) As String
   On Error GoTo catch

   Dim i As Integer
   i = ExtraDepth

   Dim CurrentStackLine As String
   CurrentStackLine = CallersLine(i)
   'as long as CallersLine returns something, stacktrace continues
   While CurrentStackLine <> ""
      'get a nicer format for the stack line
      CurrentStackLine = i & ": " & Logging_StackLine(CurrentStackLine) & _
         Mid(CurrentStackLine, InStr(1, CurrentStackLine, "]") + 1)

      'Add current line to the stack trace
      Logging_StackTrace = Logging_StackTrace & CurrentStackLine & vbNewLine

      'increment and try to get the next line (CallersLine returns blank if none)
      i = i + 1
      CurrentStackLine = CallersLine(i)

      'protect against trying to log a large stack
      If i > 10 Then
         Logging_StackTrace = Logging_StackTrace & i & ": ...Stack continues beyond " & _
            i - 1 & " frames..."
         Exit While
      End If
   Wend

   'on error exit
   catch:
End Function


'Returns string from ScriptExecutionMode Enum to indicate which module is running
Public Function Logging_ExecutionModeString() As String
   On Error GoTo catch

   'There is not currently a way to tell if a script is executing in a rich or thin client
   '  This is important because MsgBox cannot be used if we are in a thin client
   '  If a thin client is enabled for the project and we are in that module,
   '  we must assume it is a thin client
   THIN_CLIENT = False

   Select Case Project.ScriptExecutionMode
      Case CscScriptModeServer
         Logging_ExecutionModeString = "Server " & Project.ScriptExecutionInstance
      Case CscScriptModeServerDesign
         Logging_ExecutionModeString = "ServerDesign " & Project.ScriptExecutionInstance
      Case CscScriptModeUnknown
         Logging_ExecutionModeString = "Unknown"
      Case CscScriptModeValidation
         Logging_ExecutionModeString = "Validation " & Project.ScriptExecutionInstance
         If Project.WebBasedValidationEnabled Then
            THIN_CLIENT = True
         End If
      Case CscScriptModeValidationDesign
         Logging_ExecutionModeString = "ValidationDesign " & Project.ScriptExecutionInstance
      Case CscScriptModeVerification
         Logging_ExecutionModeString = "Verification"
         If Project.WebBasedVerificationEnabled Then
            THIN_CLIENT = True
         End If
      Case CscScriptModeVerificationDesign
         Logging_ExecutionModeString = "VerificationDesign"
      Case CscScriptModeDocumentReview
         Logging_ExecutionModeString = "DocumentReview"
         If Project.WebBasedDocumentReviewEnabled Then
            THIN_CLIENT = True
         End If
      Case CscScriptModeCorrection
         Logging_ExecutionModeString = "Correction"
         If Project.WebBasedCorrectionEnabled Then
            THIN_CLIENT = True
         End If
      Case Else
         Logging_ExecutionModeString = "BeyondUnknown (" & Project.ScriptExecutionMode & ")"
   End Select

   If THIN_CLIENT Then
      Logging_ExecutionModeString = Logging_ExecutionModeString & " (TC)"
   End If

   Exit Function

   catch:
   Logging_ExecutionModeString = "Unknown Module (Error " & Err.Number & ")"
End Function


'return [classname|subname#linenum]
'  input is the return of WinWrap's CallersLine function: "[macroname|subname#linenum] linetext"
'  refer to the WinWrap documentation for "CallersLine Function" regarding the Depth parameter
Public Function Logging_StackLine(ByVal Caller As String) As String
   On Error GoTo catch

   'the function name (subname) and linenum are followed by a ]
   Dim EndPos As Integer
   EndPos = InStr(Caller, "]")

   'the function name will follow a |
   Dim StartPos As Integer
   StartPos = InStrRev(Caller, "|", EndPos) + 1

   'get the function name
   Dim FunctionAndLine As String
   FunctionAndLine = Mid(Caller, StartPos, EndPos - StartPos)

   'combine with class/folder
   Logging_StackLine = "[" & Logging_SheetClass(Caller) & "|" & FunctionAndLine & "]"

   Exit Function

   catch:
   FunctionAndLine = "Unknown Function (Error " & Err.Number & ")"
End Function


'return the name of the folder or class of the script at the given depth
'  input is the return of WinWrap's CallersLine function: "[macroname|subname#linenum] linetext"
'  refer to the WinWrap documentation for "CallersLine Function" regarding the Depth parameter
Public Function Logging_SheetClass(ByVal Caller As String) As String
   On Error GoTo catch

   'the sheet name (macroname) is followed by a |
   Dim EndPos As Integer
   EndPos = InStr(Caller, "|")

   'the sheet name will follow a \ from Project Builder
   'Project Script: [C:\ProjectFolder\ScriptProject|Document_BeforeProcessXDoc#827] 'Code
   'Other Classes: [C:\1|ValidationForm_ButtonClicked# 18] 'Code
   Dim StartPosPB As Integer
   StartPosPB = InStrRev(Caller, "\", EndPos) + 1

   'the sheet name will follow a * from runtime modules
   '[*ScriptProject|Document_BeforeProcessXDoc#881] 'Code
   Dim StartPosRuntime As Integer
   StartPosRuntime = InStrRev(Caller, "*", EndPos) + 1

   'Use whichever start position is found
   Dim StartPos As Integer
   If StartPosPB > StartPosRuntime Then
```

```
      StartPos = StartPosPB
   Else
      StartPos = StartPosRuntime
   End If

   'get the sheet name
   Dim Sheet As String
   Sheet = Mid(Caller, StartPos, EndPos - StartPos)

   'numeric sheet names should be classes or folders
   If IsNumeric(Sheet) Then
      Dim SheetNum As Long
      SheetNum = CLng(Sheet)

      'sheet numbers higher than zero are classes
      If SheetNum > 0 Then
         Dim TheClass As CscClass
         Set TheClass = Project.ClassByID(SheetNum)

         'make sure the class actually exists to prevent an error accessing the name
         If Not TheClass Is Nothing Then
            Logging_SheetClass = TheClass.Name
         Else
            Logging_SheetClass = "Unknown Class (" & SheetNum & ")"
         End If
      Else
         'negative sheet numbers are folders (use absolute value for folder level)
         SheetNum = Abs(SheetNum)
         Dim TheFolder As CscFolderDef
         Set TheFolder = Project.FolderByLevel(SheetNum)

         'make sure the folder actually exists to prevent an error accessing the name
         If Not TheFolder Is Nothing Then
            Logging_SheetClass = TheFolder.Name
         Else
            Logging_SheetClass = "Unknown Folder (" & SheetNum & ")"
         End If
      End If
   ElseIf Sheet = "ScriptProject" Then
      'Project level script has the special designation "ScriptProject"
      Logging_SheetClass = "Project"
   Else
      Logging_SheetClass = "Unknown Class (" & Sheet & ")"
   End If

   Exit Function

   catch:
   Logging_SheetClass = "Unknown Class (Error " & Err.Number & ")"
End Function

'Meant to be called from Batch_Close, this will log routing, rejection, and other details
Public Sub Logging_BatchClose(ByVal pXRootFolder As CASCADELib.CscXFolder, _
   ByVal CloseMode As CASCADELib.CscBatchCloseMode)
   On Error GoTo catch

   Select Case CloseMode
      'routing is evaluated after Final, Suspend, and Error modes
      Case CscBatchCloseMode.CscBatchCloseError
         ErrorLog(Err, "Closing batch in error:" & BATCH_ID & "/" & BATCH_ID_HEX & ", " & _
            BATCH_NAME, Nothing, pXRootFolder, 0, FORCE_ERROR, SUPPRESS_MSGBOX)
         Logging_Routing(pXRootFolder)

         'find any rejected docs
         Dim RejectedMsg As String
         Logging_RejectedDocs(pXRootFolder, RejectedMsg, pXRootFolder.XValues)
```

```
          'if there are rejected docs/pages
          If RejectedMsg <> "" Then
             RejectedMsg = "The following have been rejected: " & vbNewLine & RejectedMsg
             ErrorLog(Err, RejectedMsg, Nothing, pXRootFolder, 0, FORCE_ERROR, SUPPRESS_MSGBOX)

             'Potentially take extra action if there is a script error
             '   (set by Logging_RejectedDocs)
             If pXRootFolder.XValues.ItemExists("LOGGING_SCRIPT_ERROR") Then
                'script error action
             End If
          End If

       Case CscBatchCloseMode.CscBatchCloseSuspend
          ScriptLog("Suspending Batch:" & BATCH_ID & "/" & BATCH_ID_HEX & ", " & _
             BATCH_NAME, LOCAL_LOG)
          Logging_Routing(pXRootFolder)

       Case CscBatchCloseMode.CscBatchCloseFinal
          ScriptLog("Batch Close")
          Logging_Routing(pXRootFolder)

       Case CscBatchCloseMode.CscBatchCloseParent
          'Application_InitializeBatch is not called between Child and Parent Batch_Close
          '  so initialize logging paths again otherwise Parent logging will go to the Child log
          Logging_InitializeBatch(pXRootFolder)

          'Log that we have "opened" the parent batch
          Logging_BatchOpen(pXRootFolder)
          ScriptLog("Routing complete, closing parent batch.")

       Case CscBatchCloseMode.CscBatchCloseChild
          'Note that if a child batch has been routed to a new batch class,
          '   this will Batch_Close will not fire for the child

          'Log that we have "opened" the child batch
          Logging_BatchOpen(pXRootFolder)

          'See if we can find out which tag this was created with during routing
          Dim i As Integer
          Dim BatchTag As String
          For i = 0 To pXRootFolder.XValues.Count - 1
             If Mid(pXRootFolder.XValues.ItemByIndex(i).Key, 1, _
                Len("KTM_DOCUMENTROUTING_QUEUE_")) = "KTM_DOCUMENTROUTING_QUEUE_" Then
                BatchTag = Mid(pXRootFolder.XValues.ItemByIndex(i).Key, _
                   Len("KTM_DOCUMENTROUTING_QUEUE_") + 1)
                ScriptLog("This batch has been created as a result of routing with " & _
                   "the tag: " & BatchTag)
             End If
          Next
          If BatchTag = "" Then
             ScriptLog("This batch has been created as a result of routing.")
          End If

       Case Else
          ErrorLog(Err, "Unknown Batch Close Type!", Nothing, pXRootFolder, 0, _
             FORCE_ERROR, SUPPRESS_MSGBOX)
    End Select

    Exit Sub

    'if there is an error, log it and try to keep going
    catch:
    ErrorLog(Err, "", Nothing, pXRootFolder, 0, ONLY_ON_ERROR, SUPPRESS_MSGBOX)
    Resume Next
 End Sub

 'Recursive function to check for rejected docs/pages, called from Logging_BatchClose
```

```
Public Sub Logging_RejectedDocs(ByVal XFolder As CscXFolder, ByRef msg As String, _
   ByRef XValues As CscXValues)
   On Error GoTo catch

   Dim i As Integer

   'recurse into folders
   For i = 0 To XFolder.Folders.Count - 1
      Logging_RejectedDocs(XFolder.Folders.ItemByIndex(i), msg, XValues)
   Next

   Dim RejectionNote As String

   'check documents
   Dim XDocInfo As CscXDocInfo
   For i = 0 To XFolder.DocInfos.Count - 1
      Set XDocInfo = XFolder.DocInfos.ItemByIndex(i)

      'check if the doc is rejected
      If XDocInfo.XValues.ItemExists("AC_REJECTED_DOCUMENT") Then
         'identify doc
         msg = msg & Logging_IdentifyFolderDocPage(Nothing, XDocInfo.XDocument)

         'add rejection note if exists
         If XDocInfo.XValues.ItemExists("AC_REJECTED_DOCUMENT_NOTE") Then
            RejectionNote = XDocInfo.XValues.ItemByName("AC_REJECTED_DOCUMENT_NOTE").Value
            msg = msg & ": " & RejectionNote & vbNewLine

            'if the rejection note mentions (S/s)cript,
            '   note for later that there has been a script error
            If InStr(1, RejectionNote, "cript") > 0 Then
               XValues.Set("LOGGING_SCRIPT_ERROR", "True")
            End If
         Else
            msg = msg & vbNewLine
         End If
      End If

      'check pages
      Dim PageIndex As Long
      For PageIndex = 0 To XDocInfo.PageCount - 1
         'check if the page is rejected
         If XDocInfo.XValues.ItemExists("AC_REJECTED_PAGE" & CStr(PageIndex + 1)) Then
            'identify page
            msg = msg & Logging_IdentifyFolderDocPage(Nothing, XDocInfo.XDocument, PageIndex + 1)

            'add rejection note if exists
            If XDocInfo.XValues.ItemExists("AC_REJECTED_PAGE_NOTE" & CStr(PageIndex + 1)) Then
               RejectionNote = XDocInfo.XValues.ItemByName("AC_REJECTED_PAGE_NOTE" & _
                  CStr(PageIndex + 1)).Value
               msg = msg & ": " & RejectionNote & vbNewLine
            Else
               msg = msg & vbNewLine
            End If
         End If
      Next

   Next

   'on error log and exit
   catch:
   ErrorLog(Err, "", Nothing, Nothing, 0, ONLY_ON_ERROR, SUPPRESS_MSGBOX)
End Sub


'Called from Logging_BatchClose to log documents that will be routed
Public Sub Logging_Routing(ByVal pXRootFolder As CscXFolder)
```

```
   On Error GoTo catch

   'This will hold the routing information we find:
   '   key=LOGGING_ROUTING_batchtag, value=folders and docs
   Dim RoutingGroups As CscXValues
   Set RoutingGroups = pXRootFolder.XValues

   'Set a flag saying all documents have been routed
   '  Finding an unrouted document will set this to false
   RoutingGroups.Set("LOGGING_ALLROUTED", "True")

   'recursively check folders for routing, adding results to RoutingGroups
   Logging_RoutingFolder(pXRootFolder, RoutingGroups)

   'If all docs are routed this batch will get deleted, so log this to local log
   If pXRootFolder.XValues.ItemByName("LOGGING_ALLROUTED").Value = "True" Then
      ScriptLog("All documents in the batch appear to be routed.  " & _
         "Batch will be deleted.", LOCAL_LOG)
   End If

   'log if the original batch will be routed to a module
   If RoutingGroups.ItemExists("KTM_DOCUMENTROUTING_QUEUE_THISBATCH") Then
      ScriptLog("This original batch will be routed to " & _
         pXRootFolder.XValues.ItemByName("KTM_DOCUMENTROUTING_QUEUE_THISBATCH").Value)
   End If

   'go through the document routing groups and log details
   Dim msg As String
   Dim BatchTag As String
   Dim i As Integer
   For i = 0 To RoutingGroups.Count - 1
      'if the XValue key begins with "LOGGING_ROUTING_"
      If Mid(RoutingGroups.ItemByIndex(i).Key, 1, _
         Len("LOGGING_ROUTING_")) = "LOGGING_ROUTING_" Then

         'the part after "LOGGING_ROUTING_"
         BatchTag = Mid(RoutingGroups.ItemByIndex(i).Key, Len("LOGGING_ROUTING_") + 1)

         msg = msg & "Routing group (" & BatchTag

         'check if it is being routed to a specific queue
         If pXRootFolder.XValues.ItemExists("KTM_DOCUMENTROUTING_QUEUE_" & BatchTag) Then
            msg = msg & ", Queue=" & _
               pXRootFolder.XValues.ItemByName("KTM_DOCUMENTROUTING_QUEUE_" & BatchTag).Value
         End If

         'check if it is being routed with a specific batch name KTM 5.5+
         If pXRootFolder.XValues.ItemExists("KTM_DOCUMENTROUTING_BATCHNAME_" & BatchTag) Then
            msg = msg & ", Batch Name=" & _
               pXRootFolder.XValues.ItemByName("KTM_DOCUMENTROUTING_BATCHNAME_" & BatchTag).Value
         End If

         'check if it is being routed to a new batch class KTM 5.5+
         If pXRootFolder.XValues.ItemExists("KTM_DOCUMENTROUTING_NEWBATCHCLASS_" & BatchTag) Then
            msg = msg & ", Batch Class=" & _
               pXRootFolder.XValues.ItemByName("KTM_DOCUMENTROUTING_NEWBATCHCLASS_" & _
               BatchTag).Value & _
               " (module will be ignored)"
         End If

         msg = msg & "): " & RoutingGroups.ItemByIndex(i).Value & vbNewLine
      End If
   Next

   'if there were any routing groups, msg won't be empty
   If msg <> "" Then
      ScriptLog(msg)
   End If


   'on error log and exit
   catch:
   ErrorLog(Err, "", Nothing, Nothing, 0, ONLY_ON_ERROR, SUPPRESS_MSGBOX)
End Sub


'recursively check folders for routed documents (or first level routed folders),
'   adding results to RoutingGroups
Public Sub Logging_RoutingFolder(ByVal XFolder As CscXFolder, ByRef RoutingGroups As CscXValues)
   On Error GoTo catch

   'only 1st level folders can be routed (but any documents can be routed)
   Dim IsFirstLevelFolder As Boolean
   IsFirstLevelFolder = False

   If XFolder.IsRootFolder = False Then 'not the root
      If XFolder.ParentFolder.IsRootFolder = True Then 'parent is the root
         IsFirstLevelFolder = True
      End If
   End If

   Dim BatchTag As String

   'check if this folder is being routed
   If IsFirstLevelFolder And XFolder.XValues.ItemExists("KTM_DOCUMENTROUTING") Then
      BatchTag = XFolder.XValues.ItemByName("KTM_DOCUMENTROUTING").Value

      Dim FolderName As String
      FolderName = Logging_IdentifyFolderDocPage(XFolder)

      'check if we've already added this group
      If RoutingGroups.ItemExists("LOGGING_ROUTING_" & BatchTag) Then
         'add this folder
         RoutingGroups.Set("LOGGING_ROUTING_" & BatchTag, _
            RoutingGroups.ItemByName("LOGGING_ROUTING_" & BatchTag).Value & "," & FolderName)
      Else
         'create it and add this document
         RoutingGroups.Set("LOGGING_ROUTING_" & BatchTag, FolderName)
      End If

      'if the folder is being routed, it will route the contents,
      '  and if routing instructions were set on these contents, they will be ignored
   Else
      'if the folder is not being routed, check if its subfolders
      Dim SubFolder As CscXFolder
      Dim i As Integer
      For i = 0 To XFolder.Folders.Count - 1
         Set SubFolder = XFolder.Folders.ItemByIndex(i)
         Logging_RoutingFolder(SubFolder, RoutingGroups)
      Next

      Dim oXDocInfo As CscXDocInfo
      Dim DocName As String

      'check for routed docs in this folder
      For i = 0 To XFolder.DocInfos.Count - 1
         Set oXDocInfo = XFolder.DocInfos.ItemByIndex(i)
         DocName = Logging_IdentifyFolderDocPage(Nothing, oXDocInfo.XDocument)

         'check if this document is being routed
         If oXDocInfo.XValues.ItemExists("KTM_DOCUMENTROUTING") Then
            BatchTag = oXDocInfo.XValues.ItemByName("KTM_DOCUMENTROUTING").Value

            'check if we've already added this group
            If RoutingGroups.ItemExists("LOGGING_ROUTING_" & BatchTag) Then
               'add this document
               RoutingGroups.Set("LOGGING_ROUTING_" & BatchTag, _
                  RoutingGroups.ItemByName("LOGGING_ROUTING_" & BatchTag).Value & "," & DocName)
            Else
               'create it and add this document
               RoutingGroups.Set("LOGGING_ROUTING_" & BatchTag, DocName)
            End If
         Else
            'If a document is not being routed, we know they are not all routed
            RoutingGroups.Set("LOGGING_ALLROUTED", "False")
         End If
      Next
   End If


   'on error log and exit
   catch:
   ErrorLog(Err, "", Nothing, Nothing, 0, ONLY_ON_ERROR, SUPPRESS_MSGBOX)
End Sub


'Identifies structure and files from Folder/Doc/Page.
Public Function Logging_IdentifyFolderDocPage(Optional ByVal XFolder As CscXFolder = Nothing, _
   Optional ByVal XDoc As CscXDocument = Nothing, _
   Optional ByVal PageNum As Integer = 0) As String
   'valid parameter combinations:
   'only folder
   'only doc, implies folder
   'doc and page number (page object doesn't link to parent doc)

   On Error GoTo catch

   'if doc was provided, use that to set folder
   If Not XDoc Is Nothing Then
      Set XFolder = XDoc.ParentFolder
   Else
      'if doc was not provided, make sure we have a folder, or exit
      If XFolder Is Nothing Then
         Exit Function
      Else
         'also exit if we only have root because we are omitting root folder info
         If XFolder.IsRootFolder Then
            Exit Function
         End If
      End If
   End If

   'Structure will show info like F#\F#\D#\P#
   Dim DocStructure As String

   'Files will show info like (#.xdc\#.tif(#))
   '  (#) is the page number in that document
   '  All folders are in Folder.xfd, so no need to add that
   Dim Files As String


   'get the folder structure (other than root folder since it is always there)
   Do While Not XFolder.IsRootFolder
      Set XFolder = XFolder.ParentFolder
      DocStructure = "F" & XFolder.IndexInFolder + 1 & "\" & DocStructure
      Files = Mid(XFolder.FileName, InStrRev(XFolder.FileName, "\")) & Files
   Loop

   'get the document
   If Not XDoc Is Nothing Then
      DocStructure = DocStructure & "D" & XDoc.IndexInFolder + 1
      Files = Files & Mid(XDoc.FileName, InStrRev(XDoc.FileName, "\") + 1)
   End If

   'get the page
   If PageNum > 0 And Not XDoc Is Nothing Then
      DocStructure = DocStructure & "\P" & PageNum

      Dim Page As CscXDocPage
      Set Page = XDoc.Pages.ItemByIndex(PageNum - 1)
      'make sure the page exists
      If Not Page Is Nothing Then
         Files = Files & Mid(Page.SourceFileName, InStrRev(Page.SourceFileName, "\"))
      End If
   End If

   Logging_IdentifyFolderDocPage = DocStructure & " (" & Files & ")"


   Exit Function

   catch:
   Logging_IdentifyFolderDocPage = "Unknown Folder/Doc/Page"
End Function


'Wrapper and drop-in replacement for MsgBox.
Public Function MsgBoxLog(ByVal Message As String, _
   Optional ByVal MsgType As VbMsgBoxStyle, _
   Optional ByVal Title As String, _
   Optional ByVal pXDoc As CscXDocument=Nothing, _
   Optional ByVal pXFolder As CscXFolder=Nothing, _
   Optional PageNum As Integer=0) As VbMsgBoxResult

   On Error GoTo catch

   'Figure out what kind of MsgBox style this is (one from each group)
   Dim TypeString As String

   'Buttons
   If CInt(MsgType And vbOkOnly) = vbOkOnly Then
      TypeString = "vbOkOnly, "
   ElseIf CInt(MsgType And vbOkCancel) = vbOkCancel Then
      TypeString = "vbOkCancel, "
   ElseIf CInt(MsgType And vbAbortRetryIgnore) = vbAbortRetryIgnore Then
      TypeString = "vbAbortRetryIgnore, "
   ElseIf CInt(MsgType And vbYesNoCancel) = vbYesNoCancel Then
      TypeString = "vbYesNoCancel, "
   ElseIf CInt(MsgType And vbYesNo) = vbYesNo Then
      TypeString = "vbYesNo, "
   ElseIf CInt(MsgType And vbRetryCancel) = vbRetryCancel Then
      TypeString = "vbRetryCancel, "
   End If

   'Icon
   If CInt(MsgType And vbCritical) = vbCritical Then
      TypeString = TypeString & "vbCritical, "
   ElseIf CInt(MsgType And vbQuestion) = vbQuestion Then
      TypeString = TypeString & "vbQuestion, "
   ElseIf CInt(MsgType And vbExclamation) = vbExclamation Then
      TypeString = TypeString & "vbExclamation, "
   ElseIf CInt(MsgType And vbInformation) = vbInformation Then
      TypeString = TypeString & "vbInformation, "
   End If

   'Default
   If CInt(MsgType And vbDefaultButton1) = vbDefaultButton1 Then
      TypeString = TypeString & "vbDefaultButton1, "
```

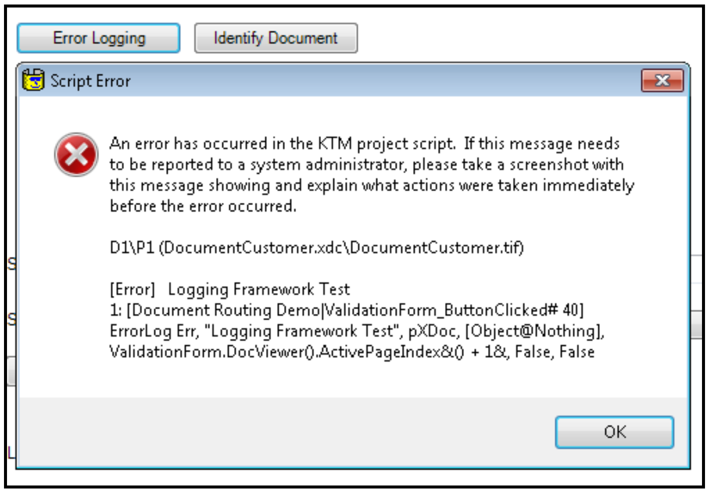
```
    ElseIf CInt(MsgType And vbDefaultButton2) = vbDefaultButton2 Then
       TypeString = TypeString & "vbDefaultButton2, "
    ElseIf CInt(MsgType And vbDefaultButton3) = vbDefaultButton3 Then
       TypeString = TypeString & "vbDefaultButton3, "
    End If

    'Default
    If CInt(MsgType And vbApplicationModal) = vbApplicationModal Then
       TypeString = TypeString & "vbApplicationModal"
    ElseIf CInt(MsgType And vbSystemModal) = vbSystemModal Then
       TypeString = TypeString & "vbSystemModal"
    ElseIf CInt(MsgType And vbMsgBoxSetForeground) = vbMsgBoxSetForeground Then
       TypeString = TypeString & "vbMsgBoxSetForeground"
    End If


    'log an error as a warning if this is used during server
    If Project.ScriptExecutionMode = CscScriptExecutionMode.CscScriptModeServer Then
       ErrorLog(Err, "Skipping a MsgBox and forcing an OK result because " & _
          "it is running during Server. Message: """ & Message & """ (" & TypeString & _
          ")", pXDoc, pXFolder, PageNum, FORCE_ERROR, SUPPRESS_MSGBOX)
       MsgBoxLog = vbOK
       Exit Function
    End If

    'MsgBox also cannot be used from a Thin Client
    If THIN_CLIENT Then
       ErrorLog(Err, "Skipping a MsgBox and forcing an OK result because " & _
          "it is running during Thin Client. Message: """ & Message & """ (" & TypeString & _
          ")", pXDoc, pXFolder, PageNum, FORCE_ERROR, SUPPRESS_MSGBOX)
       MsgBoxLog = vbOK
       Exit Function
    End If

    'show the message and grab the result
    Dim Result As VbMsgBoxResult
    Result = MsgBox(Message, MsgType, Title)

    'Find out what the user clicked
    Dim ResultString As String
    Select Case Result
       Case vbOK
          ResultString = "OK"
       Case vbCancel
          ResultString = "Cancel"
       Case vbAbort
          ResultString = "Abort"
       Case vbRetry
          ResultString = "Retry"
       Case vbIgnore
          ResultString = "Ignore"
       Case vbYes
          ResultString = "Yes"
       Case vbNo
          ResultString = "No"
       Case Else
          ResultString = "Unknown"
    End Select

    'log the details
    ScriptLog("MsgBox: User clicked " & ResultString & " for message """ & Message & """ (" & _
       TypeString & ")", BATCH_LOG, pXDoc, pXFolder, PageNum, IGNORE_CURRENT_FUNCTION)

    'return the result just like a normal MsgBox
    MsgBoxLog = Result

    Exit Function
```

```
   'if there is an error, log it and try to keep going
   catch:
   ErrorLog(Err, "", pXDoc, pXFolder, 0, ONLY_ON_ERROR, SUPPRESS_MSGBOX)
   Resume Next
End Function


'========  END LOGGING CODE =======
```

Testing the Logging Framework

To test some of the features of this framework, copy the code from the prior section into a project then create buttons on a Validation Form with the following names: LogError, LogMsgBox, and OpenLogFile. Finally, copy the following code into the script of the class with the Validation Form:

```
Private Sub ValidationForm_ButtonClicked(ByVal ButtonName As String, _
   ByVal pXDoc As CASCADELib.CscXDocument)
   On Error GoTo catch

   Select Case ButtonName
      Case "LogError"
         'Logging_Test_ErrorLogging(pXDoc)
         Err.Raise(vbObjectError, "", "Testing Error Logging")
      Case "LogMsgBox"
         Dim MsgBoxResult As VbMsgBoxResult
         Set MsgBoxResult = MsgBoxLog("This message, style, and user choice will be logged.", _
            vbYesNoCancel Or vbCritical, "MsgBoxLog Example", pXDoc, Nothing, _
            ValidationForm.DocViewer.ActivePageIndex + 1)
      Case "OpenLogFile"
         'Notepad is available in a directory already on the system PATH variable
         Dim TextEditorCmdLine As String
         TextEditorCmdLine = "notepad"

         'check if we are in design or runtime
         If Project.ScriptExecutionMode = CscScriptModeServerDesign Or _
            Project.ScriptExecutionMode = CscScriptModeValidationDesign Or _
            Project.ScriptExecutionMode = CscScriptModeVerificationDesign Then

            Shell(TextEditorCmdLine & " """ & Logging_CaptureLocalLogs() & Format(Now(), _
               "yyyymmdd") & DESIGN_LOG_FILENAME & """", vbNormalFocus)
         Else
            Shell(TextEditorCmdLine & " """ & BATCH_IMAGE_LOGS & BATCH_ID_HEX & _
               BATCH_LOG_FILENAME & """", vbNormalFocus)
         End If
   End Select

   Exit Sub

   catch:
   ErrorLog(Err, "Logging Framework Test", pXDoc, Nothing, _
      ValidationForm.DocViewer.ActivePageIndex + 1)
   Resume Next
End Sub
```

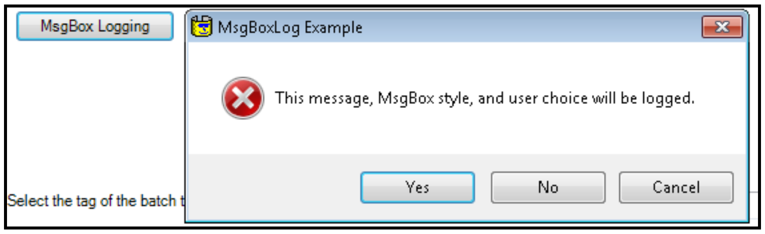


An error has occurred in the KTM project script. If this message needs to be reported to a system administrator, please take a screenshot with this message showing and explain what actions were taken immediately before the error occurred.

D1\P1 (DocumentCustomer.xdc\DocumentCustomer.tif)

[Error] Logging Framework Test
1: [Document Routing Demo|ValidationForm_ButtonClicked# 40]
ErrorLog Err, "Logging Framework Test", pXDoc, [Object@Nothing], ValidationForm.DocViewer().ActivePageIndex&() + 1&, False, False

Clicking the LogError button



This message, MsgBox style, and user choice will be logged.

Clicking the LogMsgBox button

After clicking the LogError button and the LogMsgBox button, clicking the OpenLogFile button will show Notepad, where the design log will show that the following has been logged:

```
7/23/2011 9:38:56 PM (77936.24) -- D1\P1 (DocumentCustomer.xdc\DocumentCustomer.tif) --
ValidationDesign 1 -- [Document Routing Demo|ValidationForm_ButtonClicked# 40]
        [Error]   Logging Framework Test
        1: [Document Routing Demo|ValidationForm_ButtonClicked# 40] ErrorLog Err, "Logging
Framework Test", pXDoc, [Object@Nothing], ValidationForm.DocViewer().ActivePageIndex&() + 1&,
False, False

7/23/2011 9:39:04 PM (77943.87) -- D1\P1 (DocumentCustomer.xdc\DocumentCustomer.tif) --
ValidationDesign 1 -- [Document Routing Demo|ValidationForm_ButtonClicked# 19]
        MsgBox: User clicked Cancel for message "This message, MsgBox style, and user choice will
be logged." (vbOkOnly, vbCritical, vbDefaultButton1, vbApplicationModal)
```

Summary

For a KTM project with script customization, it is essential to write organized script code. Implementing script logging, such as the framework presented here, helps diagnose problems and can aid in maintaining the project. If problems occur, having an informative log can be invaluable in diagnosing and resolving the issue.