Replaying ADK Files for Testing Blaise Applications

Jason Ostergren and Rhonda Ash, Health and Retirement Study, Institute for Social Research, University of Michigan

1. INTRODUCTION

The Health and Retirement Study (HRS) is a major national longitudinal survey on the health and economics of aging and retirement. It consists of a complex multi-sectional hour long interview of 22,000+ participants and is collected every two years using CAI (computer aided interviewing) technology. It utilizes preload of more than 350 variables from previous wave data, and incorporates seven mode and language combinations. The scope of the HRS instrument exacerbates the common difficulties inherent in accurately programming and testing a CAI application.

This paper will cover the conceptualization, implementation and practical results of a program which replays audit trail files with the goal of increasing efficiency and accuracy of instrument programming and testing on a project of this scale.

2. PREMISE

Testing a CAI instrument presents a host of complexities. The desire to find a means of reducing the burden of testing and a means of automating parts of the process is apparently quite widespread.[5] For some time HRS had been interested in having an application at its disposal that would allow a tester to retrace the path through an interview as recorded in an audit trail. During the transition to Blaise from another survey language in 2002, HRS began to work on the concept for just such an application.

Much of the specification, programming and initial testing of a survey instrument falls outside the purview of this program. The application is not intended to supersede Q by Q and scenario testing, but rather to improve the process of fixing and retesting errors that have already been found and to reduce the likelihood of introducing new errors in the process. Additionally, HRS requires a way to routinize certain testing activities where a particular scenario or path through the instrument is repeated numerous times, possibly by different people. The different needs and types of testing that are addressed by this program are described below.

Multitask testing is difficult, at best. It is nearly impossible for a tester to focus on several detailed aspects of a large application, such as skip patterns, codeframe values, screen format and colors, and QxQ help, all at once. However, a single set of paths through an application, as recorded in the audit trail files, will allow testers who specialize in different areas of testing to replay those same paths, focusing, in the process, on their designated area of concern. As an example, HRS has testers that have experience with particular kinds of content that are often focused in a small set of sections. They know the expected population that should flow through their sections and how to recognize problems when testing scenarios. Audit trail files (or "keystroke files" for shorthand) can be captured from these specialists' testing activities. These keystroke files can then be reused or used by additional testers to review other areas of interest,

[5] Tarnai, John and Moore, Danna L., Methods for Testing and Evaluating Computer-Assisted Questionnaires, *Methods for Testing and Evaluating Survey Questionnaires*, Ed. By Stanley Presser, et al., New Jersey: John Wiley & Sons, 2004: pp. 319-335.

such as screen formatting. Further, minor changes can be made during keystroke replay to increase the utility of the process for these testers. For instance, a language specialist may wish to change the language keystroke at the beginning of the application and replay the rest of the audit trail, reviewing the text and codeframes in a different language than was originally recorded.

The complexity and length of a survey are major factors in the amount of time required for testing. In a large survey, programmers and testers typically expend enormous amounts of time repeatedly keying their way to a critical testing juncture. HRS sees a need to be able to replay a set of keystrokes to get to a particular point in an interview for situations in which a programmer or tester needs to repeatedly work in a given area of the instrument. Moreover, answers to a section in the middle of the application could impact the flow and/or fills in sections at the end, so it may be necessary to test repetitively through large parts of the instrument, stopping to alter a particular set of answers at a given point. A great deal of time can be saved by having the ability to reproduce part of an interview or testing path, change a few answers and continue to replay keystrokes up to the area of key interest for testing. Further, this functionality can be used more generally to replay a keystroke file and to diverge from it at a certain point to test other routes.

The most important function of this application is to aid in reproducing errors and retesting fixes. This is needed for both the development phase and during the field period. During development, audit trail files can be gathered from testers who report errors. Replaying those audit trails takes the programmer who is investigating the error directly to the problem, offering an advantage in reproducing problems that is fairly unique to the field of survey programming. Additionally, the programmer can readily replay the keystrokes to the point of the error to test and to confirm the validity of any fixes for the problem. Once in the field, this allows the replay of cases for which an interviewer reported a problem. By reproducing the interview through this process, the staff can diagnose, fix, re-test and document the problem rapidly. This can aid the correction of problems while still in the field and assist in the documentation of the data in cases that have already been completed.

The continual changes to programming specifications that HRS commonly experiences during development, even after the programming of a section has been completed, often affect other sections beyond the one being altered. Once a section has been fully tested, the ability to preserve a set of keystrokes reflecting the functioning section is important. It is then possible to replay them at a later date to identify any errors that were introduced by alterations to other sections. This can also allow for comparing data between runs to quickly pick out any differences.

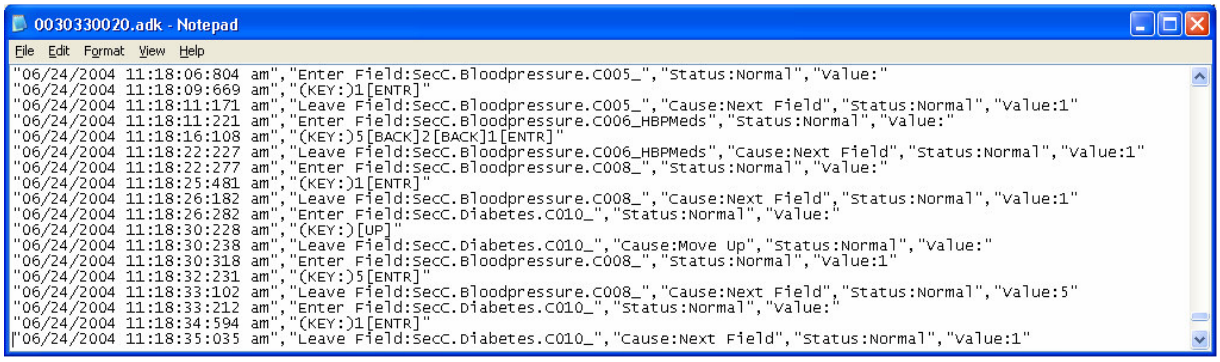
HRS endeavored to meet the need for a utility to reduce testing difficulties by conceiving of and building a utility called “Replayer.” This program makes use of the keystroke files produced as a byproduct of Blaise tests and interviews to allow the rapid playback of common scenarios, reports of problems by testers (the reporting system saves keystroke files for this purpose), and programmer fixes for verification. The “Replayer” needs to allow for “fast forwarding” to a chosen point in a given interview, as well as for playback and review of individual keystrokes, and for switching between the “Replayer” program and the interview in the Blaise Data Entry Program (DEP) to manually adjust values when needed.

3. IMPLEMENTATION

There were a few significant hurdles to overcome in order to build an effective program to meet the aforementioned needs (hereafter referred to as the "Replayer" program). Discussion will include the key problems involved in programming such a utility using BCP and Visual Basic 6. The most obvious of these is how to make effective use of the keystroke files, despite some deficiencies, in order to extract the necessary details of the interviewer's actions. Next, the need to send keystrokes reliably to the Data Entry Program (DEP) is best met using the Windows API; this method has distinct advantages over other methods. Finally, the principle technical obstacle involves finding a way to communicate the necessary information from the DEP to the "Replayer" program, particularly the currently focused field. This discussion will cover solutions to each of these problems and sample code will be provided as well.

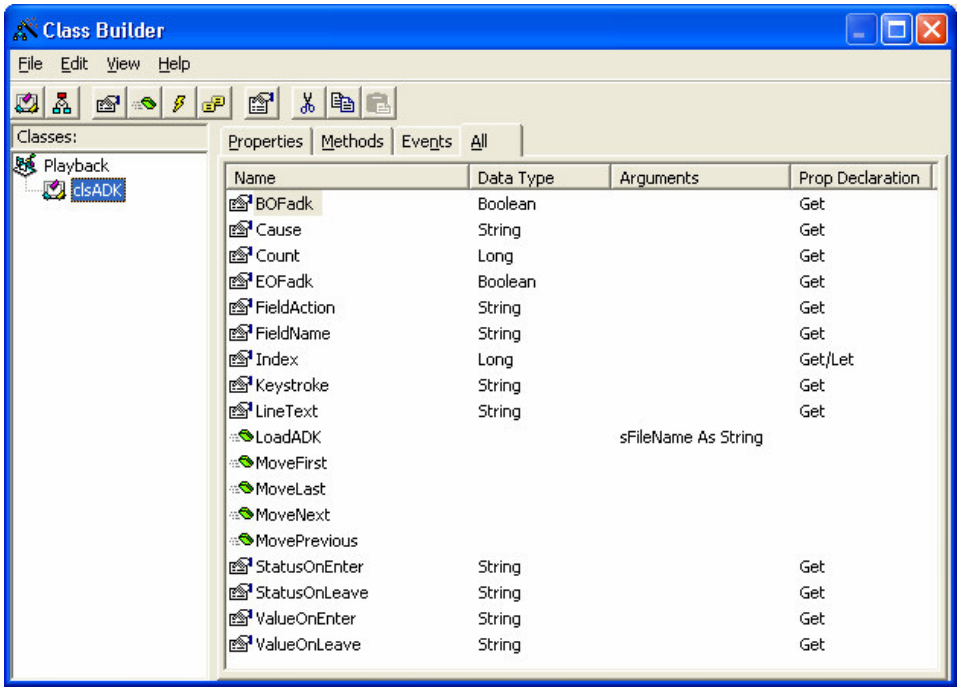
3.1 IMPLEMENTATION - BASICS

The prerequisite to building a "Replayer" program is a case management system or simulator that can serve up an accurate reproduction of the environment of the original test or interview during which the audit trail file (.adk) was generated. In some cases, it may be possible to use the same sample management system as the interviewers use in the field. In other cases, it may be necessary to develop an application to simulate this process. It is beyond the scope of this paper to describe this in detail. However, its basic functionality must include the ability to gather the correct datamodel, preload, and .adk file and to initiate the DEP with the preload correctly loaded.



A typical segment of an Audit Trail file (.adk)

The first task involved in programming a "Replayer" is to build a means to load the .adk data (see illustration above) in order for the program to know which values to place in which fields. This can be done, as needed, reading one line at a time in various parts of the program, but in the long run it will save time to program a simple .adk handler class. The class should contain cursor-like methods able to navigate an array containing the data which can be loaded from an adk by another method, and should provide easy access to the relevant data in the focused line (see illustration).



Properties and Methods of a class for handling .adks

As part of the design of a "Replayer" program, it is also necessary to consider the variety of functions needed, such as playback of single or multiple keystrokes, the ability to skip keystrokes when necessary, the ability to halt playback at a specified field, the ability to switch between playback of individual keystrokes and final values, and so on. HRS ultimately found it necessary to incorporate all of the above functionality.

3.2 IMPLEMENTATION - INHERENT OBSTACLES

There are a number of technical obstacles that should be considered which prefigure problems for the idea of retracing one's steps through a case. These can either be the result of questionnaire design decisions or they can be traced to deficiencies in the .adk generation process.

Questionnaire design can impede the process of retracing an .adk in a number of ways. If a datamodel is designed with conditions that affect branching that are based on time sensitive or randomized values generated on the fly, it may not be possible to expect that the instrument will play through the same as the .adk on any given occasion. For example, if the instrument calculates an age based on the current time and potentially branches differently based on that age, then an attempt to replay the case at a later date may result in a different path through the instrument. One solution may be to generate such values as preload. However, the instrument specifications may necessitate these session-sensitive values and special provision may have to be made, such as instrument-specific procedures for replaying certain problematic fields or even altering the computer clock during replay, for example. If such problems are infrequent then it may be sufficient to manually adjust the instrument when these issues arise, however. The HRS instrument does make use of session-specific data, but, due to the rarity of such problems during replay, no effort has yet been made to remedy this programmatically.

Similarly, if an instrument uses alien routers, these may also require workarounds, because .adk files contain little information about router activity. It is simple enough to manually answer routers when they appear, but there may be a need to replay cases without manual intervention. In

some cases, it may be possible to adjust the programming of a router to accommodate this need, or it may work to simply remove the files during replay if they do not alter values in the instrument. In the case of HRS, which calls two types of routers that output data, it has not yet been possible to adjust the programming of the routers, so special functions were built into the "Replayer" program to accommodate them. In this case, doing so involved making a separate version of the router for automatic playback which returned control to the DEP immediately. Then, by reading ahead in the .adk to determine the final value, the original router result could be output by the "Replayer" program in place of the router (in this case by means of a text file on the local drive).

Perhaps the most important instrument design issue to take into account is the complexity of preload. This is not specifically a concern for the "Replayer" program itself, but rather for the accompanying case setup application that may be required. For the program to retrace the original route of the .adk, it is of course necessary for the preload environment to match the original. In the case of HRS, the very complex preload has necessitated the development of more than one sample management simulation program to serve up the correct preload environment for different kinds of testing.

There are also deficiencies in the .adk generation process in Blaise (some probably by design, others perhaps are errors) that must be taken into account in the design of a "Replayer" application. The most significant issue is that mouse activity is not recorded in a direct way. This is a problem for two reasons. First, the mouse can be used to change the focused field of the questionnaire in a non-linear fashion. This is very disruptive to the job of retracing an .adk because it is not simple to determine how to move the current focus back (or forward) to the field put in focus by the mouse originally. Further, the act of realigning the focus can really only be done by adding keystrokes that did not originally occur (several up-arrow keystrokes, for example). HRS has approached this problem from two angles. First, HRS has implemented, in our testing and in the field, a small program which prevents mouse input to the FormPane and the InfoPane, thus forcing the tester or interviewer to move through the questionnaire using only keystrokes. Second, a function has been implemented in the "Replayer" program that optionally attempts to automatically realign with the .adk when the fields do not match. The latter has the added advantage of helping in situations where other kinds of problems have caused the actual and the recorded fields to become misaligned.

The second problem with mouse input is that answers can be chosen in the InfoPane without leaving information about the individual mouse clicks, but only the final value when exiting the field. This, too, has been addressed by HRS by means of disabling mouse input to the InfoPane. As an alternative, the "Replayer" also provides an option to use the values recorded when the field is exited rather than the actual keystrokes. This prevents the user from watching the actual original input, but works well for other purposes.

A sort of problem similar to changing focus with the mouse occurs when certain keys are pressed and held by the tester or interviewer. The .adk only records the original keypress, not the duration or the effect. Thus, pressing and holding the up-arrow key may move the focus by several fields, but will record only one keypress in the .adk. This also applies to keys such as Shift and Control; it is not possible to tell if the Shift key was pressed and released before the subsequent keys, if it was in effect for only one key, or if it affected several subsequent keystrokes, for example. Primarily, this issue affects focus and the solutions are similar to those used to combat the mouse-focus problem. This is one of the areas where the auto-realign feature of the "Replayer" might be used, or the realignment can always be done manually, if the process is supervised. In this case, HRS also instructs its testers and interviewers not to press and hold keys.

Other problems that occur are more difficult to pin down, such as lines missing or truncated in the .adk files, and a host of other rare and seemingly inexplicable mistakes. For example, one problem that comes up occasionally is that keystrokes appear between the Leave Field and Enter Field lines of an .adk, as if they were suspended in the gap between fields. What makes this issue particularly thorny is that sometimes it appears that these keys were processed by the DEP and at other times they appear to have been ignored. This inconsistency makes it very difficult to program a workaround into the "Replayer." Currently, HRS treats such keystrokes as if they should be ignored.

As much as these and other obstacles make the process of building a "Replayer" complicated, it is still possible to build an effective application to meet the testing needs outlined at the outset.

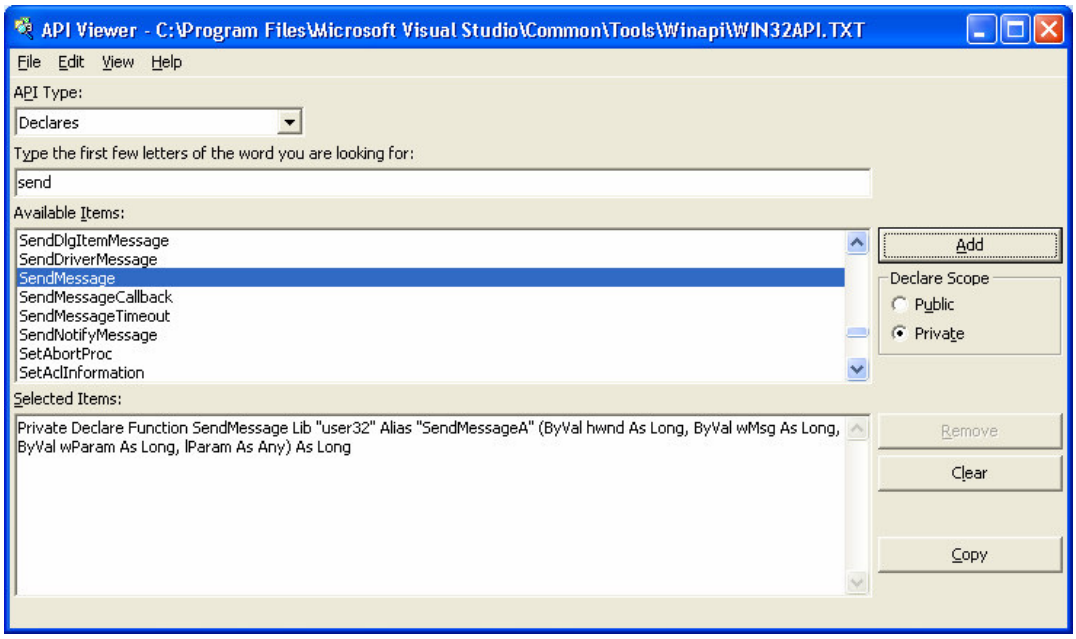
3.3 IMPLEMENTATION - TALKING TO THE DEP

The first requirement of such an application is that it be able to send information to the DEP in such a way as to simulate keyboard input. The basic concept is to transmit each new keystroke read from the .adk file to the DEP, mimicking the original test or interview. There are a number of ways to do this in Visual Basic 6; this section will touch on a few alternatives, but will focus mostly on a couple of useful Windows API commands.

The simplest way to simulate a keystroke in Visual Basic 6 is to use the `SendKeys` statement, which is implemented as follows: `SendKeys ("String to be sent")` This command will send any keystrokes given as an argument to the active window. However, the `SendKeys` statement has a number of shortcomings and eccentricities that make it problematic. The most important of these is that it requires the receiving window to be the active window, necessitating tight control over window switching. This limitation makes the application much less flexible, particularly insofar as there may be cases when it is necessary to alter the entries in the DEP manually during playback in order to test alternate scenarios or to compensate for the sort of inherent problems mentioned earlier that occasionally appear.

In order to bypass the problems of the `SendKeys` statement, it is necessary to use the Windows API, which is, in fact, the mechanism being used behind the scenes by `SendKeys` as well. Making direct use of it allows for greater flexibility and control in the production of simulated keystrokes. What is more, its use is required for other functions of the "Replayer."

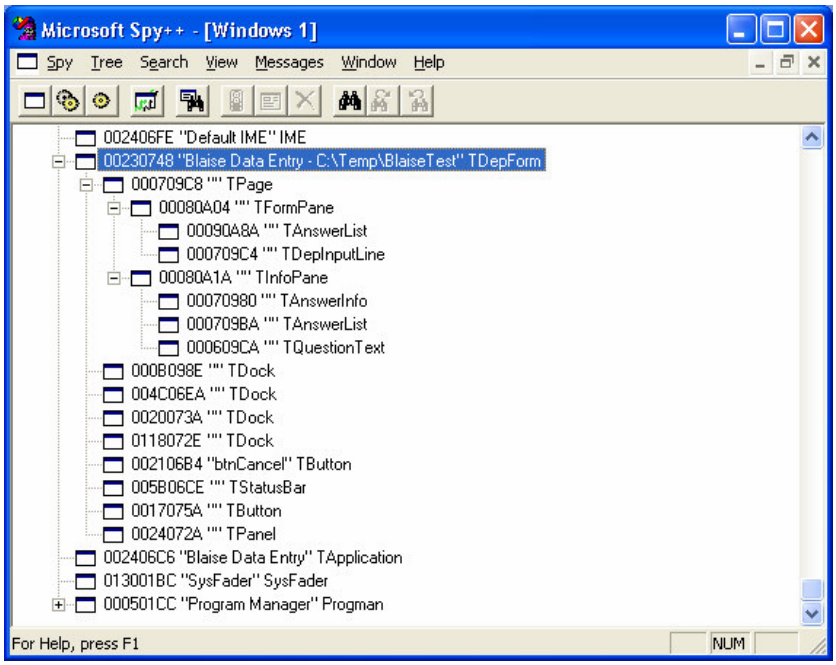
At this point, it may be worthwhile to review the basics of the Windows API and its use in Visual Basic 6 briefly. An API (or Application Programming Interface) is a collection of functions and properties of an operating system or application exposed for use by other developers. For example, BCP is an API for Blaise. The Windows API, which is of interest here, exposes much of the internal functionality of Windows to the developer. For purposes of the "Replayer," it offers direct access to the message systems upon which Windows functions. To make use of the API in Visual Basic 6, it is necessary to declare the API function in the general declarations section, and then it is possible to call the function just as a normal function. To find the correct declaration for an API function, launch the following utility (from the Start menu, assuming Visual Studio 6 is installed): Start > All Programs > Microsoft Visual Studio 6.0 > Microsoft Visual Studio 6.0 Tools > API Text Viewer (see illustration).



API Text Viewer

Using the Windows API, keystrokes can be simulated using the `keybd_event` function or the `SendInput` function which superseded it in NT-based Windows operating systems. However, these pose similar problems to `SendKeys` in terms of windows switching. Therefore, this paper will proceed directly to the `SendMessage` function and its variants, which are used by the HRS "Replayer" program.

Here, again, it may be useful to pause briefly to consider the relationship of these functions to the internal processes of Windows. A messaging mechanism in the operating system controls communication with the windows that appear on the desktop. For example, when a key is pressed, a message is generated by the operating system that is sent to a queue of messages associated with the active window. When the window receives the message, it will learn that a particular key was pressed and may take an appropriate action, such as displaying it on the screen. Each apparent window may also contain a number of other windows within. For example, the DEP contains a number of window objects, including the DEPInputLine, which will be the focus of much of this section. Each window object is referenced by a "handle," which will be an argument passed to many of the procedures described below. Note that the handle for an application will differ each time it is run; it only stays unchanged while the application remains open. The list of windows and their handles, and the associated message queues can be viewed with the following utility, which will be referred to again later: (from the Start menu): Start > All Programs > Microsoft Visual Studio 6.0 > Microsoft Visual Studio 6.0 Tools > Spy++ (see illustration).



Microsoft Spy++

The `SendMessage` function provides a way to send a message (containing a key to be processed) directly to the desired window, even if it is hidden. This is accomplished by passing as arguments of the function the handle of the window, the type of message, and a value indicating the desired key. First, the handle of the receiving window must be determined. The following code can be used to retrieve the handle of the DEP window and then of the DEPInputLine window. (From this point forward, where code is presented as follows, it can be considered to be part of a sample project referred to by the default name of "Project1" that will demonstrate the most important mechanisms of the "Replayer.")

[Put this code in a module]

```
Public Declare Function EnumWindows _
  Lib "user32" _
  (ByVal lpEnumFunc As Long, ByVal lParam As Long) As Boolean
Public Declare Function GetWindowText _
  Lib "user32" Alias "GetWindowTextA" _
  (ByVal hwnd As Long, ByVal lpString As String, _
  ByVal cch As Long) As Long
Public Declare Function GetWindowTextLength _
  Lib "user32" Alias "GetWindowTextLengthA" _
  (ByVal hwnd As Long) As Long
Public gsWndTxt As String
Public gsWndFind As String

Public Function EnumWindowsProc(ByVal lhwnd As Long, _
 ByVal lParam As Long) As Boolean
  Dim sGetWndTxt As String
  Dim lRet As Long
  lRet = GetWindowTextLength(lhwnd)
  sGetWndTxt = Space(lRet)
  GetWindowText lhwnd, sGetWndTxt, lRet + 1
  If sGetWndTxt <> "" Then
    If InStr(1, sGetWndTxt, gsWndFind) Then
```

```
      gsWndTxt = Trim$(sGetWndTxt) 'get exact window text
    End If
  End If
  EnumWindowsProc = True 'continue enumeration
End Function
```

[Put this code in a form]

```
Private Declare Function FindWindow _
  Lib "user32" Alias "FindWindowA" _
  (ByVal lpClassName As String, ByVal lpWindowName As String) As Long
Private Declare Function GetWindow _
  Lib "user32" _
  (ByVal hwnd As Long, ByVal wCmd As Long) As Long
Private Declare Function GetClassName _
  Lib "user32" Alias "GetClassNameA" _
  (ByVal hwnd As Long, ByVal lpClassName As String, _
  ByVal nMaxCount As Long) As Long
Private Const GW_CHILD = 5
Private Const GW_HWNDNEXT = 2
Private mlhWnd As Long
Private mlhInputLine As Long

Private Sub FindWindowHandle(sFind As String, Optional sChildFind As
String)
  gsWndFind = sFind
  gsWndTxt = ""
  EnumWindows AddressOf EnumWindowsProc, ByVal 0&
  If gsWndTxt <> "" Then
    mlhWnd = FindWindow(vbNullString, gsWndTxt)
    GetChildHandle mlhWnd, sChildFind, mlhInputLine
  Else
    mlhWnd = 0
    mlhInputLine = 0
  End If
End Sub

Private Sub GetChildHandle(lhwnd As Long, sFind As String, _
 ByRef lhwndChld As Long)
  Dim lChldWnd As Long
  Dim sChldTxt As String * 250
  Dim sGetWndTxt As String
  Dim lRet As Long
  lChldWnd = GetWindow(lhwnd, GW_CHILD)
  If lChldWnd <> 0 Then
    Do Until lChldWnd = 0
      GetClassName lChldWnd, sChldTxt, 250 'get child window's
classname
      lRet = GetWindowTextLength(lChldWnd)
      sGetWndTxt = Space(lRet)
      GetWindowText lChldWnd, sGetWndTxt, lRet + 1
      If InStr(1, sChldTxt, sFind) > 0 Then
        lhwndChld = lChldWnd 'find class name like TDEPInputLine
      ElseIf InStr(1, sGetWndTxt, sFind) > 0 Then
        lhwndChld = lChldWnd
      End If
      GetChildHandle lChldWnd, sFind, lhwndChld
```

```
      lChldWnd = GetWindow(lChldWnd, GW_HWNDNEXT) 'Search next child
    Loop
  End If
End Sub

Private Sub Command1_Click()
  'Make a button and three text boxes for this Sub
  FindWindowHandle "BlaiseTest", "TDepInputLine"
  'the argument "Blaise Test" in the line above should be the optional
  'text description of the datamodel or part of the datamodel filename
_
  'it simply has to be part of the text that appears in the title bar
of
  'the DEP window which distinguishes it from other windows
  Text1 = gsWndTxt 'full title bar of DEP Window
  Text2 = mlhWnd 'handle of DEP window
  Text3 = mlhInputLine 'handle of DEP Input Line
End Sub
```

Once the handle of the DEPInputLine has been established for a given session, it is possible to make values appear in it by using the `SendMessage` function. The declaration for the Windows API `SendMessage` function is as follows. Additionally, a few helpful constants are specified as well.

[Add this code to the declarations section (the top) of the form]

```
Private Declare Function SendMessage _
  Lib "user32" Alias "SendMessageA" _
  (ByVal hwnd As Long, ByVal wMsg As Long, _
  ByVal wParam As Long, lParam As Long) As Long
Private Const WM_CHAR = &H102
Private Const WM_KEYDOWN = &H100
Private Const WM_KEYUP = &H101
```

Then it is quite simple to send a character to the DEP using the `SendMessage` function. An example of this follows. Note that it makes use of the `WM_CHAR` constant to send a character to the DEPInputLine window and that it requires an Ascii value representing the character to be sent.

[Add this code to the form]

```
Private Sub Command2_Click()
  'Make another button for this Sub
  Dim s As String
  s = "1" 'This is the character to be sent to the DEP
  Call SendMessage(mlhInputLine, WM_CHAR, Asc(s), 1)
  'Arguments:
  'mlhInputLine - the handle of the DEPInputLine
  'WM_CHAR - a constant that indicates a character message
  'Asc(s) - the Asc() function turns the string, s, into an Ascii value
  '1 - the last argument indicates one repetition - see MSDN for a
list:
  'go to http://msdn.microsoft.com and search for WM_CHAR
End Sub
```

It is also quite easy to send non-character keystrokes, like a carriage return, a tab or an arrow key. In this case, the `WM_KEYDOWN` and `WM_KEYUP` constants can be used in succession to simulate a

press and a release of a key. Here, the arguments for the `SendMessage` function include a virtual key constant representing the non-character key to be pressed. The code that follows is an example of this.

[Add this code to the declarations section (the top) of the form]

```
Private Const VK_RETURN = &HD 'virtual key constants
Private Const VK_UP = &H26
Private Const VK_DOWN = &H28
Private Const VK_LEFT = &H25
Private Const VK_RIGHT = &H27
Private Const VK_TAB = &H9
Private Const VK_HOME = &H24
Private Const VK_END = &H23
Private Const VK_PRIOR = &H21 'Page Up
Private Const VK_NEXT = &H22 'Page Down
Private Const VK_DELETE = &H2E
Private Const VK_BACK = &H8
Private Const VK_CLEAR = &HC 'NumPad 5 with NumLock off
Private Const VK_SHIFT = &H10
Private Const VK_ESCAPE = &H1B
Private Const VK_CAPITAL = &H14 'Caps Lock
```

[Add this code to the form]

```
Private Sub Command3_Click()
  'Make another button for this Sub
  Dim s As String 'string representing non-character key to send
  Dim iKey As Integer 'counter of keys pressed in the current field
  Dim lVK As Long 'virtual key constant to send
  Dim lhwnd As Long 'window handle to receive message
  lhwnd = mlhWnd 'mlhwnd is the handle of the DEP window
  s = "[TAB]" 'would normally be passed in as an argument
  iKey = 1 'would normally be passed in as an argument
  Select Case s
    Case "[ENTR]":
      lVK = VK_RETURN
    Case "[BACK]":
      lVK = VK_BACK
      lhwnd = mlhInputLine 'this key should go to the input line handle
    Case "[DEL]":
      lVK = VK_DELETE
      lhwnd = mlhInputLine 'this key should go to the input line handle
    Case "[TAB]":
      lVK = VK_TAB
    Case "[DOWN]":
      lVK = VK_DOWN
    Case "[UP]":
      lVK = VK_UP
    Case "[LEFT]":
      lVK = VK_LEFT
      If iKey <> 1 Then 'If it is not the first keystroke on a line,
        lhwnd = mlhInputLine 'this key should go to the input line
handle
      End If
    Case "[RIGHT]":
      lVK = VK_RIGHT
      If iKey <> 1 Then 'If it is not the first keystroke on a line,
```

```
        lhwnd = mlhInputLine 'this key should go to the input line
handle
      End If
    Case "[HOME]":
      lVK = VK_HOME
    Case "[END]":
      lVK = VK_END
    Case "[PGUP]":
      lVK = VK_PRIOR
    Case "[PGDN]":
      lVK = VK_NEXT
    Case "[ESC]":
      lVK = VK_ESCAPE
    Case "[CAPS]":
      lVK = VK_CAPITAL
  End Select
  If lVK <> 0 Then 'Send Virtual Key to message queue
    Call SendMessage(lhwnd, WM_KEYDOWN, lVK, 0)
    Call SendMessage(lhwnd, WM_KEYUP, lVK, 0)
  End If
  If s = "[BACK]" Or s = "[DEL]" Then
    'This fixes a problem with nonresponse after sending Delete
    Call SendMessage(lhwnd, WM_KEYDOWN, VK_CLEAR, 0)
    Call SendMessage(lhwnd, WM_KEYUP, VK_CLEAR, 0)
  End If
End Sub
```

With the information outlined above, it is possible to construct a program which reads the .adk file, locates the DEP window, and sends the keystrokes recorded within the .adk file to the DEP one by one. This assumes, of course, that the DEP is prepared to receive the keys in the manner and in the order specified by the .adk file. The next section will discuss ways to get information from the DEP about its status, such as the currently focused field.

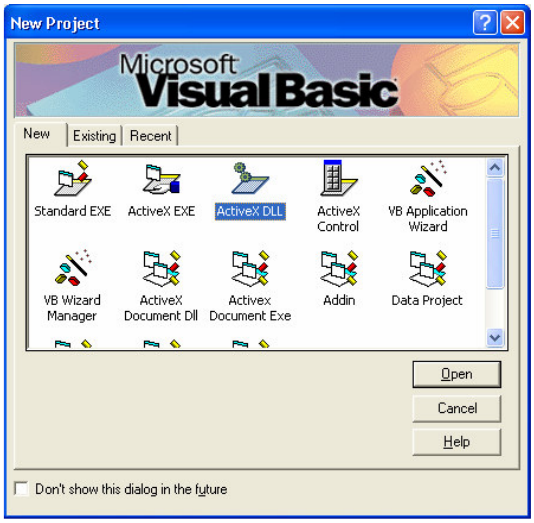
Finally, there are other potential options. It would be interesting to consider a program that recorded keystrokes and mouse movements separately from the DEP, forming its own trail for playback that could be better tailored to the specific needs of the utility. The fundamental problem for that hypothetical utility, and, indeed, for the one under consideration here, is that the keystroke information has to be matched to field information from the DEP in order for replay to occur in a well-orchestrated fashion. A solution to this problem will be covered in detail in the next section.

3.4 IMPLEMENTATION - BUILDING A TWO-WAY BRIDGE

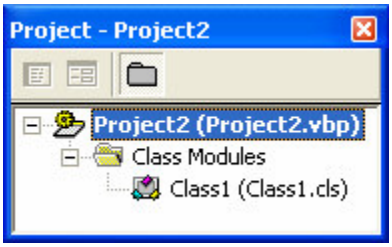
At this point, a system for receiving information from the DEP is needed. There are a variety of ways to build a communication bridge between the “Replayer” program and the DEP. This paper will present three variants in order of performance. All three are based on the same basic scheme, but differ in the way that they return data from the DEP to the “Replayer” program. The first method to be discussed was not implemented by HRS because it seemed too slow, but it has the virtue of being quite simple to implement and it lays a nice foundation for understanding the other two. The latter two methods have been used by the HRS “Replayer,” with the last one being in current use.

The basic scheme is to add a menu entry to the DEP which can call an external procedure to provide information about the status of the DEP. This menu item can be activated by the "Replayer" program as needed. Currently, the preferred type of external procedure call utilizes COM objects in ActiveX libraries. These are longstanding Microsoft technologies that are currently obsolescent, but which are not difficult to use with Visual Basic 6. This section will discuss building a simple ActiveX dynamic link library (DLL) which will provide information about the DEP. Then the discussion will turn to making use of the DEP Menu Manager (found under tools in the Blaise Control Centre) to add a menu entry which will invoke the COM object method in the ActiveX DLL. Finally, the discussion will cover how to activate the DEP menu entry from the Sample project covered in the previous section and how to receive the information which this process makes available. (For more information about using COM with Blaise, search for "calling COM DLL" in Blaise Help.)

For this discussion, a second Visual Basic project will be referenced. This project will be an ActiveX DLL project and will be named "Project2." The name is important because it will be used by the Blaise Menu Entry that will be added. This project will also need to reference BCP.



Start an ActiveX DLL Project…



Rename the project to "Project2"…

The Active X DLL should have at least two procedures. One of these should be of public scope so as to be able to be called directly by the Blaise Menu Entry. The other should be a supporting function, called from the first, which will transmit the information. This second procedure can initially be set up to transmit the information by the simplest means possible, which is to write a text file in a predetermined location.

[Put this code in the class]

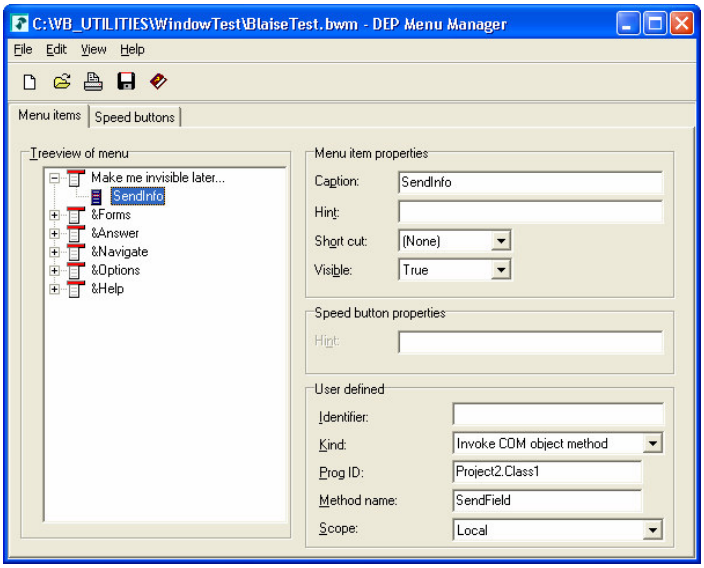
```
Public Sub SendField(db As BlAPI4A2.Database, ds As BlAPI4A2.DepState)
  'Don't forget to select "Blaise API Objects Library 2.0"
  'under Project > References...
  'See Blaise Help for more information about this code
  Dim Quest As Question
  Dim Layout As LayoutSet
  Dim Parallel As Parallel
  Dim Page As StoredPage
  'Get hold of the currently active question...
  'The properties of the DepState parameter need to be used for this...
  Set Layout = db.Screens.LayoutSetCollection(ds.LayoutSetIndex)
```

```
  Set Parallel = Layout.ParallelCollection(ds.ParallelIndex)
  Set Page = Parallel.StoredPageCollection(ds.StoredPageIndex)
  Set Quest = Page.QuestionCollection(ds.QuestionIndex)
  'send name of current field
  SendData Quest.Field.Name
End Sub

Private Sub SendData(sData As String)
  Open "C:\DEPInfo.txt" For Output As #1
  Print #1, sData
  Close #1
End Sub
```

When the code for this DLL is compiled (File > Make Project2.dll…), it will automatically be registered for use in its initial location. If it is later moved or installed on another machine, it will need to be registered in its new location. This can be done by accessing a command prompt in that location and typing "regsvr32 Project2.dll" at the prompt.

The DEP Menu Manager makes possible the generation of a .bwm file which can add new menu items to the DEP. To proceed with this example, select "Add Menu Entry" and enter a name for the new menu entry in "Menu item properties." Then set the values in the "User defined" segment as follows: Kind = Invoke COM object method; Prog ID = Project2.Class1; Method Name = SendField (see illustration).

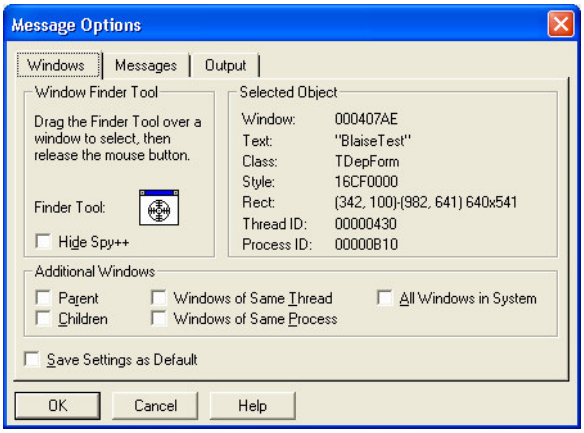


DEP Menu Manager

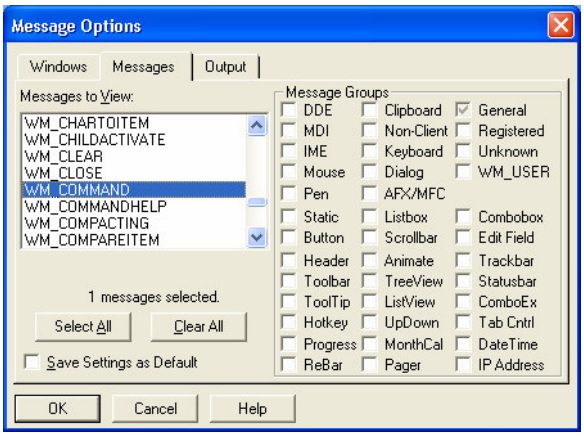
After saving and exiting the DEP Menu Manager, select the menu file under Run > Parameters in the Blaise Control Centre or specify it in the command line parameter of the DEP. This will allow the new menu entry to appear in the DEP when it is run.

At this point, it is necessary to run the DEP with the new menu file. When the DEP is running with the new menu option, start the program called Spy++ which was referred to in the previous section. Open the Message Options dialog (Spy > Messages) from the menu. This will cause a dialog box to appear with a target-shaped cursor called the Finder Tool. Click and drag on the finder tool to pull it over the DEP title bar and then release. The information in the Selected Object segment should appear as in the illustration below (with different numbers). Now that the window has been selected, switch to the messages tab and press "Clear All." Then select WM_COMMAND in the

Messages to View list and press the OK button to close the dialog. This will make it easy to see only the WM_COMMAND message that is of interest.



The Finder Tool



Message Options in Spy++

After closing the Message Options dialog, switch to the DEP window and select the new menu entry; incidentally, this action should produce the C:\DEPInfo.txt file containing the currently focused fieldname. Switch back to Spy++ and record the ID of the menu item (see the illustration below). After all of this has been done, it is possible to make the menu entry invisible in the .bwm so that the user cannot activate it unintentionally.

The ID of the menu item as seen in Spy++ (here the ID is 136)

At this point, the menu entry can be activated from the "Replayer" program whenever the program needs to know what field is currently active in the DEP. For this operation, the windows API function that should be used is called `PostMessage` and the message type is `WM_COMMAND`. Internally, `PostMessage` operates slightly differently than `SendMessage`, but it still takes the window handle as an argument. The following code examples are intended to be added to the sample project called "Project1" from the preceding section.

[Add this code to the declarations section (the top) of the form]
```
Private Declare Function PostMessage _
  Lib "user32" Alias "PostMessageA" _
  (ByVal hwnd As Long, ByVal wMsg As Long, _
  ByVal wParam As Long, ByVal lParam As Long) As Long
Private Const WM_COMMAND = &H111
'ID of menu item for calling ActiveX dll (Get from Spy++)
Private Const wID_SendInfo = 136
```

Two procedures are given below. The first activates the new menu entry when the DEP is running. The second simply loads and displays the fieldname that is saved when the first procedure is executed.

[Add this code to the form]
```
Private Sub Command4_Click()
  'Make another button for this Sub
  Call PostMessage(mlhWnd, WM_COMMAND, CLng(wID_SendInfo), 0)
End Sub

Private Sub Command5_Click()
  'Make another button and another text box for this Sub
  Dim s As String
  Open "C:\DEPInfo.txt" For Input As #1
  Line Input #1, s
  Close #1
  Text4 = s
End Sub
```

This setup is clearly quite inefficient and lacks automation, but there are many ways to automate the process. For example, a loop could be added to the first procedure which would wait for the date on the file to change (indicating that new data had arrived) before loading the data from the

file. However, there are even better ways to automate this process and they will be described below.

The beginning of this section announced that three different methods would be presented. The first has already been covered and uses a text file to return the data to the calling program. The other two methods operate on the same principle, but offer a more complicated yet more efficient means of returning the data.

3.4.1 IMPLEMENTATION - BUILDING A TWO-WAY BRIDGE – RETURN METHOD 2

The second method of returning data is to send it to a hidden textbox in the "Replayer" program; this was used in the HRS "Replayer" for about a year and a half and worked quite well. In this method, the data is sent to a textbox in a hidden form in the "Replayer" program by means of a `WM_SETTEXT` message.[6] The Change event of the textbox will fire when the data arrives, at which point the textbox contents can be used to check the alignment of the DEP and the .adk. When the whole process first begins with the message to activate the menu item, the main program is set to wait until this data shows up before it proceeds.

As before, the window handle of the receiving object must be determined, and the same code can be used as shown above, except that it will search for the name of the hidden form and textbox to receive data. The message can be sent using the `SendMessage` function as follows:

```
SendMessage(longHandleOfTextBox,       WM_SETTEXT,       0,       ByVal
stringDataToBeSent)
```

On the hidden form, the Change event of the textbox can be set to do something with the data, such as change the value of a global variable. If the textbox is called Text1 then the `Text1_Change` event can also include this line: `Call ReplyMessage(1)` This requires reference to the `ReplyMessage` API function in the declarations section:

```
Private Declare Function ReplyMessage _
  Lib "user32" _
  (ByVal lReply As Long) As Long
```

3.4.2 IMPLEMENTATION - BUILDING A TWO-WAY BRIDGE – RETURN METHOD 3

For the past several months, HRS has been using a new system to send data back to the "Replayer" using `WM_COPYDATA`. This process makes use of undocumented pointer functions in Visual Basic to copy a data structure and is based on *Microsoft Knowledge Base Article #176058 - How To Pass String Data Between Applications Using SendMessage*.[7] Please note the warning about using undocumented functions. The article covers the needed declarations and retrieval code that should be put in a module in the "Replayer," but, as an example, the code for the send procedure in the DLL might look like this:

```
Private Sub SendData(sData As String)
  Dim cds As COPYDATASTRUCT
  Dim buf(1 To 255) As Byte
  Dim l As Long
  Call CopyMemory(buf(1), ByVal sData, Len(sData))
  cds.dwData = 3
  cds.cbData = Len(sData) + 1
  cds.lpData = VarPtr(buf(1))
  lhWnd = FindWindow(vbNullString, "Title of Target Replayer Window")
  l = SendMessage(lhWnd, WM_COPYDATA, 0, cds)
End Sub
```

[6] For further discussion of this method, see http://www.thescarms.com/vbasic/PassString.asp
[7] See http://support.microsoft.com/default.aspx?scid=kb;en-us;176058

This method seems to operate considerably faster than the ones previously described and has worked well in testing up to this point.

3.5 IMPLEMENTATION – CLOSING ERROR DIALOGS

Most Blaise instruments will incorporate some consistency checks. These, and the messages that appear, for example, when an invalid value is input, have to be handled if replay is to proceed without the need for constant supervision. This can easily be accomplished by sending a `BM_CLICK` message to the handle of the button which will close the dialog. In order to determine when to send this message, a timer can be set up to call a procedure that will search for known error dialog types using code similar to that used to find the DEP and its handle. This needs to happen asynchronously, so HRS initially set up a small external executable, the sole function of which was to find and close error dialog boxes. This executable was loaded to coincide with the start of the process and was terminated at the end of replay. Later, as the need to return information about the error dialogs surfaced, HRS replaced the separate program with an object in an ActiveX .exe which could be referenced directly. This object performs the same function as the separate executable described above but also exposes information about its activity so that the "Replayer" program is aware of any attempts to close a dialog. This solves certain occasional problems that arise during replay in which the "Replayer" needs to know whether certain keystroke combinations were caused by an error dialog.

Despite the apparent complexity of the "Replayer" due to the number of operations that must be taken into account, the amount of program code needed is rather small. Neither is it terribly time consuming or difficult to program.

4. RESULTS

The purpose of developing this application was to improve data quality while reducing cost and increasing testing efficiency. During the year and a half that the "Replayer" program has been in use, HRS has seen a marked decrease in the amount of time and effort expended on investigating and solving problems as part of the instrument development process. In 2003 and 2004, the principal users of this application were programmers who used it to replay both tester and field .adks. It proved capable of significantly reducing the amount of time programmers spent on diagnostic activities, such as reproducing problems, and drastically reduced the amount of time needed to retest fixes. Programmers found that they were able to quickly replay keystroke files that they received from the testers who reported problems in order to verify the existence of an error being reported. After making changes to the Blaise code intended to fix an error, programmers were able to use the "Replayer" again to check the effectiveness of the fixes.

The successful incorporation of this program into the testing process is reflected in the fact that the number of reported errors from the field has declined by more than half against a comparable period in the previous wave. As an added bonus, the "Replayer" has proven to be an effective tool in reviewing interviews exactly as they were taken and has substantially reduced the time it takes to investigate problems reported from the field and from the data review process. This ability not only reduces the time involved in program corrections, but also in reviewing issues of Interviewer performance. For example, at the time of this writing, the program had just been used to replay a couple of cases in which the interview was conducted following the wrong path (e.g. Self-R instead of proxy; living R instead of deceased) for the situation. In these cases, it was necessary to examine the incorrect flow to diagnose the situation and to determine whether the interviews in question needed to be retaken. More generally, this program has helped HRS to spot less than optimal interviewer behavior, such as backing up and changing fields from earlier sections. Sometimes this leads to improvements in the field process that may result in better data being collected.

5. CONCLUSION

Plans to expand the use of this utility have already been made for the 2006 wave of HRS. HRS is planning on incorporating this utility fully into the overall testing process and is developing a database for specific scenarios that will capture scenario attributes and matching ADK files. Testers will then be allowed to select and replay a scenario at any phase of testing and check results against a number of saved formats.

The success HRS has had in using this utility would seem to be applicable to other studies as well. It is hoped that a discussion of this program will generate interest in this aspect of instrument development.