A Bevy of Timers

Whil Hentzen

Visual FoxPro applications left open while a user is away can cause a couple of problems. One is the display and access of confidential data. In these days of HIPAA and other strict regulation about who can see what data, it's more important than ever to be able to restrict access granularly. Another reason is needing the ability to run system utilities that require exclusive access of files; if an application leaves files open, those utilities can't be run.

Application timeouts are generally better left to the operating system to handle, but there are still situations where it can be advantageous to have VFP do the work instead. And it can be difficult, if not impossible, for the operating system to close individual forms in VFP.

In this article, I'll explore several ways to have VFP time out an application as well as to time out a single form.

Better yet, because examples of these methods on the Web are typically provided as code snippets, left up to you to decipher and implement, I'll also provide a simple, yet complete framework that demonstrates the use of each mechanism in concert with an event loop, a menu, and a pair of working forms.

Application level vs form level timouts

Timeouts can be implemented on either an application level, a form level, or both.

An application level timeout can be implemented in a couple of different ways. In each case, care has to be taken to handle open forms; if they are open, deal with whether there are pending edits in those forms, and, finally, if there are processes such as reports, file copying, or lengthy queries running at the moment the timeout executes.

A form level timeout that works 99% of the time is easy to implement using a VFP timer. It, too, though, should deal with the possibility of a pending edit (one would hope a user wouldn't walk away from a form in the middle of an edit, but it happens, particularly if they're off hunting down info to enter) or processes launched by the form and now running in the background.

Individual concerns aside, you'll also want to consider and plan a strategy for the situation when you have both application level and form level timeout mechanisms in place. For example, you could configure your system to have one timeout value for the application and a second one for forms. (It would also be possible to have individual values for each form, but the issues to deal with would be similar as having a single value for all forms.)

So suppose you've got one timeout value for the application and another for your forms. You need to decide how they work in concert with each other - do they run in parallel or in serial?

Parallel timers count down at the same time - as soon as a form is opened, the application level timer restarts (because of user activity) and the form level timer starts running as well. So, if the application level timeout value is ten minutes and the form level timeout value is four minutes, the form will close in four minutes and the application will close six minutes after that.

Serial timers, on the other hand, run one after the other. Once a form is opened, that timer starts, but the application level timer halts. Once the form timeout finishes and the form closes, the application timer starts again. So in the previous example, once the form closes, after four minutes, the application timer will then count down, and close ten minutes later, not just six.

Obviously, with parallel timers, you need to take care of the situation where the application level timeout value is shorter than the form level timeout value. If the form timeout is six minutes but the app level is five minutes, the app will timeout before the form closes. That may be acceptable, indeed, it may be by design (not sure why, but who am I to question another's design?), but if not, you need to deal with it. Either prohibit the situation from happening in the first place by testing the timeout values and changing one of the values upon application startup, or dealing with the possibility of the application closing while forms are open.

Let's take a look at a couple of methods to implement an application level timeout mechanism first, and then address a form timer next.

Application level timeouts

There are a couple of popular methods to time out on the application level, one using VFP 9's BindEvents mechanism to hook into the Windows API (described by Christof Wollenhaupt in a blog article of his), another only using the Windows API, described by Stefan Wuebbe on www.foxite.com. Gather a half-dozen developers together and they'll proffer seven opinions on why one is better than the other; I'll leave that decision to you and your proclivities.

They both work the same general way - setting a stake identifying the last time activity occurred, then checking the current time to see if the timeout span has passed since that last activity. The way that the last activity is detected is what differs between the two mechanisms.

The BindEvents mechanism has more pieces to it, so I'll discuss the Windows API approach first.

The sample code provided with this article uses a single PRG that implements both mechanisms, and uses a flag to drive which one is executed. The flag can be passed as a parm to the PRG and defaults to using BindEvents.

```
* parm of IDLE starts WinAPI mechanism
(Stefan's)
* parm of INACTIVE starts BindEvents mechanism
(Christof's)

lparameters lcWhichTimer
if pcount() < 1
  lcWhichTimer = 'INACTIVE'
endif
```

The PRG contains both a main program that sets up the application as well as the class definitions for the application object and the two timers. It also creates a global object for the application level timer. Which timer class was used was driven by the flag:

```
case upper(lcWhichTimer) = 'IDLE'
  goTmr=CreateObject("IdleTimer")
case upper(lcWhichTimer) = 'INACTIVE'
  goTmr = createobject("InactivityTimer", 0.1)
```

The class definition for the application object also contains a couple of properties that define the timeout values

```
iSecondsForFormTimeout = 15
iSecondsForDialogTimeout = 15
iSecondsForAppTimeout = 10
```

and a flag that allows you to turn debug statements on and off at one centralized location.

```
lShowTimerDebug = .f.
```

Using the Windows API

Stefan Wuebbe answered a post on foxite.com regarding this issue with a solution that queried the Windows API to detect user activity inside a VFP application.

http://www.foxite.com/archives/vfp-code-to-know-if-computer-is-idle-0000301035.htm

Implementing a timer with the Windows API is straightforward, although it requires a bit of magic if you're not experienced with the interface or syntax.

The main program needs just one line, the call to instantiate the class that's defined later in the same PRG. After instantiation, the event loop was started, and the application level timer is running.

The class definition is very straightforward.

```
**********************************************
Define Class IdleTimer as Timer
* Stefan Wuebbe
**********************************************
* Interval, LastInput and CurrentTime
* are all in milliseconds
* Timeout is in seconds

Interval=1000 && check every second
TimeOutInSeconds = goApp.iSecondsForAppTimeout

procedure Init
Declare Integer GetLastInputInfo ;
    in win32api string @
Declare Long GetTickCount ;
    in win32api
endproc

123456789_123456789_123456789_123456789_12345
procedure Timer
with this
Local lcBuf, lnLastInputMS, lnCurrentTimeMS
lcBuf = BinToC(8,'4rs')+BinToC(0,'4rs')

* last input, from winapi, defined in caller
GetLastInputInfo(@lcBuf)
lnLastInputMS ;
  = CToBin(Substr(lcBuf,5,4),'4rs')

* current time, from winapi, defined in caller
lnCurrentTimeMS=GetTickCount()

* developer feedback
goapp.debugox(goApp.lShowTimerDebug, ;
  'goTmr.timer() lnLastInput', ;
  lnLastInputMS, ;
  ': lnCurrentTimeMS:', lnCurrentTimeMS, ;
  ': .TimeOutInSeconds:', ;
  .TimeoutInSeconds*1000, ':')
_screen.Caption '
  = "Oh Hi! I started at " ;
  + transform(lnLastInputMS/1000) ;
  + ". I'm going to quit at " ;
  + trans(lnLastInputMS/1000 ;
  + .TimeOutInSeconds) ;
  + " and it is currently " ;
  + trans(lnCurrentTimeMS/1000)

* if the last input (a few minutes ago)
* + the timeout span is earlier than now,
* QUIT
if lnLastInputMS+(.TimeOutInSeconds*1000) ;
  < lnCurrentTimeMS
  goapp.debugox(goApp.lShowTimerDebug, ;
    'goTmr.timer() lnLastInputMS ';
    + '.TimeOutInSeconds < lnCurrentTimeMS';
    + '-> so... quitting!')
  if messagebox( ;
    "Idle Timer timeout! Quit?",4)=6
    goApp.quit()
  else
    messagebox("Not quitting!")
  endif
endIf
endwith
endproc

enddefine
**************************************
```

This mechanism uses the VFP timer class in concert with the Windows API to query when Windows events (keyboard presses and mouse moves) occurred. A memvar to be passed to the GetLastInputInfo API call is initialized. Then the last time of an event is stored to lnLastInputMS via the getLastInputInfo function, and immediately afterward, the current time is stored to lnTime via the GetTickCount() WinAPI function.

Note that the values in the lnLastInputMS and lnCurrentTimeMS memvars have precision to approximately a thousandth of a second. For example, the values captured would look like this:

```
lnLastInputMS      7381072020
lnCurrentTimeMS    7381213670
```

So the value of the time out span (defined as a number of seconds) has to be multiplied by a thousand to match significant digits.

Then the time out span is added to the last time and the net is compared to the current time. If the current time is greater, that means the allowed time span has passed. In this sample application, I update the _screen caption so you can see what's going on, send off countdown messages to the Debug Output window, and display a messagebox to indicate the timeout period has passed before calling the app's shutdown method. In an actual implementation, you'll want to remove all three.

Using BindEvents()

Back in 2009, Christoph Wollenhaupt wrote an article about using BindEvents to time out an application.

```
http://www.foxpert.com/knowlbits_200903.htm
```

Unfortunately, the article just showed a snippet of code, leaving 'the implementation left to the reader', and since this implementation was at first glance more complex than just instantiating a class, I decided that a broader article was in order.

On the other hand, all calculations are done with seconds; no translating from milliseconds needed. Yes, it's just arithmetic, but why add more complexity when not needed?

As with Wuebbe's, the main program needs just one line, the call to instantiate the class that's defined later in the same PRG. The complexity arose with the class itself, as it had more pieces than just a single Timer method like the WinAPI mechanism.

Instantiation of the InactivityTimer class, after creating a few properties, sets up the magic in the init(), via two BindEvent() calls:

```
  BindEvent(0,WM_KEYUP,This,"WndProc")
  BindEvent(0,WM_MOUSEMOVE,This,"WndProc")
```

The first traps keypresses while the second intercepts mouse movements. In each case, when one of those events occurs, the WndProc() method assigns the current values of seco() to the nLast Activity timer property.

The Timer method of the class (this is a class based on the VFP Timer class, after all), does a very similar thing as the WinAPI mechanism - compares the combination of the last activity and the timeout value span to the current time, and if the timeout span has been exceeded, calls a method to handle it.

```
*=============================================
* Detects user activity and fires an event
* after the specified period of inactivity.
* Christof.Wollenhaupt @ foxpert.com
*=============================================
Define Class InactivityTimer as Timer

*--------------------------------------------
* API constants
*--------------------------------------------
#define WM_KEYUP                         0x0101
#define WM_SYSKEYUP                      0x0105
#define WM_MOUSEMOVE                     0x0200
#define GWL_WNDPROC         (-4)

*--------------------------------------------
* internal properties
*--------------------------------------------
nTimeOutInSeconds =
goApp.iSecondsForAppTimeout
nLastActivity = 0
nOldProc = 0

*--------------------------------------------
* Timer configuration
*--------------------------------------------
Interval = 1000 && in MS, 1000 = 1 second
Enabled = .T.

*--------------------------------------------
* Listen to API events when the form starts. *
You can pass the timeout as a parameter.
*--------------------------------------------
Procedure Init(tnTimeOutInSeconds)
    DECLARE integer GetWindowLong IN
WIN32API ;
      integer hWnd, ;
      integer nIndex
    DECLARE integer CallWindowProc IN WIN32API
;
      integer lpPrevWndFunc, ;
      integer hWnd,integer Msg,;
      integer wParam,;
      integer lParam
    THIS.nOldProc = ;
      GetWindowLong(_VFP.HWnd,GWL_WNDPROC)

  If Vartype(m.tnTimeOutInSeconds) == "N"
    This.nTimeOutInSeconds ;
      = m.tnTimeOutInSeconds
  EndIf

  This.nLastActivity = seco()
  BindEvent(0,WM_KEYUP,This,"WndProc")
  BindEvent(0,WM_MOUSEMOVE,This,"WndProc")

EndProc

*---------------------------------------------
* Stop listening
*---------------------------------------------
Procedure Unload
  UnBindEvents(0,WM_KEYUP)
  UnBindEvents(0,WM_MOUSEMOVE)
EndProc

*---------------------------------------------
* Every event counts as activity
*---------------------------------------------
Procedure WndProc(hWnd as Long, ;
  Msg as Long,wParam as Long,lParam as Long )
this.nLastActivity = seconds()
Return CallWindowProc(this.noldproc, ;
  hWnd,msg,wParam,lParam)

*---------------------------------------------
* Check last activity against time out
*---------------------------------------------
Procedure Timer

* developer feedback
goapp.debugox(goApp.lShowTimerDebug, ;
  'goTmr.timer() nLastActivity', ;
  this.nLastActivity, ;
  ': Current::', seconds(), ;
  ': .TimeOutInSeconds:', ;
  this.nTimeoutInSeconds, ':')
_screen.Caption ;
  = "Oh Hi! I started at " ;
  + transform(this.nLastActivity) ;
  + ". I'm going to quit at " ;
  + trans(this.nLastActivity ;
  + this.nTimeOutInSeconds) ;
  + " and it is currently " + trans(seconds())

* if current time is bigger than last
* activity + timeoutspan, we've exceeded
* the timeout, so offer to quit
if this.nLastActivity ;
  + This.nTimeOutInSeconds < seconds()
    This.eventTimeout()
endIf
endproc

*---------------------------------------------
* Override this event or bind to it
* to respond to user inactivity. You can
* change the nTimeOutInMinutes to offer
* multiple stages of timeouts.
*---------------------------------------------
procedure eventTimeout
if messagebox( ;
if messagebox( ;
    "Inactivity Timer timeout! Quit?",4)=6
    goApp.quit()
else
    messagebox("Not quitting!")
endif

enddefine
```

Sample code
The sample code included with this article has a lot of bells of whistles that you can play with, but to run it the first time, just load all of the programs into a single folder and run the main 'it.prg' program.

Not passing a parameter will cause Christof's Inactivity Timer to execute; passing the parm of 'IDLE', like so

```
do it with 'IDLE'
```

with cause Stefan's Idle Timer to execute instead.

Form level timeouts

You may want to close a form independently of closing the application. If this is the case, a form level time out is mechanism what you want.

A simple form level time out mechanism
A simple form level timeout mechanism has four pieces.

First, the application object has a property that holds the form level timeout value.

Next, the form has properties that store the 'last time' a KeyPress or MouseMove event was detected. They are initialized in the form's Init().

Third, code in the form's KeyPress and MouseMove events updates those properties with the new value each time either of those events fire.

Finally, a timer control on the form has code in timer event that adds the 'last time' to the timer time span. This total is compared to the current time. If the current time is bigger, that means the timeout has expired, and the form is closed.

Let's look at the actual code.

The main program, it.prg, initializes the time span for the form timeout.

```
giSecondsForFormTimeout = 10
```

The form's Init() method initializes the properties used to store the last time the keypress and mousemove events fire as well as change the caption to display a countdown of the timer.

```
Init()
this.caption ;
  = "Oh Hi! I'm going to disappear in " ;
  + transform(goApp.iSecondsForFormTimeout) ;
  + " seconds."
thisform.tmr.Interval = 1000
thisform.tKeypress = datetime()
thisform.tMousemove = datetime()
```

The form's KeyPress() and MouseMove() events update the form's 'last time' properties when executed.

```
KeyPress()
thisform.tKeypress = datetime()
```

and

```
MouseMove()
thisform.tMousemove = datetime()
```

Finally, the timer control's Timer() method does all the heavy lifting. The timer Interval property controls how often the timer will check to see if the timeout span has been exceeded. The timeout span is added to the 'last time' and the total is compared to the current time. If the current time is greater than the total, the span as been exceeded. If the span has been exceeded, the method that handled the closing of the form is called. Additionally, the form's countdown in the caption is updated.

```
Timer.timer()
local ltLast, ltCurrent

ltLast = max(thisform.tMousemove, ;
  thisform.tKeypress)
ltCurrent = datetime()

thisform.Caption ;
  = "Oh Hi! I'm going to disappear in " ;
  + transform(goApp.iSecondsForFormTimeout ;
  - (ltCurrent - ltLast)) ;
  + " seconds."

if tlLast + goApp.iSecondsForFormTimeout ;
  < ltCurrent
  thisform.Release()
endif
return
```

Timer resolution

While this example uses datetime() to track the last time activity was detected as well as identify the current time. The datetime() function only has a resolution to a second; if you need better, use seconds().

Problem

This mechanism isn't foolproof – the form level's KeyPress and MouseMove events do not fire when movement occurs within a control or when the mouse is positioned over a control. Try this:

In the sample app, run the Timer Form - Simple form from the File menu. The Done command button has focus. Tab to the textbox. Once the textbox has focus, the countdown timer resets. Spend a few seconds typing and deleting in the textbox, and you'll see the countdown timer won't reset.

Second, move the mouse over the listbox. As the mouse is moved, the countdown timer resets. Now, as the timer counts down, move the mouse only within the listbox. You'll see that the countdown timer won't reset.

This may or may not be a big deal to you. If it is, what are the options?

The easy way out is to add code to each control's base class, such that the control's KeyPress and MouseMove events update the form's tKeypress and tMousemove properties.

Retrofitting an application can be more complicated, because an instance of a control may already contain code in the KeyPress or MouseMove event, and thus the parent class's method code won't get executed. You'd have to manually include a dodefault() call in each instance that contains code in one of those events.

A more complicated form level time out mechanism

If the previous retrofitting methodology doesn't cut it for you, you can use BindEvent to watch for all activity, and if none is detected within the timeout period, call the ActiveForm's mechanism to close it, replacing the goApp.quit() statement, like so:

```
Procedure eventTimeout
if messagebox("Form timeout! Quit?",4) =6
  _screen.ActiveForm.release()
else
  messagebox("Not quitting!")
endif
```

Other issues to consider

Demonstrating the handling of open edits in a form while trying to close it is beyond the scope of this article because of the wide variety of potential edit mechanisms. Believe it or not, earlier this year, I saw a VFP 9 application that still used the 1.x/2.x SCATTER and GATHER from memvars technique.

Timeouts are not trapped during certain circumstances, including file-system operations like COPY FILE and long-running SQL queries. Other developers have posited that timeout mechanisms aren't reliable when an ActiveX control has focus (I generally don't use them in my applications, so I have no direct experience there.)

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com