December, 2003

Taking Advantage of Idle Cycles

Make Your Application Work When the User Isn't

by Tamar E. Granor, Technical Editor

A couple of years ago at a conference, a fellow asked me if there was a way to have a VFP application do some background processing when the user wasn't doing anything. He had in mind something like a screen saver, except that it would do useful work. I thought for a moment and suggested he might be able to do something with a timer, but didn't really have an answer.

I found the problem interesting and, on the plane home, started playing with the idea. But it quickly got complicated and when I got back to the office, I set it aside.

A couple of weeks ago, the idea surfaced again and I realized that VFP 8's event binding capabilities make the task much simpler. I sketched out the idea and in less than a day, had the whole thing working. The result is a structure that lets you take advantage of the time when your application is running but the user isn't working with it.

The Idea

The goal is to watch for inactivity and when the user pauses long enough, start some processing. When the user starts working again, the background processing needs to stop pretty quickly. The next time the user pauses, processing should pick up where it left off.

The requirement to be able to interrupt processing and return to it means this approach isn't appropriate for long, monolithic tasks, but is better suited to tasks that works on a record at a time in a loop. For example, you might use it to consolidate data and move it to a data warehouse.

The Strategy

Two cooperating classes make this work. The first is based on a timer—it watches for inactivity. The second is based on the session class (in order to have a private data session) and does the actual background processing.

The key to the whole thing is the ability to bind events. The KeyPress and MouseMove events for every control and form are bound to the timer's Reset. That prevents the timer from firing until the user pauses long enough. "Long enough," of course, is measured by how long it takes for the timer's Interval to elapse, which you can set appropriately for the application.

The Timer class

The timer subclass, tmrUseIdleCycles, has four custom properties, three of which connect it to the processing class:

- cProcess – Name of the processing class.
- cProcessLib – Name of the PRG file containing the processing class. Since it's based on a session, it can't be subclassed visually.
- oProcess – Object reference to the processing class, populated programmatically when the class is instantiated.
- lNowProcessing – Indicates whether background processing is currently occurring.

The main action occurs in the timer's Reset and Timer events. When the Timer event fires, we need to start processing. Because this might be the first time background processing was triggered, the first step is to make sure the processing object exists. If the processing object exists and if it has a StartProcessing method, we set the lNowProcessing flag to .T. and call the object's StartProcessing method. Here's the code for Timer:

```
This.Enabled = .F.

IF ISNULL(This.oProcess)
   IF NOT EMPTY(This.cProcess) AND ;
      NOT EMPTY(This.cProcesslib)
      This.oProcess = NEWOBJECT(ALLTRIM(This.cProcess), ;
                          ALLTRIM(This.cProcesslib))
   ENDIF
ENDIF


IF NOT ISNULL(This.oProcess) AND ;
   PEMSTATUS(This.oProcess,"StartProcessing",5)
   This.lNowProcessing = .T.
   This.oProcess.StartProcessing()
ELSE
   This.Enabled = .T.
ENDIF
```

The Reset method fires when an event bound to it (KeyPress or MouseMove) fires, indicating user action. If we're not processing, all that happens is that the timer starts counting again from 0. If we're currently processing, though, we need to stop:

```
* Need to receive parameters from bound methods
LPARAMETERS p1, p2, p3, p4

* Stop processing, if currently processing.
IF This.lNowProcessing
   This.lNowProcessing = .F.
   IF NOT ISNULL(This.oProcess) AND ;
      PEMSTATUS(This.oProcess, "StopProcessing",5)
      This.oProcess.StopProcessing()
   ENDIF
ENDIF

* Elapsed time has been reset to 0
* Restart the timer
This.Enabled = .T.
```

Note the LPARAMETERS line. Bound events pass their parameters to delegate methods (the methods they're bound to). So, we need to have enough parameters to handle either of the events we've registered, KeyPress or MouseMove.

The only other code needed for the timer is one line in Destroy to release the processing object:

```
This.oProcess = .NULL.
```

The Processing class

The processing class, sesUseIdleCycles, is based on the Session class, which maintains its own private data session. This is important because the processing class is likely to work with tables and you don't want to have to worry about resetting the environment each time you stop processing.

There are four custom properties, three of which are for tracking the progress of the background process:

- lSetupDone – indicates whether initial setup of the processing task has been completed.
- lProcessingDone – indicates whether all the actual processing has been completed.
- lCleanedUp – indicates whether clean-up from the processing task has been done.

- lStop – indicates whether processing needs to stop right now.

The class has five custom methods, two of which are exposed to the outside world. StartProcessing is called by the timer class' Timer event when the user is idle and processing can continue. StopProcessing is called by the timer class' Reset method when the user performs an action.

StopProcessing is simple. It sets the lStop flag to indicate that processing should pause:

```
PROCEDURE StopProcessing
* User acted again. Pause processing.

This.lStop = .T.

RETURN
ENDPROC
```

StartProcessing is the heart of the processing code. It checks how far we've gotten and continues processing until the lStop flag is set. StartProcessing calls three protected methods to perform the actual processing task: Setup, ProcessRecord, and CleanUp. It assumes that Setup opens necessary tables and leaves the current work area set to the table that drives the processing task.

```
PROCEDURE StartProcessing
* Timer fired. Begin processing

IF This.lProcessingDone
   RETURN
ENDIF

This.lStop = .F.
IF NOT This.lStop AND NOT This.lSetupDone
   * It should be enough to check This.lSetupDone, but
   * check This.lStop just in case something else
   * created a wait state and the user acted.
   This.Setup()
   This.lSetupDone = .T.
ENDIF

SCAN REST WHILE NOT This.lStop AND NOT This.lProcessingDone
   This.ProcessRecord()
   DOEVENTS
ENDSCAN

IF EOF()
   This.lProcessingDone = .T.
ENDIF
```

```
IF This.lProcessingDone AND NOT This.lStop
   This.CleanUp()
   This.lCleanedUp = .T.
ENDIF

RETURN
ENDPROC
```

One key line of code is DOEVENTS. Given the nature of background processing, the ProcessRecord method shouldn't contain any code that waits for events to fire (like READ EVENTS or WAIT). DOEVENTS tells VFP to stop for a moment and check whether any events have fired and, if so, process them. That allows processing of a user action, such as a key press or mouse movement; otherwise, the tight SCAN loop would run until it reaches the end of the table. Because the KeyPress and MouseMove events are bound to the timer's Reset event, when either is processed, the StopProcessing method gets called, which sets the lStop flag. As soon as the lStop flag is set, the SCAN loop reaches the end of this pass and processing stops for now.

From that explanation, it's clear that the user should never have to wait longer for control to return than once through the SCAN loop. Thus, it's important that one execution of ProcessRecord not be too long. If it takes 30 seconds to process a single record, the user might have to wait 30 seconds for background processing to stop when he wants to start working again.

The three methods called by StartProcessing are abstract (or "template") methods to be coded in subclasses:

- Setup – opens necessary tables and does any other pre-processing.
- ProcessRecord – processes one record from the active work area.
- CleanUp – closes tables and does any other post-processing.

Binding the events

It turns out that the trickiest part of the whole thing is binding the various form and control events to the timer. At first glance, it might seem that you could handle it in the timer's Init method with code like this:

```
* Bind actions of all open forms.

LOCAL oForm
```

```
FOR EACH oForm IN _Screen.Forms
   This.BindActions(oForm)
ENDFOR

This.Enabled = .T.
RETURN
```

The custom BindActions method, in turn, calls a recursive method that drills down through the form and binds each control:

```
* Bind keystrokes and mouse movements to the Reset method
* so the timer doesn't fire until the specified idle time
* has passed. Binding also allows processing to stop when
* the user resumes activity.

LPARAMETERS oForm
   * oForm = form to bind. If omitted, bind on containing form

LOCAL oWhatToBind

IF PCOUNT()=1 AND VARTYPE(oForm) = "O"
   oWhatToBind = oForm
ELSE
   oWhatToBind = ThisForm
ENDIF

This.BindOne( oWhatToBind )

RETURN
```

BindOne binds a single control and then drills down into that control's contained controls:

```
* Bind events of one control
LPARAMETERS oControl

IF PEMSTATUS(oControl, "KeyPress", 5)
   BINDEVENT(oControl,"KeyPress",This,"Reset")
ENDIF

IF PEMSTATUS(oControl, "MouseMove", 5)
   BINDEVENT(oControl, "MouseMove", This, "Reset")
ENDIF

IF PEMSTATUS(oControl, "Objects", 5)
   FOR EACH oControl IN oControl.Objects
      This.BindOne(oControl)
   ENDFOR
ENDIF

RETURN
```

This works just fine if all the relevant forms are open when you instantiate the timer. However, in an application setting, it's more likely that you set up the timer as part of the application's start-up activities. Then, each time a form opens, you need to bind its user-action events.

One possibility is to build the binding into the base form class. In the Init method, look for the appropriate object (a subclass of tmrUseIdleCycles) and call BindActions, passing the form itself as a parameter. While this would work, it means that you can't just plug the timer in and go. My preference is to make the whole thing self-contained. To do so, I created another timer subclass whose sole purpose is to detect unbound forms and bind their user-action events.

The new subclass, tmrBindForms, has only one custom property, oContainer, which is an object reference to the other timer, the subclass of tmrUseIdleCycles that created it. It has code in two methods. Init sets the oContainer property based on a parameter you pass when creating the tmrBindForms object:

```
LPARAMETERS oContainer

This.oContainer = oContainer
```

As is typical for timers, the main action occurs in the Timer event, where it calls the CheckBindings method of the containing timer:

```
* Turn timer off so it doesn't fire recursively
This.Enabled = .F.
IF NOT ISNULL(This.oContainer)
   This.oContainer.CheckBindings()
ENDIF
* Turn timer back on
This.Enabled = .T.
```

To accommodate the new timer, tmrUseIdleCycles needs a few more properties and one custom method. The additional properties are:

- lCheckBindings – determines whether this timer should instantiate a tmrBindForms timer to check for new forms periodically.
- nCheckBindingInterval – determines how often the tmrBindForms timer fires, that is, how frequently we'll check for unbound forms.
- oBindingTimer – object reference to the tmrBindForms timer.

As the code above indicates, the new method, CheckBindings, is called when the tmrBindForms timer fires. It loops through all open forms and checks whether the events for that form are already bound. If not, it calls BindActions to bind them. It also binds the KeyPress and MouseMove events of the main VFP window, so that user action outside forms prevents or stops background processing, as well.

```
* Loop through running forms to make sure all controls are bound
* to this timer's events.

LOCAL aBound[1], oForm, nBound, lFound

nBound = AEVENTS(aBound, This)

* Check forms
FOR EACH oForm IN _SCREEN.Forms
   * Check whether we have it already
   lFound = .F.
   FOR nItem = 1 TO nBound
      IF aBound[nItem, 2] = oForm
         lFound = .T.
         EXIT
      ENDIF
   ENDFOR

   IF NOT lFound
      This.BindActions(oForm)
   ENDIF
ENDFOR

* Check screen events
lFound = .F.
FOR nItem = 1 TO nBound
   IF aBound[nItem,2] = _SCREEN
      lFound = .F.
      EXIT
   ENDIF
ENDFOR

IF NOT lFound
   This.BindActions(_SCREEN)
ENDIF

This.Enabled = .T.
```

CheckBindings uses the AEVENTS() function to get a list of events that are currently bound. Unfortunately, the only way to test whether a particular form's events are bound is to loop through the array, looking for that form as an event source. If you have performance problems with this technique, consider tracking bound forms manually using an array or collection.

The tmrUseIdleCycle class' Init method needs additional code to set up the tmrBindForms. Here's the new version of the Init method:

```
DODEFAULT()

* Add timer to check bindings
IF This.lCheckBindings
   This.oBindingTimer = ;
      NEWOBJECT("tmrBindForms","IdleCycles","",This)
   WITH This.oBindingTimer
      .Interval = This.nCheckBindingInterval
      .Enabled = .T.
   ENDWITH
ENDIF
```

Finally, while not strictly necessary, it's neater if tmrUseIdleCycles' Destroy method cleans up both the bindings and its object references:

```
UNBINDEVENTS(This)

This.oBindingTimer = .NULL.
This.oProcess = .NULL.
```

Putting it to work

Most of the work in setting up background processing comes in subclassing the sesUseIdleCycles class. You need to provide code for the template methods SetUp, ProcessRecord and CleanUp. To do so, you need to carefully consider the processing involved and organize it so that interruptions won't cause a problem.

To demonstrate, I created a subclass that creates a data warehouse based on order data from the TasTrade database. It creates a single record for each product showing the total number sold and the total income for that product. While this particular task can be done with a single query and, given the size of the data set, would run reasonably quickly, it's an example of the sort of processing you might do with this technique.

The Setup method opens the tables needed and creates a cursor to hold the results. (In a production application, presumably you'd use a table rather than a cursor.) The code ensures that the Order_Line_Items table is selected upon return, so that it becomes the driving table for record processing.

```
PROCEDURE Setup

OPEN DATABASE _SAMPLES + "TasTrade\Data\TasTrade"
USE Order_Line_Items ORDER Product_ID IN 0
```

```
* Setup up cursor to hold results
CREATE CURSOR ProductWareHouse (Product_ID C(6), ;
                                TotalCount N(12), ;
                                TotalSales Y)

SELECT Order_Line_items

RETURN
ENDPROC
```

The ProcessRecord method actually processes more than one record on each pass. It collects data for a single product and creates a record in the warehouse for that product. Note the careful handling of the record pointer, with the use of SCAN REST WHILE to process only the relevant records and SKIP –1 to ensure the correct record pointer position upon return. The output here (the "?" line) is to demonstrate that the process works; in production code, you'll want to avoid any UI code in your background processing.

```
PROCEDURE ProcessRecord
* Process one Product_ID

LOCAL cCurrID, nCount, nTotal
LOCAL nOldSelect

nOldSelect = SELECT()
SELECT Order_Line_Items

cCurrID = Product_ID

ACTIVATE SCREEN
? "Processing ", cCurrID


nCount = 0
nTotal = 0
SCAN REST WHILE Product_ID == cCurrID
   nCount = nCount + Quantity
   nTotal = nTotal + (Quantity * Unit_Price)
ENDSCAN

* Reset record pointer to the last
* record of the current processing group
SKIP -1

* Add warehouse record
INSERT INTO ProductWarehouse ;
   VALUES (m.cCurrID, m.nCount, m.nTotal)

SELECT (nOldSelect)

RETURN
```

The CleanUp method, not surprisingly, cleans up. As in ProcessRecord, the output here is for testing purposes, not something you'd do in production code:

```
PROCEDURE CleanUp

WAIT CLEAR
USE IN Order_Line_Items
SET DATABASE TO TASTRADE
CLOSE DATABASE
ACTIVATE SCREEN
? "Results are in Data Session:", DatasessionID



RETURN
ENDPROC
```

Once you have a subclass of sesUseIdleCycles, you need to hook it up to a timer. You can do that by instantiating tmrUseIdleCycles directly and setting the necessary properties, like this:

```
oIdleTimer = CREATEOBJECT("tmrUseIdleCycles")
WITH oIdleTimer
   .Interval = 30000  && 30 seconds
   .cProcess = "sesProductsToWareHouse"
   .cProcessLib = "BuildWareHouse.PRG"
ENDWITH
```

However, it's probably a better idea to create a subclass that has everything set correctly and instantiate that when you need it. The class tmrBuildWarehouse in BuildWarehouse.VCX on this month's PRD is set to trigger the sesProductsToWarehouse class when the user is idle for five seconds or more. Five seconds is likely to be too short an idle time in production, but it's handy for this example.

One reason for subclassing rather than instantiating is for more control over binding. Because the code to create the timer to check bindings is in the Init method, changing lCheckBindings or nCheckBindingInterval after instantiating the object doesn't actually change anything.

Once you've created the necessary subclasses, using them is easy. Simply instantiate your subclass of tmrIdleCycles.

```
oIdleTimer = CREATEOBJECT("tmrBuildWarehouse")
```

If lCheckBindings is .T., you don't need to do anything else; forms and their controls will be bound automatically.

In a production application, you'd probably use a property of the application object or a variable declared in the main program to hold a reference to the timer, so that you can create it when the application starts and destroy it when the application shuts down.

In some cases, you might put the timer on a form (say, when you're using a top-level form in place of an application object). In that case, you need to actively clean things up when you close that form (because methods of the form itself are bound to the timer). The tmrUseIdleCycles class has a method, Cleanup, that you can call (or bind to the form's Release event) to unbind events as well as set the object references within the timer to .NULL.

Taking it farther

The code in this article focuses on table processing in interactive applications. You can extend it in a couple of ways.

First, you might want to use this technique in a non-interactive application, such as a web server. In that case, you need to consider what events indicate that the application is working and bind those events to the timer's Reset method rather than KeyPress and MouseMove. Since most VFP events are really focused on user interaction, you might have to define your own events and use the RaiseEvent() function to fire them. Another alternative in a non-interactive situation is to bind properties that change frequently to the Reset method.

You might also want to use this technique in cases when the processing to be done isn't table-based. The trick, in that case, is to create and populate a cursor in SetUp to drive the process. Organize ProcessRecord to handle one record of that cursor on each path. For example, if you want to do something in each folder of a directory tree, you could set up a cursor with one folder per record. For a multi-step, non-repetitive process, you could store the name for each step in a record and then use a CASE statement in ProcessRecord to identify the current step.

Final thoughts

Be sure to make the background processing unobtrusive to your users. You may need to experiment with the Interval setting for both tmrUseIdleCycles and tmrBindForms. In fact, you may want to spend some time watching users work to get an idea of the appropriate

intervals. Alternatively, you might let the users determine the tmrUseIdleCycles' Interval.

Some tasks that might initially seem well-suited to this kind of processing may be a problem. For example, you might consider background processing to rebuild indexes or pack data. The problem, of course, is that you need exclusive access to do those things and you might not be able to get it. However, if you're willing to take the risk of not succeeding with some tables some of the time, you can write the ProcessRecord code to check whether exclusive access is available and simply move on if it's not.

Another consideration is that, in an interactive situation, the user might be idle in your application, but working in another application and performing background processing could interfere with the responsiveness of the other application. There's no native way to handle this situation. You can create (or find) an .FLL that can respond to Windows events, and extend your timer class so that it is disabled when your application loses focus, and enabled when the application regains focus.

You can also take an entirely different approach than the one in this article. Set up the background processing as a separate process (that is, run it independently from your application) and give it "idle priority." That sets it to run only when the computer is idle. The advantage of this approach is that it looks at the whole Windows environment. The disadvantage is that you have less control over how it starts and stops.

The first person I shared this code with immediately suggested uses for it that I hadn't considered. So, let your imagination loose as you think about what tasks you can let your applications do while your users are talking on the phone, getting a cup of coffee, or just thinking.