

Creating Outlook-Style Desktop Alerts with Visual FoxPro

Kevin Ragsdale

I love the Desktop Alerts in Outlook 2003. They provide useful information without interfering with my work. I can't stand to be working on one thing and suddenly have a MESSAGEBOX appear from an app informing me that it just did something wonderful on my behalf. The Desktop Alerts simply “appear” on my screen for a few seconds, letting me know I have new mail. They even provide some details (name of sender, subject). If I want to read the email immediately, I simply click on the Alert and Microsoft Outlook opens the message. Otherwise, I can ignore the alert and it will fade away.

I thought something like this would be a great enhancement for some of my apps, so I went about the task of finding out how to mimic Outlook's Desktop Alerts.

For the rest of the document, I'll refer to the **Desktop Alerts System** as **DAS**.

Note to reader: Ask two Visual FoxPro developers the best way to handle something, and you'll probably get five or six different answers. If you think there's a better way to do something in the DAS than what I have done, or if you find errors, omissions, etc., please do not hesitate to let me know. As you will quickly see while reading this white paper and reviewing the source code: this ain't no code clinic!

Requirements for the DAS

- The alert should fade-in, wait for user-interaction, and fade-out
- Allow the user to configure alerts
- The alert should be closable, movable, and *ignorable*
- The alert should return a value to the client, based on user-interaction
- It should be easy to create an alert on-the-fly, like a MESSAGEBOX
- The alert system will be COM (EXE) – based, so I can create it in VFP 9 and use it in earlier versions of VFP (or other languages). This also allows me to package standard (default) graphics, sound files, etc., directly into the executable.

My ultimate goal for the DAS is to be able to let my apps instantiate a “global” alert manager, and issue alerts whenever necessary. This will involve the development of an AlertManager class, which we will discuss later. Since there may be times when there are more than one alert visible on the screen, I also want each alert to be unique. So, we will also be creating an Alert class.

What we will end up with is a visual class library (DESKALERT.VCX), which will contain all of the elements of the DAS UI; a session-based class named AlertManager; a session-based class named Alert; an include file name VFPALEERT.H, and a variety of support files (graphics, sounds, etc).

I'll start off by creating a folder on my development drive named VFPALEERT. In this folder, I'll create a new project named VFPALEERT.

Next up, let's start building our visual class library (DESKALERT.VCX).

Building the DAS User-Interface

Since I leveraged the idea for the DAS from Microsoft Outlook, the UI for the DAS will be very similar to the Outlook Desktop Alerts. Let's take a look at a Desktop Alert from Microsoft Outlook (Figure 1), and pick apart the pieces we'll need for the DAS (Table 1):

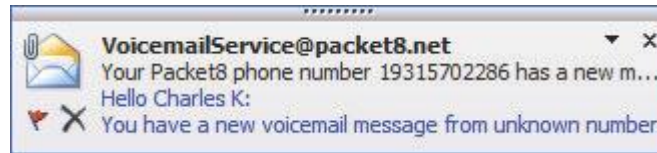


Figure 1: Microsoft Outlook Desktop Alert

Item	Picture	Description
Title Bar		This will replace the Alert form's TitleBar
Icon		Each alert will have an icon. If we do not specify an icon, a default icon will be used
Title	VoicemailService@packet8.net	Each alert will have a title. If not specified, a default Title will be used
Subject		Subject is optional
Details	Hello Charles K: You have a new voicemail message from unknown number	Each alert will have details. They can be plain (text-only), or appear as a hyperlink with an action
Settings Button		The Settings button can be hidden for each alert. Default is visible
Close Button		The Close button will appear on all alerts
Task 1 Icon		Tasks are optional. If we include a task, we can optionally provide an icon for the task, or use a default task icon
Task 2 Icon		

Table 1: Pieces of a Desktop Alert

Let's build the Alert form!

We'll start by creating a Form class (frmAlert) with the following "standard" properties (Table 2):

Property	Value	Description
Height	69	Just because I'm trying to mimic the Outlook alerts.
Width	321	
ShowWindow	2 (Top-Level Form)	Transparency requires the form to be a Top-Level form.
ShowInTaskBar	.F.	No need for the Alert to appear on the TaskBar.
ShowTips	.T.	We'll use ToolTips for the buttons and tasks.

BorderStyle	3 (Sizable)	We'll change this in the form's Init().
Caption	"DESKALERT"	We'll use this for API functions that look for other alerts.
ControlBox	.F.	No need for these, because we're not using a TitleBar. We'll turn it off by setting the TitleBar to 0.
MaxButton	.F.	
MinButton	.F.	
TitleBar	0 (Off)	See? No TitleBar.
AlwaysOnTop	.T.	Make the form appear above everything else.

Table 2: frmAlert Standard Properties

There are a few custom properties we need to add to our form (Table 3), since we are going to have it fade in and fade out. Here are a couple of rules regarding "fading forms":

1. Must be a Top-Level Form
2. Only works on Windows 2000 and higher

Believe it or not, #2 is still an issue for many of my customers.

We're going to use a combination of Windows API functions and a VFP Timer control to make the form fade-in and fade-out. I'll talk about these in a little while.

Special thanks go out to Mike Lewis, who published an article in FoxPro Advisor magazine (June 2006) with some better ways of "fading" forms than the methods I had originally developed.

Property	Value	Description
ICanFade	.F.	If we are running on Windows 98, we can't fade in and out, so we'll set it to .F. Otherwise, it will be set to .T. We'll add an Assign method to fire the initial API functions (if .T.)
nTransparency	0	Current level of transparency for the form. Ranges from 0 (completely transparent) to 255 (completely opaque). Since we're "fading-in", we'll start out at 0.
nScreenHeight	0	The height of the physical screen, taking into account the TaskBar.
nScreenWidth	0	The width of the physical screen, taking into account the TaskBar.
nStatus	-1	At any given time, the form will be either 0 (Disappearing) or 1 (Appearing).
nFadeFactor	0	Amount the transparency that will increase or decrease each time the fading timer fires.

Table 3: frmAlert Custom Properties (related to transparency)

We'll add a few more custom properties, unrelated to the fading in and out part (Table 4):

Property	Value	Description
oParent	NULL	Alert object which "creates" the Alert form
oParams	NULL	A parameters object which is passed in by oParent
nResult	0	<p>The return value from the form. Return values can be one of the following:</p> <ul style="list-style-type: none"> -1: Alert timeout 1: User clicked the Close button 2: User clicked the details link 3: User selected Task 1 4: User selected Task 2 <p>We're going to add an Assign method for this property.</p>

Table 4: frmAlert Custom Properties (Other)

We have our form class with its properties set. Let's look at the aforementioned Assign() methods that we need for a couple of the form properties (Table 5):

Property Assign() Method	Description
ICanFade_Assign()	If we are running Windows 2000 or higher, ICanFade is set to .T., and this assign method will make the initial API function calls.
nResult_Assign()	Returns the form's nResult property back to oParent (the Alert object which created this instance of frmAlert). oParent handles the Callback to the client.

Table 5: frmAlert Property Assign Methods

Now let's take a look at some custom methods (Table 6) we'll need:

Method	Description
SetupForm()	Called from the form's Init(). Calls methods to calculate nScreenHeight and nScreenWidth, and calculates the position of the form.
RenderForm()	"Draws" the form based on the parameters passed in via the oParams object.
HideForm()	Starts the fading timer, to make the form fade-out.
GetScreenHeight()	Sets the nScreenHeight property
GetScreenWidth()	Sets the nScreenWidth property

Table 6: frmAlert Custom Methods

Now that we have our form set up with our properties and methods, we need to add some controls to the form. Since I'm already working with a visual class library for the form (DESKALERT.VCX), let's create some more classes for the form objects.

Based on the items in Table 1 (when we separated the pieces of an alert form), I'm going to create a class for the buttons on the form. I'm not going to use a CommandButton, though. I'm going to use a Container Object, and add an Image to it (pictured below in the Visual FoxPro IDE).

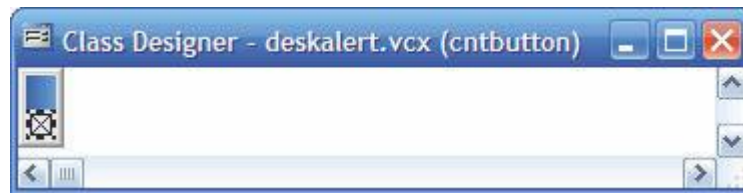


Figure 2: cntButton

Here are the properties for the Container, which is stored in DESKALERT.VCX as cntButton:

```
Width = 15
Height = 15
BackStyle = 0
BorderWidth = 0
MousePointer = 15
BackColor = RGB(192,192,192)
BorderColor=RGB(128,128,128)
Name = cntButton
```

And now the properties for the image we add to the container:

```
BackStyle = 0
Height = 11
Left = 2
MousePointer = 15
Top = 2
Width = 11
Name = "imgImage"
```

The effect I'm looking for is to have a button appear as a simple graphic (like the Close button and Settings button in Figure 1). When the mouse moves over it, the BackColor will change and a border will appear. To do this, we'll add code in the Container's MouseEnter() and MouseLeave() methods.

In the MouseEnter(), we'll change the BackColor to light gray, put a dark gray single border around the container, and set the BackStyle to opaque:

```
PROCEDURE MouseEnter
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    WITH THIS
        .BackColor = RGB(192,192,192)
        .BorderWidth = 1
        .BackStyle = 1
    ENDWITH
ENDPROC
```

When the MouseLeave() event occurs, we'll change the BackColor back to the same as the form, remove the border, and set the BackStyle to transparent:

```
PROCEDURE MouseLeave
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    WITH THIS
        .BackColor = ThisForm.BackColor
        .BorderWidth = 0
        .BackStyle = 0
    ENDWITH
ENDPROC
```

The Image will also have code in its MouseEnter() and MouseLeave(), which will basically “bubble up” to the Container’s MouseEnter() and MouseLeave():

```
PROCEDURE imgImage.MouseEnter
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    This.Parent.MouseEnter()
ENDPROC

PROCEDURE imgImage.MouseLeave
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    This.Parent.MouseLeave()
ENDPROC
```

I try not to put too much code in “events”, so I’m going to add an abstract custom method to the Container and name it OnClick(). Then I’ll add the following code to the Container’s Click() event:

```
PROCEDURE Click
    This.OnClick()
ENDPROC
```

And supplement that with the following code in the Image’s Click() event to “bubble up” to the Container:

```
PROCEDURE imgimage.Click
    This.Parent.OnClick()
ENDPROC
```

I’m going to leave the OnClick() method empty in this class, since it is an abstract method. We’ll fill-in this method on the form after we’ve added the sub-classed objects to it.

Since I’m doing the whole “change the BackColor and add a border” routine, I’ll add the following code to the MouseDown() and MouseUp() events for the Container.

In the MouseDown(), we’ll change the BackColor to a ‘darker’ gray, change the BorderColor to black, and make the BackStyle opaque:

```
PROCEDURE MouseDown
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    WITH THIS
        .BackColor = RGB(177,177,177)
        .BorderColor = RGB(0,0,0)
        .BackStyle = 1
    ENDWITH
ENDPROC
```

```

    ENDWITH
ENDPROC

```

Then in the MouseUp(), we'll revert to the light gray BackColor, the dark gray BorderColor, and the transparent BackStyle:

```

PROCEDURE MouseUp
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    WITH THIS
        .BackColor = RGB(192,192,192)
        .BorderColor = RGB(128,128,128)
        .BackStyle = 0
    ENDWITH
ENDPROC

```

And, of course, we'll want to add code to the Image's MouseDown() and MouseUp() to "bubble up" to the Container:

```

PROCEDURE imgimage.MouseDown
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    This.Parent.MouseDown()
ENDPROC

```

```

PROCEDURE imgimage.MouseUp
LPARAMETERS nButton, nShift, nXCoord, nYCoord
This.Parent.MouseUp()
ENDPROC

```

I'll also add the following code to the Container's Init() event:

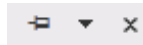
```

PROCEDURE Init
WITH THIS
    .BackColor = ThisForm.BackColor
    .SetAll("ToolTipText",.ToolTipText)
ENDWITH
ENDPROC

```

We now have a base class for our form's buttons, so let's put it to work. What we are going to do is create a Settings button and a Close button, sub-classed from the cntButton we just created. We'll also create a Push-Pin button, which will allow the user to "pin" the Alert to the desktop.

To create the Settings, Close, and Push-Pin buttons, all we need to do is change the Picture property of the Image object on the Container for each of the sub-classed objects, and add a ToolTipText (i.e. for the Close button, "Click here to close this alert"). Here's what they look like on an alert:



Tasks are also represented on the Alert form as buttons, sub-classed from the cntButton. The only real difference is a larger height and width for the container (18x18), and for the image (16x16).

As in the case of the other "buttons", we will drop Task Buttons on the Alert form at design time, and place code in the OnClick() method of each.

Now we'll create classes for the Title, Subject, and Details labels which will appear on the Alert form.

The Title is simply a label control with the following properties:

```
AutoSize = .F.  
FontBold = .T.  
FontName = "Tahoma"  
FontSize = 8  
BackStyle = 0  
Caption = "lblTitle"  
Height = 15  
Visible = .F.  
Width = 250  
Name = "lblTitle"
```

The Subject label has similar properties:

```
FontName = "Tahoma"  
FontSize = 8  
BackStyle = 0  
Caption = "lblSubject"  
Height = 15  
Visible = .F.  
Width = 250  
Name = "lblSubject"
```

And the Details label also has similar properties, though we add a custom property named `lIsALink` and an `Assign()` method for the `lIsALink` property:

```
FontName = "Tahoma"  
FontSize = 8  
WordWrap = .T.  
BackStyle = 0  
Caption = "lblDetails"  
Height = 30  
Visible = .F.  
Width = 250  
lIsALink = .F.  
Name = "lbldetails"  
  
PROCEDURE lIsALink_Assign  
LPARAMETERS vNewVal  
THIS.lIsALink = m.vNewVal  
WITH THIS  
IF .lIsALink  
.ForeColor = RGB(0,0,255)  
.MousePointer = 15  
ELSE  
.ForeColor = RGB(0,0,0)  
.MousePointer = 0  
ENDIF  
ENDWITH  
ENDPROC
```


If the Details label should have an action (result value) assigned to it, then it will be considered to be a 'link' – which will allow it to look as though it is a hyperlink. Otherwise, it will be simple text.

We only have two UI pieces left for the Alert form: the TitleBar image and the Icon. Both images are just Image controls, with no real special features.

Now that the UI pieces are in place, we will add a couple of Timer controls to the form: tmrFader and tmrWait. tmrFader handles the "fade-in" and "fade-out" features, while tmrWait causes the Alert form to stay on screen for the configured amount of time. We'll look at both timers in more detail in the next section.

We now have the UI classes built and ready. Here's how everything looks in the class browser:

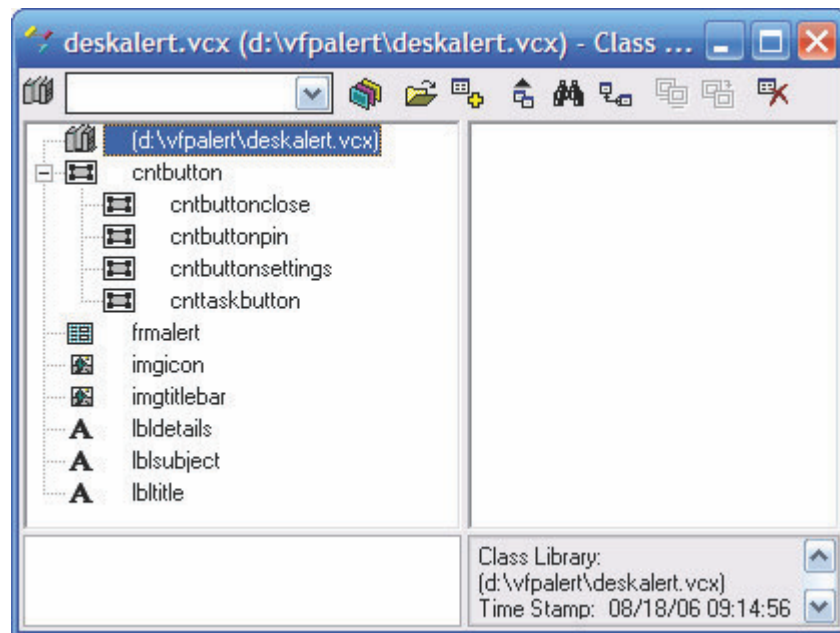


Figure 3: Class Browser View of DESKALERT.VCX

And our Alert form in the Visual FoxPro IDE:

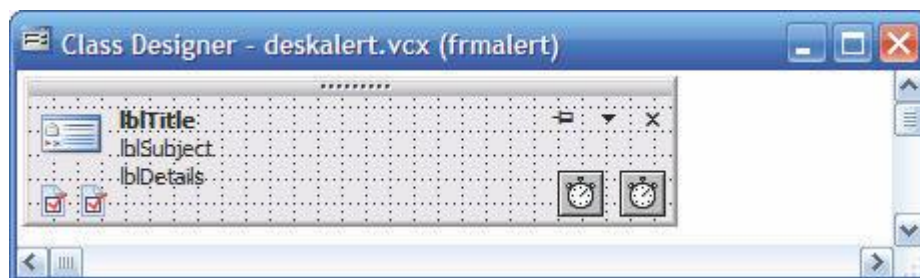


Figure 4: frmAlert in VFP IDE

Now let's take a closer look at making the form fade-in and fade-out.

During the Alert form's Setup(), we check to see if we are running Windows 2000 or higher:

```
ThisForm.lCanFade = (VAL(OS(3))>=5)
```

The `lCanFade_Assign()` method does the following:

```
LPARAMETERS vNewVal
This.lCanFade = m.vNewVal

IF This.lCanFade
    ** The SetWindowLong tells the system
    ** to make this form a "layered window".
    ** See the "Transparent Forms"
    ** solution sample in VFP for the 'code'.
    _SetWindowLong(This.hWnd, -20, 0x00080000)

    ** SetLayeredWindowAttributes
    _SetLayeredWindowAttributes(This.hWnd, 0, 0, 2)
ENDIF
```

Of course, this code assumes that the API declarations have already been done. We'll cover that later. The assign method tells the system that the Alert form will be a "layered" window, and sets the attributes to make the form completely transparent.

Still in the `SetupForm()` method, we calculate the form's 'fade factor', set the status of the form to 'appearing', and start the `tmrFader` timer (you'll notice the code below is wrapped by a `WITH/ENDWITH`):

```
** Start the tmrFader, which causes the
** form to appear gradually with API calls.
.nFadeFactor = 255/(1000/lnInterval)

IF .lCanFade
    .nStatus = 1
    .tmrFader.Enabled = .T.
ENDIF
```

The `Timer()` event for `tmrFader` fires every 20 milliseconds (or 50 times in one second). Each time it fires, it decreases the level of transparency for the form, which provides the "fade-in" effect. When the form is "fading-out", the `tmrFader` increases the level of transparency:

```
DO CASE
    CASE .nStatus = 0 && Disappearing
        .nTransparency = .nTransparency - .nFadeFactor
        _SetLayeredWindowAttributes(.hWnd, 0, .nTransparency, 2)

        IF .nTransparency <= 1
            This.Enabled = .F.
            .Release()
        ENDIF
    CASE .nStatus = 1 && Appearing
        .nTransparency = .nTransparency + .nFadeFactor
        _SetLayeredWindowAttributes(.hWnd, 0, .nTransparency, 2)

        IF .nTransparency >= 255 * (.oParent.nPercent/100)
            This.Enabled = .F.
            .tmrWait.Enabled = .T.
        ENDIF
ENDCASE
```

When the form is “fading-out”, the timer will disable itself when it reaches a transparency level less than or equal to one, then it calls the form’s Release().

When “fading-in”, the timer will disable itself once it reaches the configured level of transparency. It will then enable the tmrWait timer, which causes the Alert form to “sit” on the screen for the configured number of seconds (waiting for user interaction).

In essence, all of the fade-in and fade-out activity is driven by tmrFader.

The OnClick() Methods

We created OnClick() methods for most of the UI classes we’ve dropped on the Alert form. Let’s take a look at the code in each one:

The Close Button, TaskOne Button, TaskTwo Button, and Details label all have code to set the Alert form’s nResult property, then call the form’s HideForm() method.

If you remember, the Alert form will return one of five possible values for the nResult:

- 1: Alert timeout (user did not interact with the form)
- 1: User clicked the Close button
- 2: User clicked the Details link
- 3: User selected Task 1
- 4: User selected Task 2

So, in the Close button’s OnClick() method, we simply set ThisForm.nResult = 1, then call ThisForm.HideForm().

And the HideForm() method? It sets the ‘status’ of the form to 0 (disappearing) and enables the tmrFader timer, which increases the level of transparency until the Alert form has vanished.

Now we’re left with three more custom methods for our Alert form: GetScreenHeight, GetScreenWidth, and RenderForm.

GetScreenHeight() and GetScreenWidth() each use the SystemParametersInfo function in the Windows API to set the height and width of the physical screen, taking into account the TaskBar.

RenderForm() is called from the SetupForm() method after the screen height and width values have been captured, but before the ICanFade property is assigned a value. RenderForm() takes the parameters passed into the form and sets various properties (like label captions for the Title, Subject, and Details). It also determines whether to show or hide the Task buttons, the Settings button, whether to make the Details label appear to be a “link”, and which icon to use for the form.

One of the requirements for the DAS was to allow the Alert to be movable. BUT, we have no title bar. How can we move the form without a title bar? Windows API functions! Thanks to the great samples at the Win32 Functions in VFP web site at <http://www.news2news.com/vfp>, we can easily move the form by placing the following code in the form’s MouseDown() event:

```

PROCEDURE MouseDown
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    IF nButton = 1
        LOCAL hWindow
        hWindow = GetFocus()

        = ReleaseCapture()
        = SendMessage(hWindow, WM_SYSCOMMAND, MOUSE_MOVE, WM_NULL)
        = SendMessage(hWindow, WM_LBUTTONDOWN, 0, 0)
    ENDIF
ENDPROC

```

As before, this code assumes the API functions have already been DECLARED, and as you will see later, they have been.

What if the user clicks on one of the controls on the form, but wants to move the form? Simple. Put a call to ThisForm.MouseDown() in the MouseDown() event for each object on the form that *does not* have an OnClick() method defined. These include the imgTitleBar, imgIcon, lblTitle, lblSubject and lblDetails (if lblDetails is not a ‘link’). Any objects which have an OnClick() should not “bubble up” their MouseDown() to the form’s MouseDown(), for obvious reasons.

A couple of things we have not yet covered are the PushPin button and the Settings button OnClick() methods. The PushPin button changes its Picture property to make the pin appear “pushed”, and it “freezes” the tmrWait timer. This allows the user to keep the alert on-screen.

The settings button will create a menu which will include a menu bar to configure the DAS settings, and will also include bars for Task 1 and Task 2 (if applicable).

Besides a couple of other “setup” functions to place the Alert form on the screen at a specific position, we are done at this point with the Alert form!

The AlertManager Class

Earlier, I stated that my ultimate goal for the DAS is to be able to let my apps instantiate a “global” alert manager, and issue alerts whenever necessary. In order to accomplish the goals, we’ll need to create an AlertManager class.

After creating the AlertManager class, I can add a new “alert manager” property to my application object (alternatively, you could create a global variable):

```
oAlertMgr = .NULL.
```

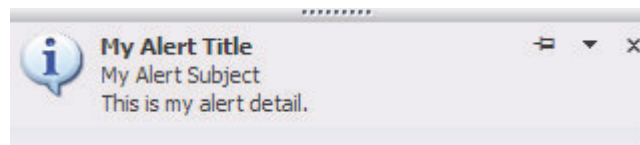
During the app startup, I’ll instantiate it:

```
MyApp.oAlertMgr = CREATEOBJECT("vfpalert.AlertManager")
```

Whenever I need to popup an alert, I’ll do something like this:

```
loAlert = MyApp.oAlertMgr.NewAlert()  
loAlert.Alert("This is my alert detail.",64,"My Alert Title", ;  
             "My Alert Subject")
```

And voila! The following desktop alert fades in on the screen:



The AlertManager session class will be OLEPUBLIC. This will be the primary programmatic interface to the DAS.

The AlertManager class contains the following custom properties:

Property	Description
Alerts	A collection object which references each alert currently in the system
nWait	Length of time alerts should appear on screen before fading out (default is 10)
nPercent	Percentage of “opaqueness” for the alerts (default is 90)
lSound	If we want to play a sound when alerts appear, .T. (default); otherwise, .F.
cSound	File name of the sound we want to play (default is ALERT.WAV)
cSettingsPath	Full path to a “writeable” folder where the settings should be stored
lCleanMemory	.T. to reduce memory consumption (default is .F.)

The AlertManager class has the following custom methods:

Method	Description
Setup()	Called from the Init(). Makes calls to the next two methods, and creates the Alerts collection
APIDeclare()	Since we need a variety of API declarations, let’s do them once from the AlertManager
ReadSettings()	Read settings from a DACONFIG.XML configuration file (if available);

	otherwise, loads the default settings
NewAlert()	Creates a new Alert, adds it to the Alerts collection, and returns a reference to the Alert to the calling program
WriteSettings()	Takes the configuration information supplied by the user (via a Settings form) and writes it out to a DACONFIG.XML file
SetSettingsPath()	Send in the full path to a “writeable” folder where the DACONFIG.XML file should be stored

When the AlertManager class is instantiated, the Init() calls the Setup() method, which in turn calls the APIDeclare() and ReadSettings() methods. It also creates the Alerts collection.

We do all of the API declarations in the APIDeclare() method, even for all of the declarations needed just by the Alert form we’ve already created. This is why all the API-related code in the Alert form assumed the declarations have already been made.

At this point, the AlertManager is waiting for us to create individual alerts (using the NewAlert() method).

The Alert Class

The Alert (session) class is the big daddy that really does most of the work in creating alerts. It contains the following custom properties:

Property	Description
oMgr	Object reference to the AlertManager which instantiated this alert
oCallback	Reference to the object which is called back (we return the result of the alert back to the caller)
oAlertForm	Actual alert form which appears on the screen
nResult	The value to return to oCallback
nWait	Same as AlertManager.nWait
nPercent	Same as AlertManager.nPercent
lSound	Same as AlertManager.lSound
cSound	Same as AlertManager.cSound

The Alert Class also contains the following custom methods:


Method	Description
Setup()	Reads configuration information from oMgr, and stores values into the appropriate properties
SetCallback()	Handles setting up the reference to oCallback
Alert()	The big daddy method that parses parameters and creates oAlertForm
nResult_Assign	Fires the oCallback event handler

When the AlertManager instantiates a new Alert, the Alert sets the oMgr property and calls the Setup() method. The Setup() method retrieves the configuration settings, and calls SetCallback() with a parameter of NULL.

At this point, the Alert is waiting for the SetCallback() method to be called by the client app, so the Alert can fire the oCallback event handler once it has the Alert Result. Once the oCallback is set, the Alert waits for the Alert() method to be called.

When I create a new alert and call the Alert() method, I'm going to pass in some parameters so the Alert object can dynamically create the Alert Form.

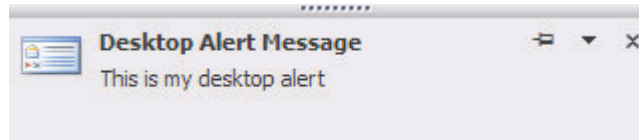
Here's the parameters I came up with:

Parameter	Description
cDetails	The actual text of the Details section of the alert
nAlertType	<p>The type of alert that will be created. A plain alert will have details, but the details will not appear to be a hyperlink. A link alert will make the details look like a hyperlink when the mouse moves over it.</p> <pre>#DEFINE DA_TYPEPLAIN 0 && No Links, No Tasks #define DA_TYPELINK 1 && One Link, No Tasks #define DA_TYPERTASK 2 && One Link, One Task #define DA_TYPERMULTI 4 && One Link, Two Tasks</pre>
cAlertTitle	Text of the Title. Defaults to "Desktop Alert Message"
cAlertSubject	Text of the Subject. Defaults to an empty string
cIconFile	<p>Filename of the icon to use (if 128 is passed as part of the nAlertType). Defaults to the following icon (which is included in the project, as are the Stop, Question, Exclamation, and Information icons):</p>  <p>If the nAlertType includes a value of 128 (custom graphic), this parameter should be the path and filename for the icon to use.</p> <p>We also have additional "built-in" icons we can use, which are also determined in the nAlertType parameter (like a MESSAGEBOX):</p> <pre>** Alert Icons #define DA_ICONDEFAULT 8 && Default Alert Icon #define DA_ICONSTOP 16 && Critical message #define DA_ICONQUESTION 32 && Question Mark #define DA_ICONEXCLAMATION 48 && Warning message #define DA_ICONINFORMATION 64 && Information message #define DA_ICONCUSTOM 128 && User-defined custom graphic</pre>
cTask1	Text to appear as a tooltip for Task One (also appears in the settings menu)
cTask1Icon	Icon file for Task One
cTask2	Text to appear as a tooltip for Task Two (also appears in the settings menu)
cTask2Icon	Icon file for Task Two

The only *required* parameter for an alert is cDetails. Using the example from earlier, the following command:

```
loAlert.Alert("This is my desktop alert.")
```

Shows the following alert on the screen:



This shows the default title and default icon, and the details line is plain text (no hyperlink).

Sometimes we may want to hide the Settings button, so we have a value defined in the VFPALERT.H include file:

```
#DEFINE DA_NOSETTINGS      4096      && Do not show the settings button
```

Passing a value of 4096 within the nAlertType parameter will cause the settings button to be hidden from the user.

This type of parameter system (in conjunction with known #DEFINED elements) allows a great amount of flexibility, at a small cost of complexity in parsing the parameter values.

Here's the complete listing of #DEFINES for the nAlertType parameter:

```
** VFP Desktop Alert Parameters
** Types of Alert (Major Form Properties)
#DEFINE DA_TYPEPLAIN      0      && No Links, No Tasks
#DEFINE DA_TYPELINK      1      && One Link, No Tasks
#DEFINE DA_TYPTASK      2      && One Link, One Task
#DEFINE DA_TYPMULTI      4      && One Link, Two Tasks

** Alert Icons
#DEFINE DA_ICONDEFAULT      8      && Default Alert Icon
#DEFINE DA_ICONSTOP      16      && Critical message
#DEFINE DA_ICONQUESTION      32      && Question Mark
#DEFINE DA_ICONEXCLAMATION      48      && Warning message
#DEFINE DA_ICONINFORMATION      64      && Information message
#DEFINE DA_ICONCUSTOM      128      && User-defined custom graphic

** Left some room in here to "build-in" some other 'default' icons.

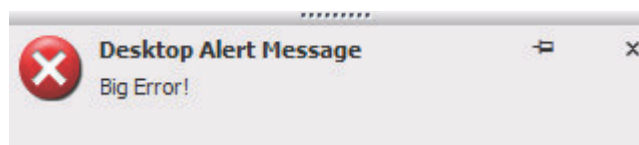
#DEFINE DA_TASKICONDEFAULT      2048      && Default Task Icon

** Alert Options
#DEFINE DA_NOSETTINGS      4096      && Do not show the settings button
```

We can add the #DEFINES (like a MESSAGEBOX) for nAlertType:

```
loAlert.Alert("Big Error!",DA_TYPEPLAIN+DA_ICONSTOP+DA_NOSETTINGS)
```

Which results in:



Since each of the DEFINED values correspond to a BIT value, the Alert() method performs a series of BITTESTs to determine what we want.

```
** Do we want to hide the settings button?
llHideSettings = BITTEST(nAlertType,12)

** What kind of ICON do we want?
FOR i = 7 TO 3 STEP -1
    IF BITTEST(nAlertType,i)
        IF i = 5
            ** Bit 5 is 32 (ICONQUESTION). We need to
            ** check to see if the next bit is "on".
            ** This will mean 48 was passed in, which
            ** is the value of ICONEXCLAMTION.
            IF BITTEST(nAlertType,i-1)
                ** Yes, bit 4 = 16. Add it to the 32.
                lnIcon = (2^i) + (2^(i-1))
                EXIT
            ELSE
                ** Nope, a 32 was passed in.
                lnIcon = 2^i
                EXIT
            ENDIF
        ELSE
            lnIcon = 2^i
            EXIT
        ENDIF
    ENDIF
ENDFOR

IF lnIcon = 0
    lnIcon = DA_ICONDEFAULT
ENDIF
```

Once we have the icon “set”, we determine the Alert Type:

```
** Now, check the Alert Type, which
** will be in either BIT 2 or 1.
DO CASE
    CASE BITTEST(nAlertType,2)
        ** MultiTask (4)
        lnType = DA_TYPEMULTI
    CASE BITTEST(nAlertType,1)
        ** Task (2)
        lnType = DA_TYPTASK
    CASE BITTEST(nAlertType,0)
        ** Link (1)
        lnType = DA_TYPELINK
    OTHERWISE
        ** Plain (0)
        lnType = DA_TYPEPLAIN
ENDCASE
```

At this point, the Alert() method will create a parameter object to pass to our Alert Form. I’m using an Empty class, and adding the properties I need.

We need to walk through all remaining parameters, and fill in defaults where necessary:

```
** We have the TYPE, and the ICON. Create a
** parameter Object to pass to the alert form.
LOCAL loParams AS Object
loParams = CREATEOBJECT("Empty")

ADDPROPERTY(loParams,"Name",lcName)
ADDPROPERTY(loParams,"Type",lnType)
ADDPROPERTY(loParams,"Icon",lnIcon)
ADDPROPERTY(loParams,"IconFile","")
ADDPROPERTY(loParams,"Title","")
ADDPROPERTY(loParams,"Subject","")
ADDPROPERTY(loParams,"AlertText",cAlertText)
ADDPROPERTY(loParams,"HideSettings",llHideSettings)

IF lnType > DA_TYPELINK
    ADDPROPERTY(loParams,"Task1","")
    ADDPROPERTY(loParams,"Task1Icon","")
    IF lnType > DA_TYPETASK
        ADDPROPERTY(loParams,"Task2","")
        ADDPROPERTY(loParams,"Task2Icon","")
    ENDIF
ENDIF

** We need to "walk" through the rest of the parameters
IF VARTYPE(cAlertTitle) # "C"
    IF VARTYPE(cAlertTitle) = "L"
        loParams.Title = DA_DEFAULTTITLE
    ELSE
        loParams.Title = SUBSTR(TRANSFORM(cAlertTitle),1,30)
    ENDIF
ELSE
    loParams.Title = SUBSTR(cAlertTitle,1,30)
ENDIF

IF VARTYPE(cAlertSubject) # "C"
    IF VARTYPE(cAlertSubject) = "L"
        loParams.Subject = ""
    ELSE
        loParams.Subject = SUBSTR(TRANSFORM(cAlertSubject),1,50)
    ENDIF
ELSE
    loParams.Subject = SUBSTR(cAlertSubject,1,50)
ENDIF

IF VARTYPE(cIconFile) # "C"
    loParams.IconFile = DA_DEFAULTICONFILE
ELSE
    IF FILE(cIconFile)
        loParams.IconFile = cIconFile
    ELSE
        loParams.IconFile = DA_DEFAULTICONFILE
    ENDIF
ENDIF
```

```

** Check for Tasks and Task Icons
IF lnType > DA_TYPELINK
    ** The next param should be the Task1
    IF VARTYPE(cTask1) # "C"
        IF VARTYPE(cTask1) = "L"
            loParams.Task1 = ""
        ELSE
            loParams.Task1 = TRANSFORM(cTask1)
        ENDIF
    ELSE
        loParams.Task1 = ALLTRIM(cTask1)
    ENDIF

    ** The next param should be the Task1Icon
    IF !EMPTY(ALLTRIM(loParams.Task1))
        IF VARTYPE(cTask1Icon) # "C"
            loParams.Task1Icon = DA_DEFAULTTASKICON
        ELSE
            IF FILE(cTask1Icon)
                loParams.Task1Icon = cTask1Icon
            ELSE
                loParams.Task1Icon = DA_DEFAULTTASKICON
            ENDIF
        ENDIF
    ENDIF

    ** If a Multi Task, the next param
    ** should be Task2
    IF lnType = DA_TYPEMULTI
        IF VARTYPE(cTask2) # "C"
            IF VARTYPE(cTask2) = "L"
                loParams.Task2 = ""
            ELSE
                loParams.Task2 = TRANSFORM(cTask2)
            ENDIF
        ELSE
            loParams.Task2 = ALLTRIM(cTask2)
        ENDIF

        IF !EMPTY(ALLTRIM(loParams.Task2))
            IF VARTYPE(cTask2Icon) # "C"
                loParams.Task2Icon = DA_DEFAULTTASKICON
            ELSE
                IF FILE(cTask2Icon)
                    loParams.Task2Icon = cTask2Icon
                ELSE
                    loParams.Task2Icon = DA_DEFAULTTASKICON
                ENDIF
            ENDIF
        ENDIF
    ENDIF
ENDIF
ENDIF

```

*** It's a lot of code. As I said before, this ain't no code clinic. The goal here is to work out all the parameters before we create the Alert form.*

We've parsed the parameters and placed them into a parameter object. Now, let's create the Alert Form:

```
** Create an instance of the alert form, passing in the
** loParams object and a reference to THIS alert.
This.oAlertForm = CREATEOBJECT("frmAlert",loParams,THIS)
```

At this point, the Alert object is done with its work. It simply waits for a result from the Alert form, then passes the result back to the oCallback, and releases itself (which in turn, removes it from the AlertManager's Alerts collection).

Returning the Alert Result Back to the Caller

The tricky part of returning the Alert Result back to the calling applications is the SetCallback() method in the Alert (session-based) class.

SetCallback() will attempt to "link" the DAS with the calling app via an interface definition. The interface is handled by a class called AlertEvents:

```
DEFINE CLASS AlertEvents AS Session OLEPUBLIC
    PROCEDURE AlertResult (tnResult AS Integer) AS Integer
    ENDPROC
ENDDEFINE
```

Here's the order of what's happening:

- The AlertManager creates an instance of Alert.
- The calling app (or object) calls Alert.SetCallback(), passing in a reference to itself.
- The Alert Form is "built" based on parameters passed in by the caller, which are parsed by the Alert method().
- The Alert Form returns the Alert Result back to the Alert class
- The Alert class uses an nResult_Assign() method to "fire" the calling object's AlertResult() method

Here's the code from the Alert.nResult_Assign():

```
PROCEDURE nResult_Assign
    LPARAMETERS vNewValue
    This.nResult = m.vNewValue

    LOCAL loException AS Exception
    loException = .NULL.

    IF This.nResult # 0
        TRY
            This.oCallback.AlertResult(This.nResult)
        CATCH TO loException
        ENDTRY
    ENDIF
ENDPROC
```

The reference to oCallback is set during the Alert's SetCallback() method. We'll wrap the call to the oCallback.AlertResult() in a TRY/CATCH/ENDTRY, just in case we run into some sort of error.

Here's the code for the SetCallback() method:

```
PROCEDURE SetCallback (toCallback AS Variant) AS VOID ;
    HELPSTRING "Gets/Sets a reference to the client event handler"

LOCAL loException AS Exception
loException = .NULL.

IF ISNULL(toCallback)
    ** Dummy instance that does nothing: virtual function
    This.oCallback = CREATEOBJECT("AlertEvents")
ELSE
    IF VARTYPE(toCallback) # "O"
        COMRETURNERROR("Function SetCallback()", "Callback object must be
            an Object")
    ENDIF

    TRY
        This.oCallback = ;
        GETINTERFACE(toCallback, "Ialertevents", "vfpalert.AlertManager")
    CATCH TO loException
    ENDTRY

    IF !ISNULL(loException)
        ** An exception was created on the GETINTERFACE line.
        ** Reference the object that came in, and hope for the best.
        This.oCallback = toCallback
    ENDIF
ENDIF
ENDPROC
```

Remember the AlertEvents class with the one method named AlertResult? THAT is what takes care of the callback.

Since we are publishing an interface called iAlertEvents (from this OLEPUBLIC class), any COM client can IMPLEMENT the interface, and automatically respond to the AlertResult!

If we get an error of some kind (or if we're using DAS without the ability to IMPLEMENT the interface), we simply set oCallback to the object that was passed in. ***This assumes the oCallback will have a method called AlertResult!!!*** Sorry, that's the best I could come up with for my purposes. I said this wouldn't be a code clinic!

Just for fun, let's create a small program to take see the different results/options available to us. Once the program runs, we'll see the following three alerts on the screen (starting in the lower-right corner of the desktop):

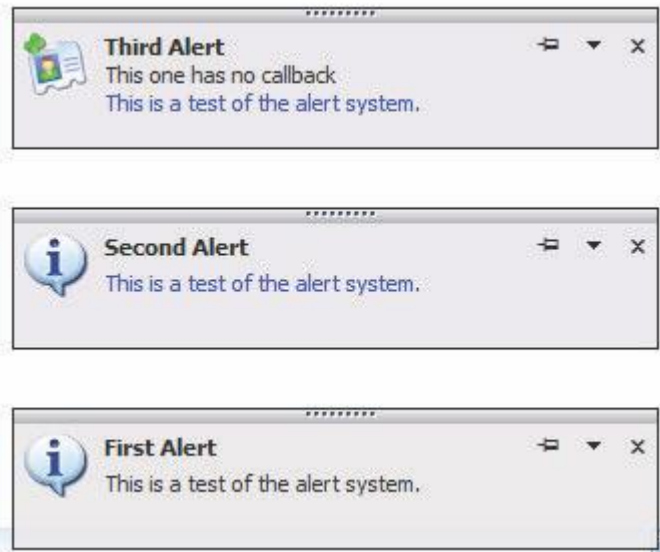


Figure 5: Demonstration of 3 Alerts

Remember, the Alert form calculates its position in the SetupForm() method, though we didn't cover that in this document.

Here's the code required to launch these three alerts:

```
LOCAL oMgr AS vfpalert.AlertManager, ;
    o AS vfpalert.Alert, ;
    o2 AS vfpalert.Alert, ;
    o3 AS vfpalert.Alert

** Create an event handler for the first alert (o)
x=CREATEOBJECT("myregularclass")

** Create an event handler for the second alert (o2)
y=NEWOBJECT("myclass")

** We won't do one for the third alert (o3)

** Create the AlertManager
oMgr = NEWOBJECT("vfpalert.AlertManager")

** First alert
o = oMgr.NewAlert()

** Second alert
o2 = oMgr.NewAlert()

** Third alert
o3 = oMgr.NewAlert()

** SetCallback() for the first two alerts
o.SetCallback(x)
o2.SetCallback(y)

** Launch the first alert form
```

```

o.Alert("This is a test of the alert system.",64,"First Alert")
INKEY(.5,"hc")

** Second alert form
o2.Alert("This is a test of the alert system.",65,"Second Alert")
INKEY(.5,"hc")

** Third alert form
o3.Alert("This is a test of the alert system.",129,"Third Alert","This one
has no callback","f:\test.ico")

** Just for demonstration purposes, 'hang' the system
** long enough to see the results.
WAIT WINDOW "" TIMEOUT 20

DEFINE CLASS myregularclass AS session
    PROCEDURE AlertResult(tnResult AS Number) AS Number
        MESSAGEBOX("You selected: " + TRANSFORM(tnresult) + " using
AlertResult() in a normal class",64,"MyRegularClass")
    ENDPROC
ENDDEFINE

DEFINE CLASS myclass AS session OLEPUBLIC
    IMPLEMENTS Ialertevents IN "vfpalert.AlertEvents"

    PROCEDURE Ialertevents_AlertResult(tnResult AS Number) AS Number
        MESSAGEBOX("You selected: " + TRANSFORM(tnresult) + " in the
Ialertevents.",64,"iAlertEvents_AlertResult")
    ENDPROC
ENDDEFINE

```

For the alerts, I'm going to perform the following actions:

Alert 1 (o): I'll do nothing. I'll let the Alert "timeout" after the default ten seconds

Alert 2 (o2): I'll click the Details 'link'

Alert 3 (o3): I'll click the Close button.

Here are the results of my actions with the alerts:



Figure 6: Result of Alert 1



Figure 7: Result of Alert 2

As you can see, there is NO result from Alert 3 (o3). Why? Because it had no event handler. We never did a SetCallback(), and since the Alert initially sets the callback object to .NULL. it creates an AlertEvents object which in effect *does nothing*. The alert appears, the user interacts, but nothing happens as a result of the alert.

Conclusion

Though DAS can be extremely useful, it is important to use Desktop Alerts sparingly. I find it best to use them when I need to present information to the user without *requiring* a response. Hence, no unnecessary modal dialogs.

With VFP 7 or later, you can drag and drop the iAlertEvents interface directly from the Object Browser and you are ready to receive the Alert results in your applications. For VFP 6, you'll need to create event handler objects with an AlertResult() method (you can even add the method to existing objects, and pass the object to the Alert's SetCallback() method.

During the Desktop Alerts presentation, we'll create an alert directly from the PowerPoint presentation, using a minimal amount of code:

```
Dim oAlertMgr As New vfpalert.AlertManager
Dim oAlert As Object
Dim Alert As Variant
Dim oCB As Variant

Public Sub AlertResult(ByVal nResult As Integer)
    Me.txtResult.Text = "The result was " & Trim(Str(nResult))
End Sub

Private Sub cmdGo_Click()
    Set oCB = Me

    Set oAlert = oAlertMgr.NewAlert()
    oAlert.SetCallback (oCB)

    Alert = oAlert.Alert(Me.txtDetails.Text, Val(Me.txtStyle.Text),
Me.txtTitle.Text, Me.txtSubject.Text, Me.txtIcon.Text)
End Sub
```

By utilizing the amazing power within Visual FoxPro, we can do anything!