THE CODEBOOK NEWS

November - December 1997

Volume 1, Issue 4

TABLE OF CONTENTS

Editor's Comments

Charles T. Blankenship

The production of TCN has hit equilibrium now. I had a feeling when this started that trying to make it a monthly publication was a bit aggressive. Therefore, this publication must become bi-monthly in order to maintain sanity as well as quality. Publication dates are scheduled for the end of each even month, February, April, June, August, October and December. If more people decide they want to contribute however, this decision may be revisited in the future.

The first article, **The Message Object Library: Application Wide User Communication**, covers how you can consolidate all application messaging through one group of messaging objects. The purpose of this is to normalize code just like a database is normalized. The goal is to have one type of functionality provided by one object. Now, all messaging is channeled through one location which makes for easier maintenance ... as described in the article.

The second article, **Fixing QSTART.APP**, describes a way to "fix" QSTART.APP. On several occasions, after placing debugging code in the startup program for QSTART.APP, I received the error message that informed me the source code was not available. This unfortunate occurrence prevented tracing through the application to see how it really worked (many nice processing treasures are stored in there). This article explains how to get that application up and running again as normal.

The third article, **Creating A New Application Using QSTART.APP**, may seem like a useless topic at first glance, after all this thing has been out forever ... right? Anyone using Codebook has used this .APP about a kabillion times. Well, don't short change this article so quickly. It contains a trick I now use to make a Codebook application portable from drive to drive with no effort involved. This is nice since I do development in at least four locations, my NT server, my laptop, my ZIP drive and on the client's server. This saved many gray hairs.

The fourth article, **cReportForm Enhancements**, shows how to redirct the Print To File option for Codebook standard report printing to a new location and OUT of the application root directory. My client required a method for placing a user's printed output into their network user directory. This article documents how I did that.

The last articles, **A New LockScreen()** and **An Even Better LockScreen**, authored by Ed Leafe, illustrates new, cleaner implementations of the old LockScreen() function that is stored in the UTILITY.PRG. These enhancements make sure the .LockScreen property is always reset to its original condition with a minimum amount of effort on the part of the programmer. These articles represent two way of accomplishing the same task.

The Message Object Library: Application Wide User Communication

Charles T. Blankenship

Reuse is a frequently discussed concept in object oriented application development. In Visual FoxPro, the opportunities for reuse abound. One key area that can benefit from reuse is the method used to communicate with the user. VFP provides many nice features to facilitate user communication, but many developers, due to the simplicity of the commands (WAIT WINDOW, SET MESSAGE TO and MESSAGEBOX( )) needlessly duplicate code throughout the application. This article describes how to decrease code duplication and as a result, the resources required to maintain finished applications.

In order to determine if you are experiencing the maximum reuse from your software designs consider your response to the following request: "What would it take to have a bell sound as each WAIT WINDOW or MESSAGEBOX is displayed in the application"? If your answer is more than 15 seconds then you have a problem (or no WAIT WINDOWS or MESSAGEBOXes in your application). If each WAIT WINDOW was hand coded, possibly hundreds of times throughout the entire application, this could possibly take many hours to complete. This is a completely unacceptable answer. It does, however, serve to illustrate several points. The first is that many developers are not realizing the degree of reuse they should be. The second is that developers frequently apply object oriented concepts to solve complex problems in business applications but disregard them to solve the more mundane development tasks they face every day.

This article illustrates a class library whose primary function is to facilitate user communication in such a way so as to increase code reuse, decrease the resources required to maintain an application and increase the user friendliness of the finished application's messaging.

Class requirements

The first step in any development effort should be the collection of the application's requirements. This is particularly important in an object oriented application in order to discover the common properties and behaviors that will be used to define the application's class hierarchy. The following are the requirements for the message object library, CSAVIMSG.PRG ... it is responsible for:

1. controlling all communications that take place from the application to the user and back again. This includes all major forms of messaging available in VFP, WAIT WINDOW, SET MESSAGE TO and MESSAGEBOX().
2. creating a common public signature which makes using each object simpler
3. ensuring that the total functionality of each wrapped messaging command is easily accessible to the developer.
4. ensuring that each communication object is flexible enough to be called upon for one time use (using the CREATEOBJ() command) as well as attachable to a global application object where they are available for the life of the session.

5. providing the developer with a place to define a default configuration for that messaging object. The object of this last requirement is to reduce each message signature to the following in as many locations as possible:

```
goApp.<messageobject>.Show( <cMessage> )
```

Notice, in the code example provided above, that the only requirements for producing an application message is the signature of the messaging object and the text to display. If no other parameters are passed, then the default characteristics of the WAIT WINDOW, MESSAGEBOX() or SET MESSAGE TO objects take over. This reduces the amount of keystrokes necessary to create a message while increasing flexibility by allowing the developer to define their default message style vice having to rely on VFP's default message style.

The WAIT WINDOW wrapper class must be able to display the developer specified message, capture a keystroke from the user and return that information to the calling program, be displayed at the specified screen coordinates and allow the developer to specify the duration of the message's display.

The SET MESSAGE TO wrapper class must be able to display a message on the status bar and remove it when the object goes out of scope or a pre-defined time limit has expired and replace the new message with the old message once the object is destroyed. It also must be configurable to display a more permanent message on the status bar, one that will only be removed when another message is displayed.

The MESSAGEBOX( ) wrapper class must be able to display the developer specified message, simplify the parameters required to configure the message box by providing a more English-like syntax, and return the identifier of the button that was pushed.

Superclass design

The first step in designing an object oriented solution to a problem is to identify common properties and behaviors *of* the problem. The second is to design and create a class that captures that functionality. After examining the requirements of the messaging class library, I identified the following properties and behaviors as common to all forms of messaging.

The first is that all forms of user communication displays message text.. This information must be validated to ensure that the message is character in type and is not empty. The second is that all messaging types must possess an activation method. This requirement, commonly known as polymorphism, makes the messaging class library easier to learn and use. Since the purpose of this behavior is to show the message to the user, an appropriate name for this polymorphic method is SHOW.

The following is the listing for the SAVIBaseMessage superclass that defines the attributes and behaviors common to all messaging classes.

Listing 1: SAVIBaseMessage superclass (created programmatically for illustrative purposes)

```
DEFINE CLASS SAVIBaseMessage AS Custom

*------------------------
*-- List member properties
*------------------------
cMessage = ""

FUNCTION Init()
************************************************
*)  Description:
*)    This class is an abstract class meant to
*)    be subclassed before it is fully
*)    operational.  Therefore, prevent in-
*)    stantiation if this class is being used to
*)    create an object directly.
*   Scope: PUBLIC
*   Parameters: None
*$  Usage: Automatically called at instantiation.
*%  Example: N/A
*   Returns:  LOGICAL
*     .T. if subclassed and usable
*     .F. if not.
************************************************
LOCAL llRetVal
llRetVal = .F.

*------------------
*-- Test for Success
*------------------
llRetVal = THIS.IsNotAbstract( "SAVIBaseMessage" )

RETURN llRetVal
ENDFUNC

PROTECTED FUNCTION IsNotAbstract
************************************************
*)  Description:
*)    This method determines if the an object
*)    is being created directly from this
*)    class definition by comparing the name
*)    of the class being created to the name
*)    of the class that should not be used
*)    passed to this method via parameter.
*   Scope: PUBLIC
*   Parameters:
*     1. <cClassName> - the name of the class
*        that cannot be used to directly
*        instantiate an object.
*$  Usage:
*$  cSAVIBaseMessage::IsNotAbstract(<cClassName>)
```

```
*%  Example:
*%  THIS.IsNotAbstract('SAVIBaseMessage")
*   Returns:  LOGICAL
*     .T. if subclassed and usable
*     .F. if an object is directly trying to
*     create itself from this class definition.
*   Notes:
*    Modified from code originally published with
*    Codebook and reprinted with permission.
************************************************
LPARAMETERS tcClassName
LOCAL llRetVal
llRetVal = .T.
IF UPPER(THIS.Class) = UPPER(tcClassName)
   ??CHR(7)
   WAIT WINDOW "Cannot instantiate class '" + ;
         ALLTRIM(THIS.Class) + ;
         "' directly!" TIMEOUT 2
   llRetVal = .F.
ENDIF
RETURN llRetVal
ENDFUNC

PROTECTED FUNCTION ValidateMessage()
*-------------- Usage Section ------------------
*)  Description:
*)    This method is responsible for validating
*)    the proposed message text and populating
*)    the cMessage member property with the text
*)    to be used to display the message.
*   Scope: PROTECTED
*   Parameters:
*     1.  <cMessage> - the text that
*          must be validated before it can
*          be used to message the user.
*$  Usage:
*$    SAVIBaseMessage( <cMessage> )
*%  Example:
*%    THIS.ValidateMessage( tcMessage )
*   Returns:  LOGICAL .T. - either the
*        message is valid or the applicable
*        error message takes its place.
************************************************
LPARAMETER tcMessage
LOCAL llRetVal
llRetVal = .T.

DO CASE
CASE TYPE('tcMessage') <> "C"
   THIS.cMessage = ;
       "BaseMessage::ValidateMessage Error: " + ;
        "Message must be character"
CASE EMPTY(tcMessage)
```

```
   THIS.cMessage = ;
       "BaseMessage::ValidateMessage Error: " + ;
       "Message cannot be empty"
OTHERWISE
   THIS.cMessage = tcMessage
ENDCASE

RETURN llRetVal

ENDFUNC
```

```
FUNCTION Show()
*-----------------------------------------------
*)  Description:
*)    This method is an abstract method that is
*)    simply a place holder for the code that
*)    SHOWS each message type, Wait Window,
*)    Status Bar, MessageBox and Trapped Error.
*)    This ensures that the name of the method
*)    for displaying the message remains constant
*)    over the entire class hierarchy.
*   Scope: PUBLIC
*   Parameters: None (Abstract)
*$  Usage: N/A
*%  Example: N/A
*   Returns:  LOGICAL .T. by default
*************************************************
ENDFUNC

ENDDEFINE
```

The WAIT WINDOW object

The are two signatures that grant access to the functionality contained in the SAVIWaitWindow wrapper class. The first one assumes that the wait window object is a child of the application object, the suggested approach, which makes its functionality available throughout the application without having to incur the cost of instantiating it every time its functionality is needed.

Listing 2: Instantiating the WAIT WINDOW object for persistent use

```
goApp.oWaitWindow = CREATEOBJ('SAVIWaitWindow')
lcRetVal = goApp.oWaitWindow.Show( <cMessage>     , ;
                                  [<cWaitStatus>] , ;
                                  [<cLocation>]     )
```

When the Wait Window object is being created for persistent use , no parameter is passed to the Init method. This causes the object to instantiate as normal and a reference is passed back to the calling program to the goApp.oWaitWindow property. Once a reference to the Wait Window object is stored in a member property of the application, it is always available for future use. To display a Wait Window, simply call the Show method of the object and pass it the message text, the wait status that you desire and the location where you want the Wait Window to appear. A return value from a call to the Show method is either an empty string if no key was pressed, or the character representation of the key the user did press.

The second signature creates the Wait Window object for one time use. The object is created, the message is displayed and the object is never allowed to completely instantiate since a .F. is returned from the Init method.

Listing 3: Instantiating the WAIT WINDOW object for one time use

```
lcRetVal = ""
=CREATEOBJ('SAVIWaitWindow', <cMessage>     , ;
```

```
[<cWaitStatus>] , ;
[<cLocation>]     )
```

When instantiating the object for one time use, one or more parameters are passed into the Init method. If the first one is of character type The Show method is called and passed the desired parameters. Once the message is displayed, processing returns to the Init method where the return value is set to .F., which prevents instantiation of the object.

Obtaining a return value from this type of call is not very straight forward. Unfortunately, since the return value from the CREATEOBJ function is either a reference to the object or a logical value, the only way to obtain the return value from the call to the Show method (inside the wait window's Init method) was to leave the lcRetVal variable unlocalized. In this case, when lcRetVal is populated by the return value of the Show method, that value appears in the lcRetVal variable contained in the calling method as well.

Defining organizational standards for the WAIT WINDOW object

A primary objective of this class definition is to allow the developer to specify what their default settings for WAIT WINDOWS should be. This is provided by populating the cDefaultWaitStatus property with the value that specifies the default duration of the wait window and populating the cDefaultLocation property with the column and row of the default location. Rules for populating these properties follow the exact same syntax as the WAIT WINDOW command itself.

To define the wait status of the wait window, place one of the following initial values into the cDefaultWaitStatus property. Making "NOWAIT" the initial value for this property causes execution to continue without clearing the wait window. Defining "TIMEOUT n" allows the developers to define the number of seconds the wait window hangs around until it disappears. And finally, placing the empty string in this property allows the developer to specify that the wait window should remain in view, suspending application execution, until a key is pressed or the mouse is moved.

To define the default location of the wait window when it is displayed, populate the cDefaultLocation property with the row and Column where the wait window should appear. The syntax for this property is <cRow,cColumn>, e.g. "2,30" to have the wait window appear in the second row and the thirtieth column. Leave this property blank for the wait window to appear in its default location, the upper right hand portion of the screen.

Overriding Default Values

To override the default values for a special purpose, simply provide the desired settings via the parameters when the Show method of the object is called. The values in the parameters take precedence over the default values initialized in the class definition. Now when a client asks how long it will take to sound a bell before the display of a wait window the answer becomes "Oh, about 15 seconds … do you have that much money?!?!"

Listing 4: The Wait Window Wrapper Class Definition

```
*================================================
*=============== SAVI Wait Window ===============
*================================================
*-- Class:       SAVIWaitWindow
*-- ParentClass: SAVIBaseMessage
*-- BaseClass:   Custom
*================================================
DEFINE CLASS SAVIWaitWindow AS SAVIBaseMessage
********* Member Properties Defined **********
*-- cDefaultWaitStatus - allows the developer
*--   to specify the default type of wait
*--   window to display (if not specified
*--   via a parameter)
*--    ""          - Wait
*--    "NOWAIT"    - Display and continue with
*--                  processing
*--    "TIMEOUT n" - times out after n seconds
*-- cWaitStatus - stores the validated value
*--   of the NOWAIT or TIMEOUT class that can
*--   be tacked onto the end of a wait window.
*-- cLocation - specifies the location where
*--   the wait window should be displayed.
*--   No parameter provided means wait window
*--   displayed at top right default position
*-- Name - the name of the object, used when
*--   referencing the object in code.
*********************************************
PROTECTED cDefaultWaitStatus

cDefaultWaitStatus = "TIMEOUT 2"
cDefaultLocation   = ""
cWaitStatus        = ""
cLocation          = ""
Name               = "SAVIWaitWindow"


PROCEDURE Init
*********************************************
*)  Description:
*)    Creates the Wait Window object if no
*)    message parameter is passed. If a
*)    message parameter is passed, this
*)    object assumes it is being employed
*)    for a one shot use, displays the
*)    message and releases the object.
*   Scope: PUBLIC
*   Parameters:
*     1.  <cMessage> - the message to display
*             in the wait window
*     2.  [<cWaitStatus>] - the duration of
*             the wait window
*         a.  This parameter directly copies
*             the string you would place on
*             the end of the WAIT WINDOW
*             command
*         b.  "NOWAIT" - displays the wait
*             window and continues with
*             processing
```

```
*         c.  "TIMEOUT n" - displays the wait
*             window for the specified
*             "n" number of seconds.
*     Default behaviour is a wait window that
*     waits (omit the <cWaitStatus> parameter)
*$  Usage:
*$     =CREATEOBJ( "SAVIWaitWindow" , ;
*$                 <cMessage>       , ;
*$                 <cWaitStatus>    , ;
*$                 <cLocation>        )
*$
*%  Example:
*%     =CREATEOBJ( "SAVIWaitWindow"      , ;
*%                 'This is the message' , ;
*%                 'TIMEOUT 2'           , ;
*%                 '2,2'                   )
*%     (Note: this is not the recommended
*$%           use of this object !!! )
*   Returns:  LOGICAL .T. by default
*********************************************
LPARAMETERS tcMessage    , ;
            tcWaitStatus , ;
            tcLocation

LOCAL llRetVal
*--------------------------------------
*-- Do not LOCALize lcRetVal, this is the
*-- only way to get a return value out
*-- of the "one shot use" method for this
*-- class
*--------------------------------------
llRetVal = .T.

IF TYPE('tcMessage') = "C"
   lcRetVal = ""
   lcRetVal = THIS.Show( tcMessage    , ;
                         tcWaitStatus , ;
                         tcLocation     )
   llRetVal = .F.
ENDIF

RETURN llRetVal

ENDPROC

PROCEDURE Show
*********************************************
*)  Description:
*)    This method validates and executes the
*)    wait window command.
*   Scope: PUBLIC
*   Parameters:
*     1.  <cMessage> - The message to display
*                      in the wait window
*     2.  [<cWaitStatus>] - The duration of
*             the wait window
*         a.  EMPTY  - standard waiting wait
*                      window
*         b.  "NOWAIT" - display window,
*                     continue with processing
```

```
*         c.  "TIMEOUT n" - display window for
*                   n seconds then continue
*                   with processing.
*     3.  [<cLocation>] - the location where
*                   the wait window should be
*                   displayed, follows the
*                  (<nRow>,<nCol>) convention
*$  Usage:
*$  SAVIWaitWindow::Show( <cMessage>      , ;
*$                       [<cWaitStatus>]  , ;
*$                       [<cLocation>]      )
*%  Example:
*%     goApp.oWaitWindow.Show( "Message"   , ;
*%                             "TIMEOUT 2" , ;
*%                             "2,2" )
*   Returns:  LOGICAL .T. by default
*********************************************
LPARAMETERS tcMessage    , ;
            tcWaitStatus , ;
            tcLocation

LOCAL lcRetVal , ;
      llRetVal , ;
      lcWaitWindowCommand, ;
      llLocationProvided


lcRetVal = ""
THIS.ValidateMessage ( tcMessage )
THIS.ValidateWaitStatus( tcWaitStatus )
THIS.ValidateLocation( tcLocation )

lcWaitWindowCommand = "WAIT WINDOW '"   + ;
                      THIS.cMessage    + ;
                      "' "             + ;
                      THIS.cWaitStatus + ;
                      THIS.cLocation   + ;
                      " TO lcRetVal"


&lcWaitWindowCommand

RETURN lcRetVal

ENDPROC


PROTECTED PROCEDURE ValidateWaitStatus
*********************************************
*)  Description:
*)    Ensure that the Wait Status is valid and
*)    place a valid value in the cWaitStatus
*)    property
*   Scope: PROTECTED
*   Parameters:
*     1. <cWaitStatus> - the duration of the
*           WAIT WINDOW display to be
*           validated
*%  Example:
*%    THIS.ValidateWaitStatus( tcMessage )
```

```
*   Returns:  LOGICAL .T. by default
*********************************************
LPARAMETER tcWaitStatus

LOCAL lcValidWaitStatus, ;
      lcWaitStatus, ;
      llRetVal

llRetVal     = .T.

IF TYPE('tcWaitStatus') == 'C'
   lcWaitStatus = UPPER(ALLTRIM( tcWaitStatus ))
ELSE
   lcWaitStatus = THIS.cDefaultWaitStatus
ENDIF

lcValidWaitStatus= " " + ;
                   ALLTRIM(UPPER(lcWaitStatus))+;
                   " "
DO CASE
CASE " NOWAIT " == ALLTRIM(UPPER( lcWaitStatus ))
CASE " WAIT "   == ALLTRIM(UPPER( lcWaitStatus ))
CASE "TIMEOUT" $ ALLTRIM(UPPER( lcWaitStatus ))
OTHERWISE
   lcValidWaitStatus = THIS.cDefaultWaitStatus
ENDCASE

THIS.cWaitStatus = lcValidWaitStatus

RETURN llRetVal

ENDPROC

PROTECTED PROCEDURE ValidateLocation
*********************************************
*)  Description:
*)    Ensure that the Location provided by the
*)    developer is a valid location signature.
*   Scope: PROTECTED
*   Parameters:
*     1.  <cLocation> - the location of the wait
*           window in <nRow>,<nCol> format
*$  Usage:
*$ SAVIWaitWindow::ValidateLocation(<cLocation>)
*% Example:
*%   THIS.ValidateLocation( tcLocation )
*  Returns:  LOGICAL .T. by default
*********************************************
LPARAMETER tcLocation

LOCAL lcLocation, ;
      llRetVal, ;
      lcFinalCheck

llRetVal   = .T.
*-----------------------------------------
*-- Assume the default location will be used
*-----------------------------------------
lcLocation = THIS.cDefaultLocation
```

```
DO CASE
CASE TYPE('tcLocation') <> "C"
CASE .NOT. ( "," $ tcLocation )
OTHERWISE
 lcFinalCheck = UPPER(ALLTRIM(tcLocation))
 lcFinalCheck = ;
    CHRTRAN( lcFinalCheck                  , ;
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ123456789," , ;
    "XXXXXXXXXXXXXXXXXXXXXXXXXX9999999999"   )
 IF AT( "X" , lcFinalCheck ) > 0
    *-------------------------------------
    *-- Invalid information provided, leave
    *-- default location selected, 12,22
    *-- would resolve to 9999 (no "X")
    *-------------------------------------
 ELSE
   lcLocation = tcLocation
 ENDIF

ENDCASE

IF .NOT. EMPTY(lcLocation)
   lcLocation = " AT " + lcLocation
ENDIF

THIS.cLocation = lcLocation

RETURN llRetVal
ENDPROC

PROCEDURE Destroy
************************************************
*)  Description:
*)    This method clears any wait window that may
*)    be displayed when the object is released and
*)    then calls superclass functionality.
*   Scope: PUBLIC
*   Parameters: None
*$  Usage: Automatically called when object goes
*$         out of focus
*%  Example: N/A
*   Returns:  LOGICAL .T. by default
************************************************
WAIT CLEAR
SAVIBaseMessage::Destroy()

ENDPROC

ENDDEFINE
```

The STATUS BAR object

The STATUS BAR messaging object has the same two usage options available to the WAIT WINDOW messaging object. It can be instantiated for one time use or it can be attached to the application object and remain available throughout the application's execution. The signatures for using this messaging class follow:

Listing 5: Instantiating the STATUSBAR object for persistent use

```
goApp.oStatusBar = CREATEOBJ('SAVIStatusBar')

lcRetVal = goApp.oStatusBar.Show( <cMessage>   , ;
                                  [<xTimeOut>] )
```

Listing 6: Instantiating the STATUSBAR object for one time use

```
lcRetVal = ""
=CREATEOBJ('SAVIStatusBar', <cMessage>  , ;
                            [<xTimeOut>]  )
```

The duration of the STATUS BAR message can be controlled by the developer through the use of the xTimeOut parameter. This parameter, whose value can either be Logical or Numeric, allows the developer to specify that the message should display for a specified number of seconds (the Numeric value) before the previous message is replaced, or that the message should display permanently and never redisplay the old status bar message (the Logical value). Notice that either a .T. or a .F. or omitting the xTimeout parameter completely, is interpreted as a request for a permanent message.

Defining organizational standards for the STATUS BAR object

To define your company's default values for the STATUS BAR object populate the nDefaultTimeout property. Place a numeric value in this property if the STATUS BAR object should display the message for a specified period of time and then replace the previous status bar message. Place a logical value in this property if the STATUS BAR object should display a permanent message that does not go away until another message is displayed.

Listing 7: The SAVI Status Bar class definition

```
DEFINE CLASS SAVIStatusBar AS SAVIBaseMessage

*-----------------------------------------------
*-- cOldMessage - stores the value of the status
*--               bar before the new message was
*--               requested.
*-- xDefaultTimeout - the default configuration
*--            for the message bar's persistence.
*--            Logical (.T. or .F.) - permanent
*--            message, the old message is never
*--            replaced.
*--            Numeric - the duration in seconds
*--            of the message's display time.
*-- xTimeOut - stores either the duration the
*--            message is displayed on the
*--            messagebar OR the desire for the
*--            message to be permanent.  This
*--            property can either by logical or
*--            numeric.
*-- cTimeoutErrorMessage - the ValidateTimeout
*--            method is responsible for
*--            discovering the TYPE of the
*--            txTimeOut parameter.  The only
*--            valid values are Logical and
*--            Numeric.  If any other type exists
*--            it is supposed to use the value
*--            provided by the developer in
*--            xDefaultTimeout.  Therefore,
*--            valid or not, the only types of
*--            variables present in the xTimeout
*--            member property shold be Logical or
*--            Numeric.  This is how the SHOW
*--            method determines what it should do
*--            IF anything else exists, an error
*--            message displays indicating a
*--            problem.  Put your error message
*--            in this property.
*-----------------------------------------------
PROTECTED coldmessage     , ;
          xDefaultTimeout , ;
          xTimeout        , ;
          cTimeoutErrorMessage

cOldMessage          = ""
xDefaultTimeout      = .F.
xTimeOut             = .F.
Name                 = "SAVIStatusBar"
cTimeoutErrorMessage = ;
                "Unknown Timeout Value ... " + ;
                "please contact developer!"
```

```
PROCEDURE Init
************************************************
*)  Description:
*)    Creates the object for one time use (if
*)    a parameter is passed) or for persistent
*)    use if no parameters are passed
*   Scope: PUBLIC
*   Parameters:
*     1.  <cMessage> - the message to display
*            in the status bar
*     2.  <xTimeOut> -
*         a. Numeric - the number of seconds
*             the message should display on the
*             status bar before the old message
*             replaces it
*         b. Logical - expresses the desire for
*             the message to be permanent.
*     3.  <xPARM3> - used to ensure the signature
*            across all message objects is the
*            same, prevents too few parameter
*            error message.
*$  Usage:
*$    1.  For persistent use
*$    goApp.oStatusBar=CREATEOBJ('SAVIStatusBar')
*$
*$    2.  For one time use:
*$    =CREATEOBJ('SAVIStatusBar', <cMessage> )
*$
*%  Example:
*%     =CREATEOBJ('SAVISttusBar','Message Text')
*   Returns:
*     1.  An object reference if created for
*         persistent use
*     2.  .F. if the object should not be
*         created for persistent use.
************************************************
LPARAMETERS tcMessage, tnTimeout, txPARM3

LOCAL llRetVal
llRetVal = .T.

*-----------------------
*-- Save the old message
*-----------------------
this.cOldMessage = SET('MESSAGE', 1)

*-----------------------------------------------
*-- Show the message if information was provided
*-- when the object was created
*-----------------------------------------------
IF TYPE('tcMessage') == "C"
   llRetVal = THIS.Show( tcMessage, tnTimeout )
```

```
ENDIF

RETURN llRetVal

ENDPROC

PROCEDURE Show
************************************************
*)  Description:
*)    Displays a validated message on the
*)    status bar
*   Scope: PUBLIC
*   Parameters:
*     1.  <cMessage> - the message to display
*            in the status bar
*     2.  <xTimeout> - not used, provided as a
*            placeholder to make the signature
*            for showing a message equivalent
*            over the entire message class
*            library without causing errors
*     3.  <xPARM3> - provided to ensure
*            a consistent signature over the
*            entire messaging class library.
*$  Usage:
*$    goApp.oStatusBar.Show( <cMessage> )
*%  Example:
*%    goApp.oStatusBar.Show( 'This is a message' )
*   Returns: .F. by design.  This prevents the
*     object from instantating when Show is
*     called from the Init.
************************************************
LPARAMETERS tcMessage, txTimeout, txPARM3

LOCAL llRetVal
llRetVal = .F.

THIS.ValidateMessage( tcMessage )
THIS.ValidateTimeout( txTimeout )

DO CASE
CASE TYPE( 'THIS.xTimeOut' ) == "L"
   SET MESSAGE TO THIS.cMessage

CASE TYPE( 'THIS.xTimeOut' ) == "N"
   SET MESSAGE TO THIS.cMessage
   INKEY(THIS.xTimeOut)
   THIS.Destroy()

OTHERWISE
   SET MESSAGE TO THIS.cTimeOutErrorMessage
ENDCASE

RETURN llRetVal
```

```
ENDPROC

PROCEDURE ValidateTimeout
************************************************
*)  Description:
*)     This method checks to see if the timeout
*)     value provided by the programmer is either
*)     numeric (indicating the message display
*)     length in seconds) or logical (indicating
*)     the message should be permanent)
*   Scope: PUBLIC
*   Parameters:
*      <xTimeOut> -
*      a.  Numeric - duration of message display
*          in seconds
*      b.  Logical - (any value .T. or .F.)
*          indicates the message is permanent
*$  Usage:
*$    SAVIStatusBar::ValidateTimeout(<xTimeout>)
*%  Example:
*%    THIS.ValidateTimeout( txTimeout )
*   Returns: .T. by default
************************************************
LPARAMETER txTimeOut

DO CASE
CASE TYPE('txTimeOut') == "N"
*--------------------------------------------
*-- A numeric value indicates the length the
*-- message should display on the status bar
*-- before it is replaced with the old message
*--------------------------------------------
   THIS.xTimeOut = txTimeOut

CASE TYPE('txTimeOut') == "L"
*--------------------------------------------
*-- A logical value indicates that the message
*-- should be displayed and never reset the
*-- old message.
*--------------------------------------------
   THIS.xTimeOut = txTimeOut

OTHERWISE
*--------------------------------------------
*-- If the user passed in something whacky, use
*-- the developer specified default
*--------------------------------------------
   THIS.xTimeOut = THIS.xDefaultTimeOut

ENDCASE

ENDPROC
```

```
PROCEDURE ValidateMessage
************************************************
*)  Description:
*)    Ensures that the message is a valid one for
*)    the status bar.  At this level of
*)    abstraction, an empty message is valid since
*)    when the old message being replaced was the
*)    empty string anyway (clear status bar)
*   Scope: PUBLIC
*   Parameters:
*     1. <cMessage> - the message text to be
*          validated
*$  Usage:
*$    SAVIStatusBar::ValidateMessage(<cMessage>)
*%  Example:
*%    THIS.ValidateMessage('Hello message boy')
*   Returns: .T. by default
************************************************
LPARAMETERS tcMessage

LOCAL llRetVal

llRetVal = .T.
*-----------------------------------------------
*-- The status bar message can be empty therefore
*-- only call superclass validation handling if
*-- the message is not character ... i.e. all
*-- types of character messages are OK for a
*-- Status Bar message.  The reason for this is
*-- that one of the functional points of this
*-- message class is to reset the old message
*-- when this message is complete.  It is very
*-- possible that the *old* message was no message
*-- at all but blank.
*-----------------------------------------------

IF TYPE('tcMessage') == 'C'
   THIS.cMessage = tcMessage
ELSE
   SAVIBaseMessage::ValidateMessage( tcMessage )
ENDIF

RETURN llRetVal
ENDPROC

PROCEDURE Destroy
************************************************
*)  Description:
*)    Replaces the message with the old message
*   Scope: PUBLIC
*   Parameters: None
*$  Usage: Automatically executed when the object
```

```
*$    goes out of scope.
*%  Example: N/A
*   Returns: .T. by default
************************************************
this.Show( this.cOldMessage )

ENDPROC


ENDDEFINE
```

The MESSAGEBOX object

The message box object, like the previous two objects, is also designed for either a one shot use or for persistent usage throughout the life of the application. The following listings demonstrate how to utilize the message box object both ways.

Listing 8: Instantiating the MESSAGEBOX object for persistent use

```
goApp.oMessageBox = CREATEOBJ('SAVIMessageBox')

lcRetVal = goApp.oMessageBox.Show( <cMessage>     , ;
                                  [<cButtonIcon>] , ;
                                  [<cTitle>]        )
```

Listing 9: Instantiating the MESSAGEBOX object for one time use

```
lnRetVal = 0
=CREATEOBJ('SAVIMessageBox', <cMessage>     , ;
                            [<cButtonIcon>] , ;
                            [<cTitle>]        )
```

As with all of the other message objects, the first parameter is reserved for the message the developer wants to display. The second parameter is designed to relieve the developer of remembering either the numbers or defined constants for the icon, button combination and default selection. In order to specify the type of icon to display on the message box, provide one of the following strings "Information", "Stop", "Warning" or "Exclamation". To specify the button combination to appear on the message box, type the names of the desired button combinations, in any order. For example "OK,Cancel", "CancelOK" (note the missing comma, the comma is not necessary but it can be included for clarity), "AbortRetryIgnore", "Retry, Ignore, Abort" both work equally well (note that the order is not important either). To request that a specific button to be the default type "First", "Second" or "Third". The third and final parameter is reserved as the place where the developer can define the title to display. As an example, the following code displays an AbortRetryIgnore message with the Ignore button as the default with the Stop icon.

Listing 10: Fully populated MESSAGEBOX object signature

```
GoApp.oMessageBox.Show( 'This is a message for you !!!' , ;
```

```
                    'Abort,Retry,Ignore,Stop,Third', ;
                    'User Communication App, Version 1' )
```

Defining organizational standards for the MESSAGEBOX object

In order to define your company's default configuration, populate the following properties with the desired values, nDefaultButton, nDefaultIcon, nDefaultSelected and cTitle. The defined FoxPro constants are included in the documentation to ease this procedure. The defined constants themselves were not used in order to reduce, completely, the reliance of this class definition on the presence of extraneous files for proper operation. Creating an organizational default message box allows your developers to reduce the signature for displaying the default message box to the following:

Listing 11: MESSAGEBOX signature taking advantage of organizational defaults

```
GoApp.oMessageBox.Show( 'This is a message for you !!!' )
```

In the above example, if Abort, Retry Ignore was defined as the default button set, Stop was defined as the default icon, Ignore was defined as the default button and "User Communication App, Version 1" was defined as the default title, the code in Listing 11 accomplishes the same thing as the code in Listing 10 but with many fewer keystrokes.

Listing 12: MESSAGEBOX object class definition

```
*================================================
*============== SAVI Message Box ================
*================================================
*-- Class:       SAVIMessageBox
*-- ParentClass: SAVIBaseMessage
*-- BaseClass:   Custom
*================================================

DEFINE CLASS SAVIMessageBox AS SAVIBaseMessage

*-------------------------------------------------
*-- nButtonIcon - stores the numeric equivalent
*--               of the requested buttons and
*--               icon combinations.
*-- cTitle      - the caption that appears in the
*--               title of the message box
*--               (defaults to APPNAME_LOC)
*-- cDefaultButtonIcon - allows the developer
*--               to identify the default button
*--               icon combination to display
*--               when none have been provided.
*--               Defaults to "Information, OK".
*-- FoxPro Defined Constants
*-------------------------------------------------
*-- MB_OK                   0
*-- MB_OKCANCEL             1
*-- MB_ABORTRETRYIGNORE     2
```

```
*-- MB_YESNOCANCEL          3
*-- MB_YESNO               4
*-- MB_RETRYCANCEL         5
*-- MB_ICONSTOP            16
*-- MB_ICONQUESTION        32
*-- MB_ICONEXCLAMATION     48
*-- MB_ICONINFORMATION     64
*-- MB_APPLMODAL           0
*-- MB_DEFBUTTON1          0
*-- MB_DEFBUTTON2          256
*-- MB_DEFBUTTON3          512
*-- MB_SYSTEMMODAL         4096
*-------------------------------------------------

PROTECTED nDefaultButton
PROTECTED nDefaultIcon
PROTECTED nDefaultSelected
PROTECTED nButtonIcon
PROTECTED cDefaultTitle
PROTECTED cTitle

nDefaultButton     = 0    && OK
nDefaultIcon       = 64   && Information
nDefaultSelected   = 0    && First
nButtonIcon        = 0
cDefaultTitle      = "Application Name Goes Here"
cTitle             = ""
Name               = "SAVIMessageBox"


PROCEDURE Init
*-------------------------------------------------
*)  Description:
*)    This method either instantiates the message box object
*)    for persistant use (no parameters provided) or
*)    instantiates it for one time user (parameters are
*)    provided)
*   Scope: Public
*   Parameters:
*     1.  <cMessage> - the message to display
*     2.  [<cButtonIcon>] - the button/icon to display
*     3.  [<cTitle>] - the title to display
*$  Usage:
*$    SAVIMessageBox::Init( <cMessage>      , ;
*$                          [<cButtonIcon>] , ;
*$                          [<cTitle>]        )
*%  Example:
*%    =CREATEOBJ("This is the message" , ;
*%               "OK, Exclamation"     , ;
*%               "Display this title"    )
*   Returns:  LOGICAL .T. by default
*-------------------------------------------------
LPARAMETERS tcMessage, tcButtonIcon, tcTitle
```

```
LOCAL llRetVal
*--------------------------------------------
*-- Do not localize lnRetVal since this is the
*-- only way to get the return value out of the
*-- "one shot use" method of showing a meesage
*-- box
*--------------------------------------------
llRetVal = .T.

IF TYPE('tcMessage') = "C"
   lnRetVal = 0
   lnRetVal = THIS.Show( tcMessage    , ;
                         tcButtonIcon , ;
                         tcTitle        )
   llRetVal = .F.
ENDIF

RETURN llRetVal

ENDPROC

PROCEDURE Show
*-------------------------------------------------
*)  Description:
*)    This method validates the provided
*)    parameters for the message box and then
*)    displays the message box.
*   Scope: PUBLIC
*   Parameters:
*     1. <cMessage> - the message to display in
*                     the message box
*     2. [<cButtonIcon>] - the parameter that
*                     allows the developer to
*                     specify the icon, button
*                     combination do display in
*                     the message box.  These
*                     strings can be presented in
*                     any order and are CASE
*                     insensitive
*          a. "OKCancel" OR "CancelOK"
*          b. "OK"
*          c. "AbortRetryIgnore" OR
*             "RetryAbortIgnore", etc.
*          d. "RetryCancel" OR "CancelRetry"
*          e. "YesNo"
*          f. "YesNoCancel" OR "NoYesCancel", etc.
*          g. "Stop"
*          h. "Exclamation"
*          i. "Question"
*          j. "Information"
*          a. - f. identify the button
*          g. - j. identify the icon
```

```
*     3. [<cTitle>] - the title to display in the
*                     message box
*$  Usage:
*$    SAVIMessageBox::Show( <cMessage>    , ;
*$                          <cButtonIcon> , ;
*$                          <cTitle>        )
*%  Example:
*%    loMessageBox.Show( "This is a message" , ;
*%                       "YesNoCancel"       , ;
*%                       "This is a title"     )
*
*   Returns:
*     Numeric - return value from MessageBox
*-------------------------------------------------
LPARAMETERS tcMessage    , ;
            tcButtonIcon , ;
            tcTitle

LOCAL lnRetVal, ;
      llRetVal

lnRetVal = 0
llRetVal = .T.

*-----------------------------------
*-- Validate the three parameters ...
*-----------------------------------
THIS.ValidateMessage( tcMessage )
THIS.ValidateButtonIcon( tcButtonIcon )
THIS.ValidateTitle( tcTitle )


*--------------------------
*-- Display the message box
*--------------------------
IF llRetVal
   lnRetVal = MESSAGEBOX( THIS.cMessage    , ;
                          THIS.nButtonIcon , ;
                          THIS.cTitle        )
ENDIF

RETURN lnRetVal
ENDPROC

PROTECTED PROCEDURE ValidateButtonIcon
*-------------------------------------------------
*)  Description:
*)   I don't know about you but if I have to type
*)   MB_ICONEXCLAMATION one more time I'm going
*)   to slit my wrists.  No more.  STOP THE
*)   MADNESS. This method turns the second
*)   parameter into a much more palatable
*)   endeavor.
```

```
*   Scope: PROTECTED
*   Parameters:
*     <tcButtonIcon> - string identifying what
*        button and icon you want to display on
*        the message box.
*$  Usage:
*$    <Object>.ValidateButtonIcon(<cButtonIcon>)
*%  Example:
*%    THIS.ValidateButtonIcon("OK,Cancel,Second")
*   Returns:  LOGICAL .T. by default
*-------------------------------------------------
LPARAMETERS tcButtonIcon

LOCAL lnButtonIcon, ;
      lcButtonIcon, ;
      llRetVal

lnButtonIcon = 0
lcButtonIcon = ""
llRetVal     = .T.

*-------------------------------------------------
*-- If no Button/Icon information has been
*-- provided, use the default provided by the
*-- programmer.
*-------------------------------------------------
IF TYPE('tcButtonIcon') <> 'C'
   tnButtonIcon = THIS.nDefaultButton + ;
                  THIS.nDefaultIcon   + ;
                  THIS.nDefaultSelected
   RETURN .T.
ENDIF

lcButtonIcon = UPPER( tcButtonIcon )

*--------------------------------
*-- Identify the ICON information
*--------------------------------
DO CASE
CASE "EXCLAMATION" $ lcButtonIcon
   lnButtonIcon = 48
CASE "STOP"        $ lcButtonIcon
   lnButtonIcon = 16
CASE "QUESTION"    $ lcButtonIcon
   lnButtonIcon = 32
CASE "INFORMATION" $ lcButtonIcon
   lnButtonIcon = 64
OTHERWISE
   lnButtonIcon = THIS.nDefaultIcon
ENDCASE

*---------------------------------------
*-- Identify the BUTTON information
```

```
*-- Please note that OKCancel must come
*-- before the simple OK since OK
*-- is contained in both, the OKCancel,
*-- a more limiting test, must come first
*--------------------------------------
DO CASE
CASE "OK"     $ lcButtonIcon AND ;
     "CANCEL" $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 1

CASE "OK" $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 0

CASE "ABORT"  $ lcButtonIcon AND ;
     "RETRY"  $ lcButtonIcon AND ;
     "IGNORE" $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 2

CASE "YES"    $ lcButtonIcon AND ;
     "NO"     $ lcButtonIcon AND ;
     "CANCEL" $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 3

CASE "YES"    $ lcButtonIcon AND ;
     "NO"     $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 4

CASE "RETRY"  $ lcButtonIcon AND ;
     "CANCEL" $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 5

OTHERWISE
   lnButtonIcon = lnButtonIcon + ;
                  THIS.nDefaultButton

ENDCASE

*--------------------------------------
*-- Identify selected button information
*--------------------------------------
DO CASE
CASE "FIRST"  $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 0

CASE "SECOND" $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 256

CASE "THIRD"  $ lcButtonIcon
   lnButtonIcon = lnButtonIcon + 512

OTHERWISE
   lnButtonIcon = lnButtonIcon + ;
                  THIS.nDefaultSelected
```

```
ENDCASE

THIS.nButtonIcon = lnButtonIcon

RETURN llRetVal

ENDPROC


PROTECTED PROCEDURE ValidateTitle
*-------------------------------------------------
*)  Description:
*)    This method ensures that a provided title
*)    overrides the default title, that the
*)    default title is the standard APPNAME_LOC
*)    for Codebook and if an error occurs (non
*)    character title), the default is used.
*   Scope: PROTECTED
*   Parameters:
*     1.  [<cTitle>] - the title to appear on the
*             message box
*         a.  Not passed, default is APPNAME_LOC
*         b.  Error (non character) the default
*             is used
*         c.  Provided, takes precedence over
*             the default
*$  Usage:
*$    SAVIMessageBox::ValidateTitle( <cTitle> )
*%  Example:
*%    THIS.ValidateTitle( tcMessage )
*   Returns:  LOGICAL .T. by default
*-------------------------------------------------
LPARAMETERS tcTitle
LOCAL llRetVal
llRetVal = .T.

DO CASE
CASE TYPE('tcTitle') <> "C"
   THIS.cTitle = THIS.cDefaultTitle

CASE EMPTY(tcTitle)
   THIS.cTitle = THIS.cDefaultTitle

OTHERWISE
   THIS.cTitle = tcTitle

ENDCASE

RETURN llRetVal
ENDPROC


ENDDEFINE
```

Bringing it all together

Notice two very important points about the message objects described in this article. The first is that each of the messaging objects provide a means to define organizational defaults. The benefits of this are as follows: 1) the signature for displaying these messages, in many situations, can be reduced to a reference to the message object's Show method with the only information being provided is the string that defines the message to be displayed. If the other parameters are omitted, the organization defaults are used to complete the command 2) if the organizational defaults change over a period of time, the effort required to accomplish this task involves the modification of only a few properties.

The second and most important point to notice about these message object class definitions is that they are designed to work interchangeably. Each of these objects, the wait window, status bar and message box Show methods are designed to accept three parameters, each are also designed to validate those parameters for accuracy. If the parameter is not of the correct type or does not contain the correct value, the organizational default is used instead of the provided value. This is what allows each of these message objects to be used interchangeably. A signature that works with the wait window object will also work with the message box object with no errors generated. This requirement is imperative if these objects are to be used in an IS-IMPLEMENTED-USING relationship with interface objects.

The IS-IMPLEMENTED-USING relationship

There are three primary relationships between objects in an object oriented application, HAS-A, IS-A and IS-IMPLEMENTED-USING. HAS-A relationships are used to create abstractions for things like a car. A car HAS-An engine. IS-A relationships are used to create abstractions that require specialization. An artist IS-A person and can be modeled using inheritance. IS-IMPLEMENTED-USING relationships allows developers to create abstractions where the object in question requires the assistance of another object but where the type of object used can vary from implementation to implementation.

An example of this is the contained controls that I use to build applications. Each contained control IS-A container and HAS-A label as well as a VFP Control. Each contained control IS-IMPLEMENTED-USING a messaging object that is responsible for managing that control's communication with the user. The control's parent container has a member property named "oMessage". The object whose reference is contained in this property could be a wait window, message box or status bar message object. The only way this implementation can work is if the Show methods between these three objects is completely interchangeable. This flexibility gives me the freedom to associate any type of messaging object I desire with any interface object I desire. The construction of these controls are the topic of another article.

Conclusion

This class library illustrates the power available to people that choose to develop in an object oriented environment. Maintenance is much easier since code that performs a specific task is located in only one place (if designed properly). If code is determined to be buggy, once the bug is fixed, it is fixed throughout the entire application. If new default behavior is desired, modifications can be made in one place and are echoed everywhere those default values are used.

Also, if the rules of polymorphism are strictly followed, a method with the same name, on a different object, can produce significantly different results and can be made available for IS-IMPLEMENTED-USING relationships.

One final, very important point to consider is that in an object oriented environment, it is imperative to perform the proper analysis and design before any code whatsoever is written. Don't be blindfolded by what, at first glance, appears to be an insignificant situation. Many developers, myself included, have replicated WAIT WINDOW, MESSAGEBOX and SET MESSAGE TO commands throughout the application hundreds of times. After all, these commands are *so* easy to type and hardly deserve the effort of analysis and design. However, trivializing this situation creates a potential maintenance problem if something global about a MESSAGEBOX or WAIT WINDOW must be changed at a client's request. The trick is to focus on things that you as a developer continuously repeat time and time again over many applications and apply object oriented analysis and design principles to those problems. You will be surprised at the results.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: ***www.savvysolutions.com****, CompuServe 76132,2575*

Fixing QSTART.APP

Several folks have frequently experienced a problem with QSTART.APP becoming "corrupted". The fix for this is extremely simple and requires about 10 minutes of your time.

Charles T. Blankenship

Introduction

On several occasions it became necessary to trace through the QSTART.APP. Unfortunately, after placing the debug code in the QSTARTMA.PRG I received the following error message when running the application "Source Out Of Date". The solution to this problem was simple. Recreate the project from scratch using its component files. This article defines the files that comprise QSTART.APP and provides the instructions for rebuilding it.

Editor's Note: It is also to receive the "Source Out Of Date" or "Source Not Available" messages if you place debug code in the QSTARTMA.PRG, recompile the application, copy the QSTART.APP from the QSTART subdirectory to the UTILS subdirectory and finally, try to run your new, modified application. Since the source files for QSTART cannot be found from the UTILS subdirectory, you get an error message that can make you think your .APP is corrupted when it is not. If you do place debug code in your QSTART application, make sure you run QSTART.APP from the \CDBK30\COMMON30\UTILS\QSTART subdirectory before you go through the process of "fixing" QSTART.APP as explained in this article.

Out with the Old

To begin the process of fixing QSTART.APP, completely erase all traces of the QSTART application by removing the following files from the disk.:

```
CDBK30\COMMON30\UTILS\QSTART\QSTART.APP
CDBK30\COMMON30\UTILS\QSTART\QSTART.PJX
CDBK30\COMMON30\UTILS\QSTART\QSTART.PJT
```

In with the New

To recreate QSTART.APP you must recreate the project files, identify its component programs and class libraries, add them back to the project and recompile the newly built application.

Recreate the Project Files

To recreate the project files, CD to the following directory CDBK30\COMMON30\UTILS\QSTART and issue the command MODI PROJ QSTART. This recreates the project file (a skeleton), that was previously deleted. The next step is to identify each component of the QSTART application.

QSTART.APP's Component Files

The following is a list of files that comprise the QSTART.APP. They are classified based upon their location in the project file.

To make this as easy as possible, go to the ALL tab of the project manager and add the following files to their specified locations from there.

Data

Free Tables

CDBK30\COMMON30\UTILS\QSTART\INCLUDE\APPINCL.DBF **(Excluded)**

CDBK30\COMMON30\INCLUDE\STRINGS.DBF **(Excluded)**

Class Libraries

CDBK30\COMMON30\LIBS\CCONTRLS.VCX
CDBK30\COMMON30\LIBS\CCTLLIB.VCX
CDBK30\COMMON30\LIBS\CCUSTCTL.VCX
CDBK30\COMMON30\LIBS\CFORMS.VCX
CDBK30\COMMON30\LIBS\CUSERPREF.VCX
CDBK30\COMMON30\LIBS\CUTILS.VCX
CDBK30\COMMON30\UTILS\QSTART\LIBS\FORMS.VCX

Code

Programs

CDBK30\COMMON30\UTILS\QSTART\PROGS\QSTARTMA.PRG **(Main)**
CDBK30\COMMON30\PROGS\SETUP.PRG
CDBK30\COMMON30\PROGS\UTILITY.PRG

API Libraries

CDBK30\COMMON30\LIBS\FOXTOOLS.FLL **(Excluded)**

Other

Text Files

CDBK30\COMMON30\UTILS\QSTART\INCLUDE\APPINCL.H **(Excluded)**

CDBK30\COMMON30\INCLUDE\FRAMEWRK.H **(Excluded)**

CDBK30\COMMON30\INCLUDE\FRAMINCL.H **(Excluded)**

CDBK30\COMMON30\INCLUDE\STRINGS.H **(Excluded)**

Other

Other Files

CDBK30\COMMON30\GRAPHICS\FLASH.ICO

Notice that seven of the above files are excluded. To exclude these files from the project, highlight the file, once it is a part of the project, go to FoxPro's main menu and select the PROJECT menu pad. Next, select the EXCLUDE menu bar from the resulting popup. When the file is properly

excluded, a circle with a diagonal line through it displays to the left of the file name in the project manager.

A special note is required when including the Other|Text Files and Other|Other Files in the project. In order to view the .H and .ICO files in the Open File Dialog that displays after pressing the Add button on the project manager, you must chose the All Files selection from the Files Of Type: ComboBox.

The final step is to mark the QSTARTMA.PRG as the main program. To accomplish this, select QSTART.PRG from the Code, Programs section of the Project Manager Tree and select Project from FoxPro's main menu. Select Set Main from the resulting project. A check mark should appear next to the Set Main menu selection (when the main program is selected in the Project Manager) and a black dot should appear to the left of the main program file in the project manager.

Once you rebuild your application, copy the new QSTART.APP to the `CDBK30\COMMON30\UTILS\` subdirectory, in preparation for common use. You should be ready to create new Codebook applications with no further problems. Remember, if you are tracing through this application using debug, ensure you launch QSTART from its home directory, not the UTILS subdirectory.

Creating a New Application using QSTART.APP

The usage of QSTART.APP has been well documented by now. The primary purpose of this article is to document a trick that can help make your application transportable over multiple drives without having to tweak hard coded drive letters.

Charles T. Blankenship

Introduction

Flash Creative Management provided QSTART.APP to help developers create new Codebook applications. The usage of this application is no longer a mystery. There is a problem when developers use QSTART.APP in the standard way, however. Hard coded drive letters in the new application lock the application onto one drive.

Each project developed for a client should be completely transferable from drive to drive. The reasons for this are as follows. First, the client may have the need to change the drive mapping on their server. If the drive letter changes from F: to G: your application should remain perfectly functional. Second, developers have been known to either 1) drop a customer or 2), God forbid, have the customer drop them. Whatever the case, the maintenance of the system that you developed could fall under the responsibility of another consultant. It would be nice if that consultant could transfer that application to their machine (another potential drive letter) and continue supporting the application from their own site without undergoing the additional hassle of changing hard coded drive mappings.

The Old Method

The following is a screen shot from a typical session of QSTART.APP.

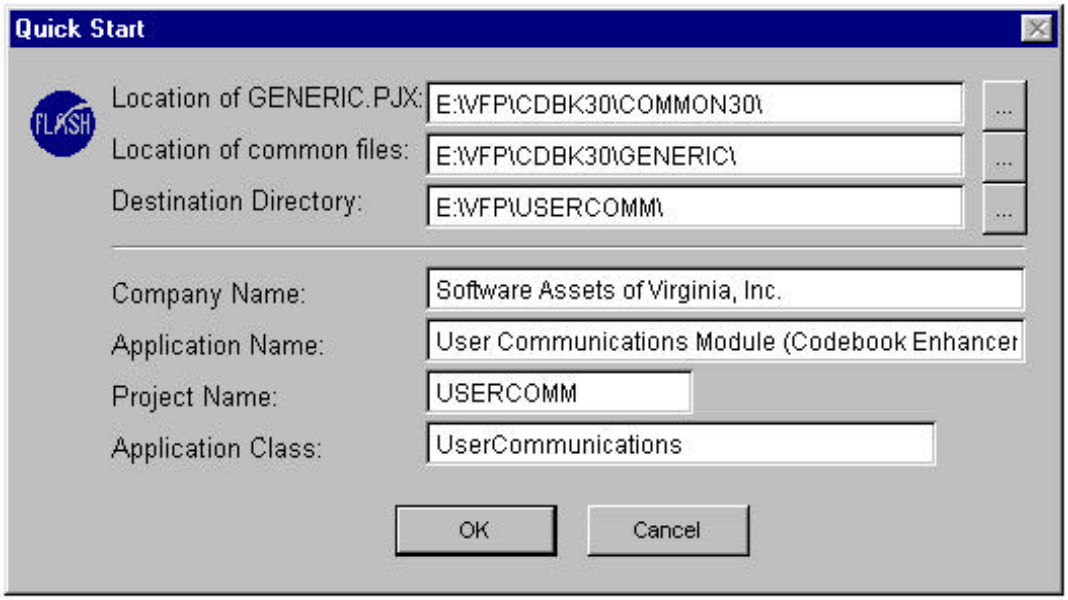
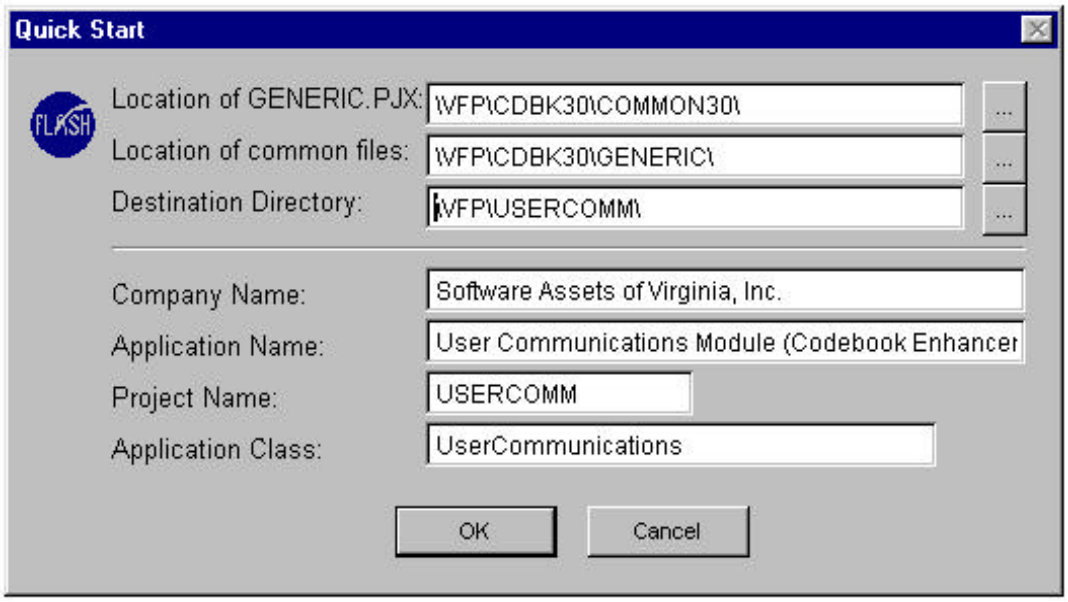


Figure 1: The Old Method for using QSTART.APP

Examine Figure 1. Do you notice the one glaring problem staring you in the face? Hard coded drive letters. The solution to this problem was as simple as following the **GIGO** rule of data processing, **G**arbage **I**n **G**arbage **O**ut. The problem was not with the QSTART.APP itself. The problem resides in the fact that programmers enter the offending drive letter from the very beginning. The question became "Would removing the drive letters from this dialog allow these applications to be transportable from drive to drive "? The answer was ***YES***.

The New Method

The following figure illustrates the new method for using QSTART.APP. To make your applications transportable over multiple drives, follow this one simple rule: Do not put the drive letters in QSTART's main dialog text boxes. You can still use the ellipsis box to the right to pick the location of the GENERIC.PJX and the Common Files. Simply perform one more step before pressing the OK button. Backspace to the beginning of each text box and remove the drive letter and colon. Do the same for the Destination Directory as well. The secret to making your applications transportable over multiple drives is in the use of relative paths, not hard coded ones.



Conclusion

The result of this simple change in operating rules is that a client's applications can now be located anywhere and moved anywhere. It can be on their server, moved to another location on their server, reside on your laptop or on the server in your main office. This gives you the freedom to support your clients at all times, all you need is a laptop, pager, a modem and a connection to the internet to transfer the fixes you make.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: www.savvysolutions.com, CompuServe 76132,2575

cReportForm Enhancements

The reporting capability of Codebook satisfies only the most basic of client needs. These two enhancements enable your users to specify simple filtering and ordering requests as well as control where the output file is printed when they select the "File" option.

Charles T. Blankenship

The reporting capabilities provided with Codebook satisfy only the most basic reporting needs of clients. Two shortcomings of this reporting engine are 1) reports, when printed to a file, are dumped in the application's root directory and 2) the users have little or no control over the report they are printing. The enhancements documented in this article describe how to implement file redirection. Another article will address how to provide them with simple filtering and ordering requests.

File Redirection

Currently, Codebook's default behavior is to place the file in the application's root directory when a user chooses **Send to File** as the report's **Output Destination** on the Codebook print dialog form (cReportForm in cCustFrm.VCX). This practice becomes rather messy over time and ends up annoying LAN managers. The solution is to provide a way to redirect these files to an output directory for each user of the application. The benefits of this are several:

1. Increased security. For example, section managers (company employees that manage a section) may decide to print payroll information to file; Human Resource managers may decide to print sensitive personnel information to file. If this information is placed in the root directory of the application, everyone who has rights to the application subdirectory (read all users of the application) can view the sensitive information in the output text file. If separate subdirectories are set up as output destinations for each user, LAN administrators can govern the rights of these directories so that sensitive information remains the sole property of the individuals who did the printing.

2. Housekeeping. The second benefit that results from specifically defining output subdirectories is the ability to clean up old files. If all of the report text files are dumped into the root directory of the application it becomes difficult to determine which ones are currently being used and which ones are old files whose effective lives have passed. Placing report files in specific output directories that are assigned to specific individuals helps a LAN manager (not to mention the user's themselves) to better manage LAN resources.

The following is an excerpt from the code that processes the users print request. It is present in the Click() method of the Run button on the print dialog form of the framework. Notice that the original method for determining the name of the output file is to extract the DOS file name from the REPOLIST table (REPOLIST.cDosName), trim it up and append ".TXT " to the end. This generated name is then used in the REPORT FORM command after the TO FILE clause to create the file in the default directory of the application.

Listing 1: Exceprt from cReportForm.cmdRun.Click() - Original

```
CASE thisform.opgOutput.optFile.Value = 1
    *-----------------------------------------------------------
    *-- Create a DOS text file from the name of the report file
    *-----------------------------------------------------------
    lcTextFile  = ALLTRIM(Repolist.cDosName)+".TXT"

    *------------------------------
    *-- Print the report to the file
    *------------------------------
    REPORT FORM (lcSeleReport) TO FILE (lcTextFile) ASCII

    *-----------------------------------------------
    *-- Inform the user what the name of the file is.
    *-----------------------------------------------
    =MESSAGEBOX(FILESAVEDAS_LOC + FULLPATH(lcTextFile), ;
                MB_ICONINFORMATION)
ENDCASE
```

Framework Enhancement Tip

A good process to follow when enhancing a framework in this manner is to describe in detail the new functionality that is needed. Don't worry too much about HOW the new code accomplishes its tasks. That comes a little later. Only worry about WHAT has to be accomplished and HOW the new method can be successfully integrated with existing code. Also, don't forget that backwards compatibility is always a critical issue when designing enhancements to a framework like Codebook.

Designing the GetReportTextFileName() method

The solution to this problem is to create a custom method whose responsibility is to get the desired name of the report file. In this situation, the GetReportTextFileName() method is responsible for the following things: it must 1) generate the fully qualified report text file name, 2) return that name via a reference parameter, 3) let the calling program know whether or not it succeeded in accomplishing its task. If file name generation was successful, execute the REPORT FORM command with the generated file name. If file name generation was not successful, notify the user that their report did not print to file.

Requirements Specification for GetReportTextFileName()

Notice that determining how the new method must integrate with current programming resulted in the generation of a rudimentary requirements specification. The requirements for the method up to this point are as follows:

1. It requires that a parameter be passed by reference. This parameter is the way to return the name of the file to which the report will print without having to use the return value to accomplish the task. The return value is reserved for informing the calling program whether or not the task was successfully completed..
2. It must generate the name of the file and fully qualify it with a drive and a path.
3. The return value must be logical and indicate the success or failure of the method to generate a file name.

4. If the method cannot generate a file name, the user must be notified of the failure. Notice that the responsibility for this messaging resides within the method itself and not in the calling program.

Additional Requirements:

1. Backwards compatibility. Some users of the framework may like the way it generates output file names. Therefore, there must be a way to turn this new functionality on and off as desired. This is best accomplished by adding a new property to the object and defaulting its value so that it behaves as the original Codebook
2. Usability. The developer must have a dedicated place to hook in their own code for generating the name of the user defined output file.

Implementing the GetReportTextFileName() method

Listing 2 illustrates how the new method is hooked into the existing framework, Listing 3 illustrates this newly created method. Notice that this new method is nicely fault tolerant. If the user defined their own method for generating a file name then that processing is used. If it fails, the Codebook default method of generating an output file name is used as a catchall. (Also note that the default method for generating the output file name is skipped if the user's method succeeds).

Listing 2: Excerpt from cReportForm.cmdRun.Click() - Modified

```
CASE thisform.opgOutput.optFile.Value = 1
    *------------------------------
    *-- MODIFIED - 10/25/1997 - CTB:
    *------------------------------
    lcTextFile = ""
    llGotReportTextFileName = THISFORM.GetReportTextFileName( @lcTextFile )
    IF llGotReportTextFileName
       *------------------------------
       *-- Print the report to the file
       *------------------------------
       REPORT FORM (lcSeleReport) TO FILE (lcTextFile) ASCII
       *-----------------------------------------------
       *-- Inform the user what the name of the file is.
       *-----------------------------------------------
       =MESSAGEBOX(FILESAVEDAS_LOC + FULLPATH(lcTextFile), ;
                   MB_ICONINFORMATION)
    ENDIF
```

Listing 3: The GetReportTextFileName() implementation

```
*---------------------- Location Section -----------------------
*   Library: cCustFrm
*   Class:   cReportForm
*   Method:  GetReportTextFileName()
*----------------------- Usage Section -------------------------
*)  Description:
*)    This methods primary function in life is to derive the
*)    name of the text file (drive + path + filename) that
*)    is created when the user chooses to print the report
*)    to a file.  The developer has two choices
*)    1.  Use Codebook's default method of placing the
*)        the output file in root of the application and
*)        naming it after the report
*)        .lUseCodebookStandardReportOutputFile = .T.
*)    2.  Use their own method of generating the fully qualified
*)        report output file.
*)        .lUseCodebookStandardReportOutputFile = .F.
*   Scope:      PUBLIC
*   Parameters: NONE
*$  Usage:
*$    cReportForm::GetReportTextFileName( @<tcTextFile> )
*%  Example:
*%    lcTextFileName = ""
*%    THISFORM.GetReportTextFileName( @lcTextFileName )
*   Returns:    LOGICAL
*     .T. if the output file name was generated
*     .F. if it is not.
*--------------------- Maintenance Section ---------------------
*@  Inputs:
*@     1.  cReportForm.lUseCodebookStandardReportOutputFile
*@         Allows the developer to either use Codebook's default
*@         method of generating the name of the output text
*@         file (.T.) OR use a user defined method of generating
*@         the name of the output text file (.F.)
*   Outputs: None
*   Pre-condition Invariants:
*      1.  The developer must properly set the value of the
*          lUseCodebookStandardReportOutputFile property
*          so that processing is switched in the proper
*          direction
*   Post-condition Invariants:
*      1.  A fully qualified ( drive, path and filename ) is returned
*          to the calling program
*?  Notes:
*?     1.  To make this method more fault tolerant note that
*?         the Codebook default method of generating an output
*?         file name is used as a catchall.  If the method
*?         the developer specified does not produce a valid result
*?         then attempt to generate the name of the file using
*?         the Codebook default method.  (The default method is
*?         skipped if the user's method succeeds.
```

```
*   Collaborating Methods:
*      1.  cReportForm::UserDefinedReportOutputFile()
*-- Process:
*--    1.  IF the user wants to use their own method for
*--        generating a file name.
*--    2.     Execute the developer defined code for doing so.
*--        ENDIF
*--    3.  IF the text file name has still not been successfully
*--        generated.
*--    4.      Use the Codebook default method for generating
*--            the name of the output file.
*--        ENDIF
*--    5.  Message the user if not file name could be generated
*   Change Log:
*       CREATED Saturday, 10/25/97 13:49 - CTB:
************************************************************
LPARAMETER tcTextFile

LOCAL llGotTextFileName, ;
      lcOutputDirectory, ;
      llGotDirectory

llGotTextFileName = .F.
tcTextFile = ""

IF .NOT. THIS.lUseCodebookStandardReportOutputFile
   llGotTextFileName = THIS.GetUserDefinedReportOutputFile( @tcTextFile )
ENDIF

IF .NOT. llGotTextFileName
   tcTextFile  = ALLTRIM( Repolist.cDosName ) + ".TXT"
ENDIF

llGotTextFileName = .NOT. EMPTY( tcTextFile )

IF .NOT. llGotTextFileName
   =MESSAGEBOX( "A name for the output file could not be generated.  "  + ;
                "Your report did not print to a file.  Please contact " + ;
                "the application administrator for assistance.", MB_OK  , ;
                 APPNAME_LOC)
ENDIF

RETURN llGotTextFileName
```

Customizing Report File Redirection

In order to implement your own code for generating the name of an output file for a report, you must first initialize the lUseCodebookStandardReportOutputFile property to .F. This is what causes the GetUserDefinedReportOutputFile() method to execute. Next, you must provide the code that generates a fully qualified, legal DOS file name and place it in the GetUserDefinedReportOutputFile() method. Your code must accept a parameter by reference.

Then, it must generate the fully qualified name of the report output file and place it into the parameter. Finally, it must test to ensure that the file name is valid and return the result. If valid, return a .T., if not valid, return a .F.

An Example Of A Custom Implementation

Listing 4 illustrates an example of a user defined method that accomplishes this task. This example uses the newly developed SAVI Security module that will be available for use in January and is implemented in SAVI's version of Codebook (which is available for free to any developer who can prove ownership of Codebook).

Listing 4: Example Implementation

```
*---------------------- Location Section -----------------------
*   Library: cCustFrm.VCX
*   Class:   cReportForm
*   Method:  GetUserDefinedReportOutputFile()
*----------------------- Usage Section -------------------------
*)  Description:
*)    This method provides developers with the place to
*)    put code that they need to generate the fully
*)    qualified report output file name.
*   Scope:      PUBLIC
*   Parameters:
*       1.  tcReportOutputFile
*$  Usage:
*$    cReportForm::GetUserDefinedReportOutputFile( @tcReportOutputFile )
*%  Example:
*%    THISFORM.GetUserDefinedReportOutputFile( @lcReportOutputFile)
*   Returns:    LOGICAL
*      .T. if a directory name was generated
*      .F. if not.
*--------------------- Maintenance Section ---------------------
*@  Inputs:
*     1.  REPOLIST.cDosName -
*         The DOS name of the report that is stored in the
*         REPOLIST.DBF
*   Outputs:  None
*   Pre-condition Invariants:
*     1.  Developer must specify the DOS name of the report
*         in the REPOLIST.DBF.
*   Post-condition Invariants:
*     1.  A fully qualified name for the output file
*         is generated and provided to the calling program
*?  Notes:
*?    1.  This method exists for one reason.  It provides
*?        another developer a place to hook in their own
*?        processing for generating the name of the
*?        output report file.  If you don't like the method
*?        that uses the security system, override this
*?        code in a subclass that defines how you want
*?        the output file name generated.
*?    2.
*   Collaborating Methods:
*       cSecure.VCX
*       SAVIApplicationSecurity::GetUserOutputDirectory()
*-- Process:
*--    1.  IF the security module exists
*--    2.     Get the user's output directory from there
*--    3.     IF the users output directory was retrieved
*--    4.        Create the output file name by adding the
*--              the DOS filename to the user's output directory
*--              and appending a .TXT file to it
*--           ENDIF
```

```
*--        ENDIF
*   Change Log:
*       CREATED Saturday, 10/25/97 13:27:00 - CTB:
************************************************************
LPARAMETER tcReportOutputFile

LOCAL lcOutputDirectory , ;
      llGotDirectory    , ;
      llRetVal

tcReportOutputFile = ""
lcOutputDirectory  = ""
llGotDirectory     = .T.

IF TYPE( 'goApp.oSecurity' ) == "O" AND .NOT. ISNULL( goApp.oSecurity )

   lcOutputDirectory = ""
   llGotDirectory = goApp.oSecurity.GetUserOutputDirectory( @lcOutputDirectory )

   IF llGotDirectory
      tcReportOutputFile = lcOutputDirectory          + ;
                           ALLTRIM(Repolist.cDosName) + ;
                          ".TXT"
   ENDIF
ENDIF

llRetVal = .NOT. EMPTY( tcReportOutputFile )

RETURN llRetVal
```

Providing Users with Pre-Printing Ordering and Filtering Processing

Basically, you have two choices here. The first is to program your own functionality, the second is to buy FoxFire! and integrate it with your Codebook application. My recommendation is to buy *FoxFire!*. At the request of a client, I created the code required to provide ordering and filtering capability to the Codebook reporting engine. However, after showing them the effort required to hand program the accompanying ordering and filtering screens for reports, it was extremely easy to sell him *FoxFire!*

The justifications for buying *FoxFire!* were many but a few of the most important ones follow: 1) *FoxFire!* places report creation capabilities in the hands of the users 2) In those few situations where the user was not capable of creating a more complicated report, *FoxFire!* reduced my development time from an average of one hour to ten minutes. With these two reasons alone, *FoxFire!* paid for itself in less than a week. It saved my client money by enabling *his* people to create their reports instead of having to pay me to do it, and it saved my client money, when *I* had to create the more complicated reports, by reducing development time on average from 60 minutes to 10 minutes per report. You owe it to your clients to merge *FoxFire!* with your Codebook applications.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: www.savvysolutions.com, CompuServe 76132,2575

A New LockScreen()

Two fundamental principles in programming are 1) keep it simple and 2) do not change the environment without returning it to its original state. A rewrite of the LockScreen() function ensures both of these principles are obeyed.

Charles T. Blankenship

The Problem

The current implementation of LockScreen() looks like the following:

Listing 1: Original Implementation of LockScreen()

```
************************************************************
*  FUNCTION LockScreen()
************************************************************
*  Author............: Paul Bienick
*  Project...........: Codebook 3.0
*  Created...........: 07/24/95  14:16:16
*  Copyright.........: (c) Flash Creative Management, Inc., 1995
*) Description.......: 15. Function to get around refresh anomolies
*)                   : in various situations where setting
*)                   : LockScreen to .T. while doing the refresh
*)                   : seems to help.
*  Calling Samples...: =LockScreen(.T.)
*                     : =LockScreen(.F.)
*  Parameter List....:
*  Major change list.:

FUNCTION LockScreen(tlValue)
  IF FormIsObject()
    _screen.ActiveForm.LockScreen = tlValue
  ELSE
    _screen.LockScreen = tlValue
  ENDIF
ENDFUNC
```

LockScreen() works on one of two objects, the form (if it exists) or the VFP _SCREEN if no form exists. One of these objects will have its .LockScreen property set to the value that you specify when you called the function, .T. if you used the =LockScreen(.T.) signature and .F. if you used the =LockScreen(.F.) signature.

Two problems arise with this simple function, the first is exemplified in Listing Two. Code like this makes the fatal assumption that the initial state of the .LockScreen property is .F. This may or may not be the case. If the .LockScreen property is .T. when this method is fired, it is a guarantee that it will not be when this method exits. The result is a violation of a fundamental programming principle ... the environment was modified and not reset to its original condition.

To solve this problem, the programmer is forced to write additional code to make sure the original value of .LockScreen is restored when the method processing is complete. An example of this type of is in Listing Three.

Listing Two: Example of a business object method that must modify the .LockScreen property of its form.

```
=LockScreen( .T. )

DO SOME PROCESSING

=LockScreen( .F. )
```

Listing Three: Code that properly resets the .LockScreen property value

```
LOCAL llOldLockScreen
llOldLockScreen = THISFORM.LockScreen

=LockScreen( .T. )

DO SOME PROCESSING

=LockScreen( llOldLockScreen )
```

The code in Listing Three is much better than in Listing Two since it makes sure that the environment is reset to its original condition ... whatever that condition is. However, this can still be improved.

The Goal

A better way to accomplish this task is to encapsulate saving and resetting the state of the interface object's .LockScreen property within the LockScreen function itself. Therefore, the goal of this rewrite is to force LockScreen() to take care of itself. This reduces the number of lines of code the programmer has to write as well as the complexity (slightly) of the application's method code. For an example of this examine the code in Listing Four.

Listing Four: The new calling signature of LockScreen()

```
LOCAL llOldLockScreen

=LockScreen( .T., @llOldLockScreen )

 ... DO SOME PROCESSING ...

=LockScreen( llOldLockScreen )
```

The Implementation

Enhancing the LockScreen() function was a simple matter of accepting another parameter and storing the initial state of the object's .LockScreen property to that parameter BEFORE changing the .LockScreen property to the specified value. Notice that the only requirement is that the variable used to store the old value of the .LockScreen property must be passed to this function by reference, this is accomplished by prefixing the variable with an "at" sign, "@"

```
************************************************************
*  FUNCTION LockScreen()
************************************************************
*  Author............: Paul Bienick
*  Project...........: Codebook 3.0
*  Created...........: 07/24/95  14:16:16
*  Copyright.........: (c) Flash Creative Management, Inc., 1995
*  Copyright.........: (c) Software Assets of Virginia, Inc. 1997
*) Description.......: 15. Function to get around refresh anomolies
*)                   : in various situations where setting
*)                   : LockScreen to .T. while doing the refresh
*)                   : seems to help.
*  Calling Samples...: =LockScreen( .T., @llOldLockScreen ) -
*                     :  Sets the .LockScreen property to the specified value
*                     :  and stores the old value so it can be reset later.
*                     :
*                     : =LockScreen( llOldLockScreen )
*                     :  Resets the .LockScreen property to its original value
*  Parameter List....:
*  Major change list.: MODIFIED Friday, 11/28/97 14:18:43 - CTB: (#71)
************************************************************

FUNCTION LockScreen( tlValue, tlOldLockScreen )

   IF FormIsObject()
      tlOldLockScreen   = _screen.ActiveForm.LockScreen
      _screen.ActiveForm.LockScreen = tlValue
   ELSE
      tlOldLockScreen   = _screen.LockScreen
     _screen.LockScreen = tlValue
   ENDIF

ENDFUNC
```

Conclusion

This small change simplifies the code and ensures that the environment is always restored to its original condition with as little as effort as possible expended by the programmer.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: www.savvysolutions.com, CompuServe 76132,2575

An Even Better LockScreen

Ed Leafe

Introduction

The LockScreen function which comes with Codebook does one thing fairly well, namely locking and unlocking the current form's screen refreshing in order to streamline updates. I've created a class which handles this process in a much smoother fashion.

Locking the form greatly speeds up code which refreshes the form, and should be used anytime more than one control will be updated in order to avoid the jerky appearance resulting from a series of separate updates. As great as using LockScreen is, probably the worst thing you can do is to forget to unlock it afterwards. While your users might like the cool effects you can get by dragging around a locked window (try it!), they will probably tire of it quickly and want to get back to work.

The benefit of using an object rather than a function call is that the memvar which holds the reference to the object will go out of scope when the method which creates it ends, and once the last reference to an object is released, the object itself is released. We can use that to our advantage by putting the 'unlock' code in the Destroy() event of the class, so that we can be sure that no matter how the method is exited, the code in the Destroy() will fire. I set the Destroy() code to automatically reset the LockScreen property to the state it was when the object was created. So instead of remembering to make (at least) two function calls, you simply create this object and forget about it. This is especially useful in methods where you may have multiple return points; instead of having to make sure you preface each RETURN with a call to unlock the screen, you simply RETURN, knowing that the local memvar holding the object reference will go out of scope, destroying the object and resetting things back to the way they were.

Using the class is pretty straightforward. All you need to do is add the following two lines to any method at the point where you would like to lock the screen:

```
LOCAL loLock
loLock = CREATEOBJECT("eLockScreen")
```

If you want the screen to unlock when the method is complete, you don't need to do anything - the **loLock** memvar will go out of scope. If, for some odd reason, you want the screen unlock earlier, simply release the memvar explicitly. That's it!

How it works

The Init() of the object allows you to pass an object reference to it. If none is passed, it defaults to using _Screen.ActiveForm. If the form exists, it stores a reference to it in its **oForm** property (so that it knows which form to unlock later on), stores its current value of **LockScreen** to its **lLockScreen** property, and then sets the form's LockScreen to .T.. If the form does not exist (say, you instantiate the object from the command window with no open forms), nothing happens.

Here is the Init() code:

```
LPARAMETERS toForm

IF TYPE("toForm.Name") == "C"
	This.oForm = toForm
ELSE
	* No form passed; default to the Active Form
	IF TYPE("_screen.ActiveForm.Name") == "C"
		This.oForm = _screen.ActiveForm
	ENDIF
ENDIF

IF TYPE("This.oForm.Name") == "C"
	This.lLockScreen = This.oForm.LockScreen
	This.oForm.LockScreen = .T.
ENDIF
```

When the object is released, either through an explicit release or by the memvar which references it going out of scope, it first checks if it has a valid form reference in its oForm property. If so, it sets that form's LockScreen to the value stored in lLockScreen during the Init().

Here is the Destroy() code:

```
IF TYPE("This.oForm.Name") == "C"
	This.oForm.LockScreen = This.lLockScreen
	This.oForm = .NULL.
ENDIF
```

Why do I store the original state instead of simply setting LockScreen = .F.? Well, imagine you have a subclass of eBizobj in which you want to lock the screen, do some custom stuff, call the default behavior, do some more custom stuff, and then unlock the screen. Your code would look like this (taking, for example, the Requery() method):

```
PROCEDURE Requery()
	LOCAL lnRetVal, loLock
	loLock = CREATEOBJECT("eLockScreen")

	(do some cool stuff here)

	lnRetVal = DODEFAULT()

	(Do some more cool stuff for which you want the screen locked)

	RETURN lnRetVal    && loLock is released, unlocking the screen
ENDPROC
```

The problem comes up in the default code, which locks the screen using eLockScreen, too. If the default code simply unlocked the screen without being concerned as to the state of LockScreen when it was called, the second part of your custom code would be run with an unlocked screen,

which isn't what you want. But by noting that the screen was locked when the default eLockScreen object was created, it knows not to unlock it when it gets released, and you end up with a smoothly updated form.

About the Class

If you examine the class, you may notice an odd thing about it: its BaseClass is **Separator,** rather than **Custom**. The reason is that I've been doing some testing, and the Separator class is the lightest weight class in VFP. This means that it instantiates fastest and has the smallest memory footprint of any class you can create. It's so small, in fact, that it doesn't even have native Init() and Destroy() events! Luckily, if you add methods with these names, they will be fired when you would expect them to be. I've been basing all my custom classes on Separator for a while now, and I've been pleased with the results.

You may also notice how I check for the existence of an object. The typical way to see if an object reference (such as 'This.oFoo') exists is as follows:

```
IF TYPE("This.oFoo") == "O" AND !ISNULL(This.oFoo)
```

You first have to check if it is of type 'Object', and if it is, if it isn't .NULL. In the last few years since VFP was in beta, I can't begin to imagine how many times I typed a line similar to that one above. But someone recently pointed out to me that every single object in VFP has a Name property, so you can always test for that. So the above line is shortened to:

```
IF TYPE("This.oFoo.Name") == "C"
```

If This.oFoo doesn't exist, then the above function will return "U"; otherwise, you get back "C". Simple, huh?

Where to Get It

The eLockScreen class is part of the entire eBizobj class library package, which can be downloaded from my website at <http://www.leafe.com/dlfile?which=ebizobj.zip>. I recently separated the class library into two separate libraries: cBizEd.VCX, which contains eBizobj-specific classes, and eUtils.VCX, which contains some of the goodies which I use in my development, and which can be used independently of the eBizobj classes.

Ed Leafe is an independent consultant from Rochester, New York. He has consulted for Fortune 500 firms on Object Oriented Analysis and Design, and also does a good deal of OO Programming using Visual FoxPro and the Codebook framework for VFP from Flash Creative Management. He is the author of the eBizobj class library, and the Webmaster of the Codebook Web Support Forum (http://codebook.leafe.com/), a free support forum for Codebook developers. He has also recently improved his tennis game, rising from a 3.0 ranking to a 3.5. Phone: 716/388-3930, Internet: ed@leafe.com, CompuServe 72530,1175

SAVI Codebook Application Form

I would like to receive a ***FREE*** and ***UNLIMITED*** copy of the SAVI Codebook, the only version of Codebook supported by its own newsletter. Stapled to this sheet you will find a page from the original Codebook book that proves I own a legitimate copy of Yair Alan Griver's Codebook (original price approx. $40.00) and a business card (if available). I understand that the information displayed in **BOLDFACE** is mandatory and I will not receive my package if those portions of the application are left incomplete.

First Name _____

Middle Initial _____

Last Name _____

Company Name _____

Address _____

_____

_____

City, State, Zip Code _____, _____ _____ -
_____

e-mail Address *
_____

Work Phone Number _____

Work Fax Number _____

Home Phone Number _____

I would like the SAVI Codebook Framework for _____ VFP Version 5.0a
_____ VFP Version 3.0b

_____ I would like to receive the Rational Rose (V4.0.14) model for Codebook as well,
VFP 5.0a version only.

Send completed application form to:

To: Software Assets of Virginia, Inc.
2109 Silbert Road
Kent Park
Norfolk, VA 23509-2126 USA
Attention: Free Codebook Offer

** - e-mail is the only method of delivery. If you do not provide a ***legible*** e-mail address, you will not receive the framework.