May I See Your License?

Doug Hennig

If you develop commercial applications, you may want to provide multiple levels of your application depending on what your users paid for, and prevent unauthorized users from installing and running the application. This month's article presents a license manager that can help with these tasks.

I'm currently working on an application that'll (hopefully) have wide commercial appeal. One of the issues that came up in our design sessions was the need to "license" our application. Different applications might have different requirements for licensing. Here were our needs:

- Multiple levels. The application has three "levels": "enterprise" (sometimes called "professional"), which has the complete set of features; "standard" (sometimes called "personal"), which has a smaller feature set than enterprise, usually intended for less experienced users; and "demo", which disables some of the functions (and possibly displays "nag" screens at certain places) so it can be freely distributed to those who want to evaluate the program before purchasing. Of course, other applications could have more or fewer levels.
- Single application. Rather than having to create three different versions of the application, we wanted a single version that can switch between levels somehow. In addition to being less work for us and simpler to distribute, it's easier for a user: if they've installed the demo version, all they have to do to make it into a standard or enterprise edition is pay for the level they want and somehow switch to the level. Reinstallation isn't necessary and they won't lose any data they've already entered.
- Source identification. We want to know where the user got the application. They may have downloaded a demo from our Web site or received a demo CD from a trade show, or they may have purchased it directly from us or from a reseller. They may also have "borrowed" it from a friend.
- Software registration. We want to keep the name and address of everyone who uses the program so we know who's eligible to receive support, and so we can market upgrades and other products to them in the future.
- Anti-piracy protection. We don't want the user to buy the application once and install it on every system they own or give it to their friends to install. The application is licensed on a per-seat basis, so there has to be a way to identify when the user installs it on different systems.

To satisfy these requirements, I created a license manager class and a mechanism for licensing the application. Here's how this mechanism works:

- The license manager is responsible for determining which level the application is at, so any code that needs to limit its functionality or change its behavior based on the level can simply ask the license manager for the current setting. The user can change the level by entering an "activation" code we give them; different codes are used for different levels. A single code base can therefore support multiple levels by simply putting level-specific code in a CASE statement or enabling certain buttons or menu items based on the level.
- To identify where the application came from, each physical copy of the application (that is, a CD) has an adhesive label with a unique serial number attached to it. When we sell an application to a user, we record the serial number in the user's registration record. When we ship copies to a reseller, we record the serial numbers; later, when the user they sell a copy to registers the product with us, we'll record the serial number in their registration record. This allows us to determine if someone tries to register the same copy twice (the serial number will already have been registered). Demo copies downloaded from our Web site won't have a serial number, so we'll assign one when the user contacts us to activate the application as a standard or enterprise edition.
- If a user wants to purchase an additional seat for their copy of the software, we won't give them a new serial number. Instead, we'll bump up the number of licenses purchased for that serial number.
- Before we give an activation code to the user, they have to provide us with their serial number and a "registration" number. The registration number is different for each computer (we'll discuss where it

comes from in a moment), so we can detect if someone tries to use the same serial numbered copy of the application on a different system without paying for an additional seat.

- The application includes a simple registration form in which the user enters their name and address information and their serial number. The form displays the registration number so they can tell us this number over the phone (the form also includes a Print button which prints an activation request form they can fax to us). The user enters the activation code we provide in this form.

I started looking for ways to determine a unique value for each computer. The address of the network interface card (NIC) is ideal, but the application could be installed on a standalone computer that doesn't have a NIC. The only thing I could find that could be read from VFP is the serial number of the volume the application is installed on. I realize that this isn't guaranteed to be unique, and that it can be changed. However, since we don't advertise where the registration number comes from (and it's actually an encoded version of the serial number), for our purposes, it'll have to do. I welcome suggestions for a better source of unique values!

SFLicenseMgr

Let's take a look at the license manager class. SFLicenseMgr, in SFLICENSE.VCX, is a subclass of SFCustom, our custom base class in SFCTRLS.VCX. Table 1 shows the custom properties I added to this class.

Property	Description
cActivation	The activation code. This property has an access method that reads the value from the Registry (if possible) the first time it's needed and an assign method that adjusts the application level.
cRegistrationNumber	The registration number. This property has an access method that determines the value the first time it's needed and an assign method that makes it read-only to anything but this class.
cRegistryKey	The key of the Registry entry (in HKEY_CURRENT_USER) where registration information for the application is stored.
cSerialNumber	The serial number. This property has an access method that reads the value from the Registry (if possible) the first time it's needed.
nAppLevel	The application level. 0 means demo, 1 means standard, and 2 means enterprise. This property has an access method that determines the value the first time it's needed and an assign method that makes it read-only to anything but this class.
oRegistry	A reference to an SFRegistry object.

Table 1. Properties of SFLicenseMgr.

Since SFLicenseMgr stores registration information in the Registry, it needs a means of reading from and writing to the Registry. Rather than coding that in this class, we'll reuse the SFRegistry class I presented in my December 1998 column ("Mining for Gold in the FFC"). A reference to an SFRegistry object goes into the oRegistry property; the ReleaseMembers method (called when the object is destroyed) nulls this property so object references are properly cleaned up.

Let's see a simple example of how the licensing manager is used, then we'll dig into the code. DEMO.PRG has the following code:

```
* Instantiate SFRegistry and SFLicenseMgr objects.

oRegistry            = newobject('SFRegistry', ;
  'SFFFC.VCX')
oRegistry.lCreateKey = .T.
oLicense             = newobject('SFLicenseMgr', ;
  'SFLicense.vcx')

* Point the license manager at the Registry object and
* tell it what Registry key to use.

oLicense.oRegistry    = oRegistry
oLicense.cRegistryKey = 'Software\Stonefield ' + ;
```

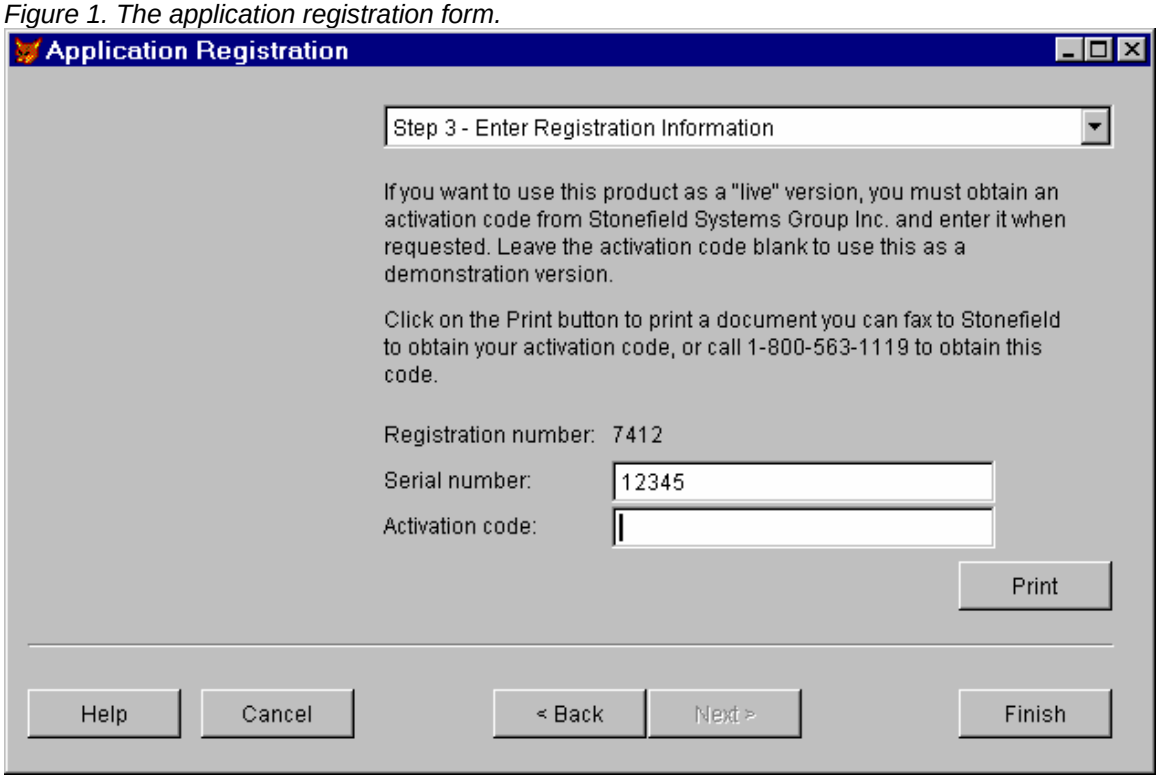
```
  'Systems Group Inc.\My Application\Registration'

* If we're in demo mode, display the registration form.

if oLicense.nAppLevel = 0
  do form Registration with oRegistry, oLicense
endif oLicense.nAppLevel = 0

* Display a message telling the user what level they're
* at.

do case
  case oLicense.nAppLevel = 0
       messagebox('You are running a demo version.')
  case oLicense.nAppLevel = 1
       messagebox('You are running a standard version.')
  otherwise
       messagebox('You are running an enterprise version.')
endcase
```

The first time this program is run, there's no registration information in the Registry, so the application is in demo mode (nAppLevel is 0). In that case, a registration form is displayed so you can see the registration number and enter the serial number and activation code (we'll see how to determine the activation code later). After the form is closed, nAppLevel may have changed (you may have entered an activation code for either standard or enterprise editions). The CASE statement shows which version you are using; similar code could be used to enable or disable certain functions in the application or change the application's appearance or behavior.

The Registration Process

The registration form is a "wizard" form that leads the user through the steps of entering their registration information. Step 1 of the wizard provides some information about the process, step 2 allows them to enter their name and address, and step 3 (shown in Figure 1) is where the registration number is displayed and they enter the serial number and activation code.

Figure 1. The application registration form.



REGISTRATION.SCX is based on SFWizardForm, a wizard form class I presented in my May 1998 column ("Build Your Own Wizards"). The Init method reads any existing name and address information the user may have previously entered from the Registry, and puts the values of the cRegistrationNumber and cSerialNumber properties of the license manager object into its own properties. Both of these properties have access methods that cause them to be determined the first time they're needed. In the case of cSerialNumber, the value is read from the Registry. Here's the code for cRegistrationNumber_Access:

```
local laVolume[1], ;
  lcVolumeSerial
if empty(This.cRegistrationNumber)
  GetVolumeInfo(justdrive(GetAppDirectory()), @laVolume)
  lcVolumeSerial = transform(abs(laVolume[2]))
  This.cRegistrationNumber = sys(2007, lcVolumeSerial)
endif empty(This.cRegistrationNumber)
return This.cRegistrationNumber
```

If the registration number is empty, GetVolumeInfo.prg is called to read information about the drive the application is on (GetAppDirectory.prg, which I presented in my November 1999 column, "Long Live PRGs!", determines the directory the application is running from, which may not be the current directory) and fills the specified array with that information. Element 2 of the array contains the volume serial number. We won't look at GetVolumeInfo.prg (or Hex2Decimal.prg, which it calls) here; it uses several Window API functions to read the volume information, and specially handles a problem Windows 95/98 systems have on remote volumes. The volume serial number is then "encoded" using SYS(2007), which returns the checksum of a string. Because the checksum is only four or five digits long, it's possible that given enough customers who buy the application, this value won't be unique. However, the combination of serial number and registration number should be unique.

When the user enters an activation code, the textbox calls the ValidateActivationCode method of the license manager. This method expects the activation code, registration number, and serial number to be passed, and returns .T. if the code is valid for either standard or enterprise editions. Here's the code for that method:

```
lparameters tcActivation, ;
  tcRegistration, ;
  tcSerial
local llReturn, ;
  lcActivation1, ;
  lcActivation2
with This
  llReturn = not empty(tcActivation)
  if llReturn
    lcActivation1 = .CreateActivationCode(;
      tcRegistration, tcSerial, 1)
    lcActivation2 = .CreateActivationCode(;
      tcRegistration, tcSerial, 2)
    llReturn      = lcActivation1 == tcActivation or ;
      lcActivation2 == tcActivation
  endif llReturn
endwith
return llReturn
```

This method calls the CreateActivationCode method, which determines the activation code for a given registration number, serial number, and application level combination, for both levels 1 (standard) and 2 (enterprise). If neither match the entered value, the method returns .F.

The CreateActivationCode method uses a fairly crude yet effective method of creating a number from two strings. It uses a different algorithm for each application level. In both cases, the activation code is the sum of the ASCII values of the each character in the strings, with a factor applied to each character that weights the value based on its position in the string. You can (and should) change this algorithm to match your needs.

```
lparameters tcRegistration, ;
```

```
  tcSerial, ;
  tnLevel
local lcSerial, ;
  lnActivation, ;
  lnI
lcRegistration = upper(tcRegistration)
lnActivation   = 0
for lnI = 1 to len(lcRegistration)
  do case
    case tnLevel = 1
      lnActivation = lnActivation + ;
        asc(substr(lcRegistration, lnI, 1)) * 2^lnI
    case tnLevel = 2
      lnActivation = lnActivation + ;
        asc(substr(lcRegistration, lnI, 1)) * lnI^2
  endcase
next lnI
for lnI = 1 to len(tcSerial)
  do case
    case tnLevel = 1
      lnActivation = lnActivation + ;
        asc(substr(tcSerial, lnI, 1)) * 2^lnI
    case tnLevel = 2
      lnActivation = lnActivation + ;
        asc(substr(tcSerial, lnI, 1)) * lnI^2
  endcase
next lnI
return transform(lnActivation)
```

When the user clicks on the Finish button in the registration form, the Finish method saves the name and address information into the Registry and sets the license manager's cSerialNumber and cActivation properties to the values the user entered. cActivation has an assign method that calls the IsActivated method to determine if the application is activated and if so at what level. Like ValidateActivationCode, IsActivated calls CreateActivationCode to determine if the activation code (this time, in cActivation rather than passed as a parameter) matches a valid code for any level, and sets nAppLevel accordingly. Here's the code:

```
local llReturn, ;
  lcActivation1, ;
  lcActivation2
with This
  llReturn = not empty(.cActivation)
  if llReturn
    lcActivation1 = .CreateActivationCode(;
      .cRegistrationNumber, .cSerialNumber, 1)
    lcActivation2 = .CreateActivationCode(;
      .cRegistrationNumber, .cSerialNumber, 2)
    do case
      case lcActivation1 == .cActivation
        llReturn   = .T.
        .nAppLevel = 1
      case lcActivation2 == .cActivation
        llReturn   = .T.
        .nAppLevel = 2
      otherwise
        llReturn   = .F.
        .nAppLevel = 0
    endcase
  else
    .nAppLevel = 0
  endif llReturn
endwith
return llReturn
```

Finally, the Finish method calls the SaveSettings method of the license manager, which saves the serial number and an encrypted copy of the activation code (encrypted using ENCRYPT.PRG, which I presented in my April 1998 column, "Application Security"; we're *really* reusing stuff in this article <s>) in the Registry.

Now that we've seen the majority of the code in SFLicenseMgr, notice an interesting thing in DEMO.PRG. It didn't set any properties of the SFLicenseMgr object (other than oRegistry), and yet nAppLevel is set correctly. How did that work? The access method of nAppLevel calls IsActivated the first time it's accessed. IsActivated accesses cRegistrationNumber (which has an access method that calculates it the first time it's accessed), cSerialNumber (which has an access method that reads the value from the Registry the first time it's accessed), and cActivation (which also has an access method that reads the value from the Registry the first time it's accessed, but decrypts the encrypted value), and then sets nAppLevel accordingly. Thus, simply asking for nAppLevel starts a chain of events that populates several properties.

OK, so how do you get an activation code so you can put DEMO.PRG into different levels? DO ACTIVATION WITH "12345", 1 to display the activation code for serial number 12345 and level 1, or DO ACTIVATION WITH "12345", 2 to get the activation code for level 2. Then, run DEMO and in the registration form, enter "12345" for the serial number and the appropriate activation code for the desired level. The messagebox should properly indicate the level the application is running at. To reset the application in demo mode, run REGEDIT, find the "Software\Stonefield Systems Group Inc.\My Application\Registration" key under HKEY_CURRENT_USER, and delete the Activation value.

Conclusion

There are a couple of things the licensing scheme I described in this document doesn't address: allowing the user to register online and handling the case where the application is installed and run from a server (assuming the client has the VFP runtime files installed on their system). The former can be implemented using some mechanism (such as ASP and a VFP COM component or West Wind Web Connection) that uses SFLicenseMgr to assign the activation code. The latter could be handled in a variety of ways, including a license table that tracks when users run and exit the application or keeping a lock on a "license" file (one per concurrent user); in either case, the application would give an error if the number of concurrent users exceeds the number of purchased licenses.

As I noted earlier, using the volume serial number isn't the most foolproof means of determining whether the application is installed on different systems, and the algorithms to convert the registration number and serial number into an activation code is fairly crude. I welcome any suggestions to improve either of these (or any other) ideas.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.