The Best of Both Worlds

Doug Hennig

You don't have to sacrifice performance for ease of installing new versions of applications. Using STARTAPP, you can run applications from local drives for best performance, and yet have them automatically updated from a common server directory when you install a new version. This solution also gives a secondary benefit: you can easily change where your data files are located without rebuilding your application.

Back in the days when we wrote single user applications, everything went on one drive: program files, runtime library files, and data. Then we moved to a networked world where shared access to applications and data was vital. So, we installed everything on the server: program files, runtime library files, and data. However, while our users gained shared access to their applications, they also took a performance hit, since everything had to load off the server. Those unfortunate souls running over WAN connections *really* had it tough. Later, Microsoft made things a little faster (but more work for installation) by requiring the runtime library files be installed on each workstation.

A few years ago, we decided we wanted the best of both worlds: local applications but shared data. Having the application running off a local drive meant that it loaded faster and used the local drive for temporary or static files (which gave additional performance improvements), but still accessed the shared data from the server. The big downside of using this approach, of course, is that when program files change, they must be manually installed on each workstation, which is an incredible pain. This article presents a solution to that problem: STARTAPP.

STARTAPP is a small application loader program. Its sole job is to ensure that the copy of the application on the user's local drive is the latest version before executing it. When a user starts the application, they actually run STARTAPP.EXE rather than the application itself. STARTAPP checks the server to see if the program files installed there are more recent than the ones of the local drive, and if so, copies them to the local drive. It then runs the application.

This scheme makes installing a new version of an application a snap: simply copy the updated program files to the server. The next time a user runs the application, STARTAPP will automatically update their version of the application before running it. This is especially important in a situation where the application is running 24 hours a day; if at least one person is running the application all the time, how can you install an updated version if they're running the one on the server you want to overwrite? Using this approach, that isn't a concern, unless the data structures change, which of course will require exclusive access to the tables. In that

case, you might want an automated mechanism to force users off the system, such as an application-wide timer that periodically checks for a signal that the application should be terminated (such as a file that normally doesn't exist suddenly appearing).

Where'd My Application Go?

STARTAPP needs to read from somewhere information about how to do its job. It needs to know where on the server to find the "home" directory for the application (where the latest copy of the program files will be located), what program files to check and possibly copy to the local workstation, and what application to execute after it's done (including any parameters that should be passed to the application). This type of information could go into one of four places: a MEM file, a table, an INI file, or the Windows Registry.

We rejected a MEM file right away because of the difficulty in updating it. We rejected a table for a similar reason: it's difficult to change settings stored in a table from outside an application unless you have a development copy of FoxPro available (which you rarely do at a client site). One of the things we frequently need to do when installing an application at the client site is figure out where to put it (has this ever happened to you: they told you it would be on drive N: but now that drive is full, so it's going on M: instead). So it came down to an INI file versus the Windows Registry. After much consideration, we decided to go with an INI file. Although Microsoft is discouraging the use of INI files, and the Registry makes more sense for some types of settings (especially user preferences), we found that INI files worked better in this instance for several reasons:

- INI files, especially those located in the same directory as the application rather than the Windows or System directories, minimize what I call "poops". The reason I call them "poops" is obvious to any pet owner: like a pet that hasn't been house-trained yet, unruly applications leave little extraneous things all over your hard drive without your knowledge, making them a nightmare to find and clean up after if you decide to remove the application. Unless an application has an uninstall feature, Registry settings can be around years after the application is no longer used.

- INI files aren't operating system specific. You can use INI files in Windows 3.1, Windows 95, Windows NT, DOS, Unix, and Macintosh. The Registry is only supported in Windows 95 and NT.

- Since they're just text files, INI files are easier for users to edit than Registry settings. I've talked inexperienced users through using Notepad to edit an INI file lots of times over the phone, but wouldn't dare try it using REGEDIT to edit some Registry settings.

Technical Details

STARTAPP.EXE is built from STARTAPP.PJX, which contains just two files: STARTAPP.PRG (the main program) and READINI.INI. All of these files are available from the Download Site.

STARTAPP.PRG accepts a single, optional parameter: the name of the INI file containing the information it needs. If it isn't specified, the current directory must contain one and only one INI file or STARTAPP will give an error. The INI file must contain a section called [Server] and the following keys:

- **Directory**: the complete path to the server directory where the program files can be located.
- **Copy<n>**: the name of a file to copy, where <n> is the number of the file (such as Copy1, Copy2, Copy3, etc.).
- **Run**: the name of the application to run, including any parameters to pass to it.

Here's an example:

```
[Server]
Directory = N:\SALES
Copy1 = SALES.APP
Copy2 = SALES.INI
Copy3 = BTRIEVE.EXE
Copy4 = DBFTRV16.DLL
Copy5 = DBFTRVE.FLL
Copy6 = WBTRCALL.DLL
Copy7 = FOXTOOLS.FLL
Run = SALES.APP
```

These entries tell STARTAPP to look in N:\SALES for the files to check, to check and possibly copy the seven files listed, and to run SALES.APP when it's done checking.

Here's an example of how STARTAPP is called (-T tells FoxPro not to display its splash screen):

```
STARTAPP.EXE -T APP.INI
```

This might be the Target setting in the Properties dialog for the Windows 95 shortcut for the application or the Command Line setting for the Windows 3.1 icon. If APP.INI is the only INI file associated with the application, you could leave that parameter out of this command if desired. The current directory should be set to the one containing STARTAPP.EXE on the local drive, so put that into the Start In setting in the Properties dialog for the Windows 95 shortcut or the Working Directory setting for the Windows 3.1 icon.

Of course, you don't have to name the EXE file STARTAPP.EXE. It can be renamed for each application if desired; for example, for a sales application, you might call it SALES.EXE. However, avoid naming the application file called by STARTAPP the same name, since FoxPro will get confused. For example, if you rename STARTAPP.EXE to SALES.EXE, don't have it run SALES.APP; instead, name your application file something different like SALESOVL.APP or MAIN.APP.

Before we look at the code for STARTAPP.PRG, I'll briefly mention its cross-version capability. One of the design goals for STARTAPP was to work with all the applications we write, meaning FoxPro 2.6 for DOS and Windows and VFP 3 and 5. Obviously, we need separate project files for each platform, but we wanted to keep the same code base if possible. This meant we either had to write code for the lowest common denominator (FoxPro 2.x) and avoid using VFP-specific commands, functions, and syntax (such as LPARAMETERS, LOCAL, and dropping the = in front of a function call when you don't care about the return value), or use #IF to bracket version-specific code so the FoxPro compiler only sees code for the current version to avoid syntax errors. We actually used both techniques in this tool; STARTAPP.PRG was written using only FoxPro 2.x code while READINI.PRG (which we'll see later) uses code bracketing since we use some VFP-specific code.

Here's the code for STARTAPP.PRG:

```
parameters tcINIFile

private laINIFiles, ;

  lnINIFiles, ;

  lcINIFile, ;

  lcServer, ;

  lcApp, ;
```

```
  lnFiles, ;
  lcFile, ;
  laFiles, ;
  lnI

* If the name of the INI file was passed, ensure
* it exists.

set talk off
set safety off
if type('tcINIFile') <> 'C' or empty(tcINIFile)
  lnINIFiles = adir(laINIFiles, '*.INI')
  if lnINIFiles <> 1
    wait window 'You must specify the name of the ' + ;
      'INI file for this application.'
    return
  endif lnINIFiles <> 1
  lcINIFile = fullpath(laINIFiles[1, 1])
else
  lcINIFile = fullpath(tcINIFile)
endif type('tcINIFile') <> 'C' ...
if not file(lcINIFile)
  wait window tcINIFile + ' cannot be located.'
  return
endif not file(lcINIFile)

* Get the server directory and name of the
* application to run.

lcServer = ReadINI(lcINIFile, 'Server', ;
  'Directory', '%')
if lcServer = '%'
```

```
  wait window tcINIFile + ;
    ' does not contain a valid entry for Directory'
  return
endif lcServer = '%'
lcServer = lcServer  + ;
  iif(right(lcServer,  1) $ '\:', '', '\')
lcApp    = ReadINI(lcINIFile, 'Server', 'Run', '%')
if lcApp = '%'
  wait window tcINIFile + ;
    ' does not contain a valid entry for Run'
  return
endif lcApp = '%'

* Get a list of files to check on the server.

lnFiles = 0
do while .T.
  lcFile = ReadINI(lcINIFile, 'Server', ;
    'Copy' + ltrim(str(lnFiles + 1)), '%')
  if lcFile = '%'
    exit
  endif lcFile = '%'
  lnFiles = lnFiles + 1
  dimension laFiles[lnFiles]
  laFiles[lnFiles] = lcFile
enddo while .T.

* Check the specified files in the server directory
* and, if necessary, copy each to the workstation.

for lnI = 1 to lnFiles
  do VerCheck with laFiles[lnI], lcServer
```

```
next lnI

wait clear


* Now start the application.


do &lcApp

return
```

Note that the specified application is called using macro expansion rather than a named expression. This allows you to specify parameters to pass to the application. For example, if you want to pass the current date and directory to the application, you'd use something like the following in the INI file:

```
Run = SALES.APP with date(), sys(5) + curdir()
```

The VerCheck function, located in STARTAPP.PRG, does the dirty work: it compares versions of a single file on the server and on the workstation using their datetime stamps. If the file doesn't exist on the workstation or is different than the version on the server, VerCheck copies it to the workstation. VerCheck uses WAIT WINDOW to display what it's doing; you can remove these statements if you don't want anything displayed. Here's the code for this routine:

```
procedure VerCheck

parameters tcFile, ;

  tcServer

private laServer, ;

  laWorkStn


* Get the version info for the server and

* workstation copies of the file.


wait window 'Checking server and workstation ' + ;

  'versions of ' + tcFile + '...' nowait
```

```
= adir(laServer,  tcServer + tcFile)
= adir(laWorkStn, tcFile)

* If the file doesn't exist on the workstation or
* doesn't match the server version, we must copy it.

if file(tcServer + tcFile) and (not file(tcFile) or ;
  laServer[1, 3] <> laWorkStn[1, 3] or ;
  laServer[1, 4] <> laWorkStn[1, 4])
  wait window 'Updating workstation version of ' + ;
    tcFile + '...' nowait
  copy file (tcServer + tcFile) to (tcFile)
endif file(tcServer + tcFile) ...
return
```

Reading From an INI File

READINI.PRG reads an entry from a section in an INI file. It accepts the following parameters:

- tcINIFile: the INI file to look in. For VFP, you must pass the fully qualified path to the INI file, not just the relative path (for example, even if the current directory is C:\MYAPP, you must specify “C:\MYAPP\APPLIC.INI”, not “APPLIC.INI”) or the function won’t work properly.
- tcSection: the section to look for.
- tcEntry: the entry to look for.
- tcDefault: the default value to use if the INI file, section, or entry aren’t found.

It returns the value of the entry or tcDefault if the entry isn’t found

READINI has different code for VFP and FoxPro 2.x; the code is bracketed in an #IF statement so FoxPro only compiles the code applicable for the current version. The reason for the different code is that VFP has the ability to call Windows API functions using the DECLARE

command, which defines where the function is found and how to call it. The Windows GetPrivateProfileString function will do everything we need, so the VFP code doesn't do much other than setting up for the call to this function. FoxPro 2.x doesn't have the capability of calling Windows API functions (well, it sort of does: I could have used the RegFN and CallFN functions in FOXTOOLS.FLL, but decided to make this a completely self-contained EXE that doesn't require FOXTOOLS), so its code consists of a brute force mechanism to read and parse the INI file.

Here's the code for READINI.PRG

```
#define ccNULL             chr(0)
#define cnBUFFER_SIZE      2048
#if 'Visual' $ version()

* Visual FoxPro code.

lparameters tcINIFile, ;
  tcSection, ;
  tcEntry, ;
  tcDefault
local lcBuffer, ;
  lcDefault
declare integer GetPrivateProfileString in Win32API ;
  string cSection, ;
  string cEntry, ;
  string cDefault, ;
  string @ cBuffer, ;
  integer nBufferSize, ;
  string cINIFile
lcBuffer  = replicate(ccNULL, cnBUFFER_SIZE)
lcDefault = iif(type('tcDefault') <> 'C', '', ;
  tcDefault)
= GetPrivateProfileString(tcSection, tcEntry, ;
```

```
    lcDefault, @lcBuffer, cnBUFFER_SIZE, tcINIFile)
return strtran(lcBuffer, ccNULL, '')
#else

* FoxPro 2.x code.

parameters tcINIFile, ;
  tcSection, ;
  tcEntry, ;
  tcDefault
private lnHandle, ;
  lcValue, ;
  lcLine

* If the INI file doesn't exist, return the default
* value.

if not file(tcINIFile)
  return tcDefault
endif not file(tcINIFile)

* Open the INI file.

lnHandle = fopen(tcINIFile)
lcValue  = tcDefault
if lnHandle >= 0

* Find the section heading.

  lcLine = '%'
  do while lcLine <> '[' + tcSection + ']' and ;
    not feof(lnHandle)
```

```
    lcLine = fgets(lnHandle)
  enddo while lcLine <> '[' + tcSection + ']' ...

* If we found the section, find the entry.

  if lcLine = '[' + tcSection + ']'
    do while lcLine <> tcEntry and not feof(lnHandle)
      lcLine = fgets(lnHandle)
    enddo while lcLine <> tcEntry ...

* If we found the entry, get the value.

    if lcLine = tcEntry
      lcLine  = alltrim(substr(lcLine, ;
        len(tcEntry) + 1))
      lcValue = alltrim(substr(lcLine, 2))
    endif lcLine = tcEntry
  endif lcLine = '[' + tcSection + ']'
  = fclose(lnHandle)
endif lnHandle >= 0
return lcValue
#endif
```

Where'd My Data Go?

One thing you may be wondering about: if the application is running from the local drive and the data is located on the server, how does the application know where to find the data, especially since the location on the server might change? Here's where the INI file comes in again: we store the data for our applications in a DATA subdirectory of the application's "home" directory on the server, so our application uses READINI.PRG to return the Directory entry in the INI file and then appends "DATA\" to it. We can then open the database and tables in that directory. Here's an example:

```
oApp.cServer = ReadINI(oApp.cINIFile, 'Server', ;
  'Directory', '')
oApp.cDataDir = oApp.cServer + ;
  iif(right(oApp.cServer, 1) $ '\:', '', '\') + ;
  'DATA\'
open database (oApp.cDataDir + 'MYDATA')
```

(oApp is our application object; we store the directories in properties of the object so anything needing to know where program and data files are installed can access them).

One slight fly in the ointment: the DataEnvironment of a form has a hard-coded path to databases and free tables. Even if the database has been opened, as soon as you run a form, it'll bomb because it can't find the database where it expects to (wherever it was on your system when you created the form). Fortunately, this path is stored in read-write properties in the DataEnvironment (Database for cursors contained in a database and CursorSource for free tables), so we just need to change these properties before trying to open the tables. Here's how we handle it: we put the following code into the Load method of our base form class (SFForm, which is contained in SFCTRLS.VCX):

```
* If the tables haven't been opened yet, set the data
* directory for all databases and free tables, then
* open the tables.

if type('oApp') = 'O' and not isnull(oApp) and ;
  type('This.DataEnvironment') = 'O' and ;
  not isnull(This.DataEnvironment) and ;
  not This.DataEnvironment.AutoOpenTables
  oApp.SetDataDirectory(This.DataEnvironment)
  This.DataEnvironment.OpenTables()
endif type('oApp') = 'O' ...
```

This code ensures that we have an application object (oApp), that we have a DataEnvironment and that AutoOpenTables is set to .F. so tables aren't automatically opened (the DataEnvironment opens tables before the form's Load method fires if AutoOpenTables is

set to its default .T., and that's too soon for this solution). If all that is true, it calls the SetDataDirectory method of the oApp object, which adjusts the DataEnvironment to use the proper directory (we'll see this code in just a moment). It then tells the DataEnvironment to open the tables now that it knows where to find them. Putting this into the Load method ensures the tables are opened from the proper directory before any object sitting on the form is instantiated.

The SetDataDirectory method accepts a reference to a DataEnvironment, then goes through each cursor object in the DataEnvironment and changes either its Database or CursorSource property to use the directory stored in the cDataDir property of the application object. The code for SetDataDirectory is listed below. If you don't use an application object or don't want to put this code into it, you could put this code into a PRG instead.

```
lparameters toDE
local laObjects[1], ;
  lnObjects, ;
  lcDirectory, ;
  lcCommon, ;
  lnI, ;
  loObject, ;
  lcDatabase, ;
  lcTable

* Get a list of the members of the DataEnvironment,
* and get the data directory from the cDataDir
* property.

lnObjects   = amembers(laObjects, toDE, 2)
lcDirectory = This.cDataDir

* Look at each member object, but only process
* cursors.

for lnI = 1 to lnObjects
```

```
  loObject = evaluate('toDE.' + laObjects[lnI])
  if upper(loObject.BaseClass) = 'CURSOR'
    lcDatabase = loObject.Database

* If this is a free table, adjust the CursorSource
* property.

    if empty(lcDatabase)
      lcTable = loObject.CursorSource
      loObject.CursorSource = lcDirectory + ;
        substr(lcTable, rat('\', lcTable) + 1)

* This cursor is part of a database, so change the
* database property.

    else
      loObject.Database = lcDirectory + ;
        substr(lcDatabase, rat('\', lcDatabase) + 1)
    endif empty(lcDatabase)
  endif upper(loObject.BaseClass) = 'CURSOR'
next lnI
```

Because this code exists in our class definitions, we don't have to worry about it at all when we create a new form. The only thing we have to do manually in a form is set the DataEnvironment's AutoOpenTables property to .F. (we can't do this in the SFForm class because a class definition doesn't have a DataEnvironment).

This scheme makes testing the application under different conditions really easy. For example, for normal development, I have the Directory entry in the INI file set to blank. Thus, when I run the application, it uses the data in the DATA subdirectory of the current directory on my local drive. When I want to test network performance, I simply change the INI file so Directory points to a server directory.

This scheme can also be used for accessing production versus test data. For example, you could have two INI files, one named PROD.INI and the other TEST.INI. PROD.INI has a Directory

entry pointing to the server directory containing the program files and production data, while TEST.INI's Directory points to a test directory on the server or even the local drive. The user then simply runs either the Production program (which executes STARTAPP.EXE -T PROD.INI) or the Test program (which runs STARTAPP.EXE -T TEST.INI).

If you want more flexibility than always assuming the data will be in a subdirectory called DATA, you could add another entry to the INI file defining where the data is located and use that entry rather than Directory in the code above.

Issues

There are a couple of issues you should be aware of with STARTAPP. The first is that the RETURN TO MASTER command will return control to STARTAPP, not the main program in your application, and since the last thing STARTAPP does is call your application, RETURN TO MASTER will effectively terminate the application. If you use RETURN TO MASTER as part of your error handler to cancel execution of the routine causing the error, you'll have to rewrite this as RETURN TO <main program>.

The second issue is that if STARTAPP.EXE itself changes (for example, you add new behavior such as eliminating the WAIT WINDOW messages), you must manually install it on each workstation, since it doesn't (and can't) automatically copy itself from the server to the workstation. Also, changes to the INI file on the server won't be utilized until the second time the application is run. The first time the user runs STARTAPP, it reads the local copy of the INI file before copying the new INI file to the workstation. The second time STARTAPP is run, the new INI file is used. If this is a concern, you could change STARTAPP.PRG to only read the name of the server directory from the local INI file and then use the INI file in that directory for everything else.

Example

A trivial example of the use of STARTAPP is provided on the Download Site. When you unzip the files into a directory, you'll end up with some common files (STARTAPP.PRG and READINI.PRG, as well as sample files SAMPLE.PRG, APPLIC.INI, and TEXTDOC.TXT) and three version-specific subdirectories containing project and APP files for FoxPro 2.6, VFP3, and VFP 5. To try out the sample, do the following:

- Copy SAMPLE.APP from the appropriate version-specific directory and the common TEXTDOC.TXT to a test directory either on a local or server drive. We'll call this the "server" directory to be clear.

- Build STARTAPP.EXE from the project in the appropriate version-specific directory.
- Copy STARTAPP.EXE and APPLIC.INI to a temporary local directory. We'll call this the "local" directory to be clear.
- Edit APPLIC.INI in the local directory so its Directory entry contains the name of the server directory.
- Run STARTAPP.EXE from the operating system (this assumes you have the FoxPro runtime libraries installed; if not, simply change directory to the local directory and run it from FoxPro). It will copy SAMPLE.APP and TEXTDOC.TXT from the server directory to the local directory, then run SAMPLE.APP (which just displays a list of files in the current directory).
- Change the contents of the server directory version of TEXTDOC.TXT.
- Run STARTAPP.EXE again. You should find that the copy of TEXTDOC.TXT in the local directory was updated as expected.

Summary

STARTAPP gives you the best of both worlds: you can improve performance by running applications from local drives, and automatically update them when you install a new version. A nice secondary benefit of this is that the same INI file used to define where the application is installed on the server can tell your application where its data is located. We've used STARTAPP for several years now for applications written in FoxPro 2.6 for DOS and Windows and Visual FoxPro applications with great success. I hope you find it as useful as we do.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. 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 user groups and regional conferences all over North America, and will be speaking at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). 75156.2326@compuserve.com or dhennig@stonefield.com.