Column: From the Editor

Figures:
File for Subscriber Downloads:

VFP Data on Your Handheld: Building a VFP App

Whil Hentzen

Last month we built a simple application in Satellite Forms, deployed it to our handheld, and then transferred data back and forth from our PC and our handheld. However, the data on our PC resided into a single DBF – and you had to know that a HotSync occurred in order to see the changes made to the DBF. Furthermore, the transfer overwrote the entire table each time. This month, I'm going to show you how to build a Visual FoxPro application that will automatically detect a hot sync – and get ready to simultaneously coordinate changes made both on your PC and on your handheld.

We now have a database application running on our handheld. Presumably we also have a VFP application running on our PC (or server). We now have two goals.

Goal #1 - Synchronization

What we would like – ideally – is to have an application running on our desktop (or server) that houses a superset of the data that we're accessing on our handheld.

For example, suppose your desktop has your entire To Do list dating back five years. You may just want the last two weeks of tasks as well as anything upcoming to be accessible on your handheld. Thus, you'll want to move a subset of your PC's data to that intermediate data store – the dBASE V DBF that the SatForms Table Designer creates - that gets transferred to your handheld. In other words, you might have 10,000 records in your VFP database. You might want to copy 480 of those to the DBF that SatForms uses. When the HotSync occurs, your handheld now has those 480 records. (Remember that the old table that your handheld had is now gone – completely replaced by these 480 records.)

Then, after you've been out and about for a while, you return to the office. You've changed five of those 480 records, and added a few more. The handheld database has 483 records. When you HotSync, those handheld records are moved into the DBF.

And, just like the previous HotSync, the DBF on the PC gets completely wiped out and replaced. The DBF now has those 483 records that were on your handheld, complete with the changes you've made to five of the initial 480. You'll now need to synchronize those 483 records with your master VFP database of 10,000 records.

To make matters worse, somebody might have snuck into your office while you were gone and added 17 more things to your To Do list. (Who let your spouse in there, anyway???) During this same HotSync where you're grabbing the 483 records off of your handheld, you're going to want to send those 17 new records back to your handheld – via the DBF, of course.

Obviously, we can do this – we're database programmers. The point is that this isn't done for us automatically – we're going to have to write code to make it happen.

Goal #2 - Magic

You know what would be great? Let's suppose you wrote the VFP code that did all this synchronization between the intermediate DBF and your big VFP database – looking at time stamps, primary keys, whatever. Wouldn't it be great if, whenever you put your handheld in the cradle and hit the HotSync button, your VFP application detected this HotSync on the PC, and, after the HotSync was done moving the handheld data to or from the PC's DBF file, executed the appropriate synchronization methods that you wrote in VFP? In other words, you didn't have to do anything on your PC or in VFP – your VFP app just sat there, in the background, doing its thing for you?

That's our second goal – to make this happen automagically.

We're going to do these in reverse – first, I'll show you how to write a VFP app that uses the SatForms ActiveX control to automatically detect a HotSync, and then moves data back and forth from the DBF to a VFP data store. (Naturally, it doesn't have to be VFP. It could be SQL Server, or anything other type of data that VFP can talk to.)

Once that's done, you'll realize that the other half – the code that actually synchronizes the DBF and your VFP data – is just database programming – stuff you've been doing for twenty years – and we'll look at that next month.

Building a HotSync-Aware form in VFP

For my first trick, I'm going to create a VFP form that binds directly to that intermediate dBASE V DBF file. This will help you get used to how the ActiveX control works. Once that's running smoothly, I'll show you how to separate the intermediate DBF from the form's controls, and integrate your own data with the form and the DBF.

It's most handy if you've already got your SatForms application done, so I'm going to assume that you've followed along with last month's article and have the following files in your SFDEMO project directory:

sfdemo1.dbf
sfdemoapp.sfa

There are some other files in that directory, like the PDA and PDB files I covered last month, but they're irrelevant for our purposes right now.

In this same directory, you're going to create your VFP form. Create a form, named SFDEMO1.SCX. Open the data environment and select the SFDEMO1.DBF table. Add some text boxes, check boxes, and command buttons to it so that it will look Figure 1 once you started adding data to it. There's a copy of this form in this month's Subscriber Downloads, by the way. I've used all VFP base controls, so you just have to open up SFDEMO1.SCX.

Bind each of the controls to the appropriate field in SFDEMO1.DBF, using the control's ControlSource property. Here's the simple code in each of the command button's click events:

```
* Back
skip -1
thisform.refresh()

* Next
skip 1
thisform.refresh()

* List
on key labe ENTER keyboard "{CTRL-W}"
brow normal
on key label ENTER
thisform.refresh()

* New
local m.lcResp
m.lcResp = thisform.txtResp.value
insert into SFDEMO1 ;
  (cTask, cResp, dDue, lDone, nHrsEst, nHrsAct) ;
  values ;
  ("NewTask", m.lcResp, date(), .f., 0, 0)
thisform.refresh()
```

By the way, don't forget to set your tab order – one thing you don't have to do on a handheld!

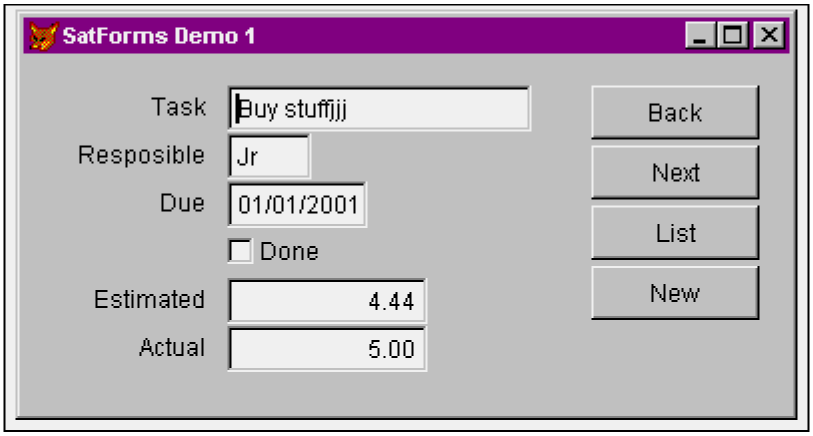


Figure 1. A simple data entry and navigation form in VFP bound to the SatForms intermediate DBF.

Run the form and make sure that you've got all the code you need in the buttons, that you've bound the controls properly, and so on.

Once you've made some changes, perhaps added a few records, close your form, and then do a HotSync with your handheld. If you have everything set up properly, you should have new data on your handheld.

Whoa! Where's the new data? It's NOT on your handheld, is it? Why the heck not? You just did a HotSync, right? Well, you have to tell the HotSync explicitly to download this DBF file. We need the SatForms ActiveX control to do so.

Adding the SatForms ActiveX control to your VFP form

Obviously, you'll need to drop the SatForms ActiveX control on your form. It should already be registered on your machine – that happened when you initially installed SatForms. Thus, the first thing you need to do (if you haven't gotten ahead of me) is register it with VFP.

Select the Tools|Options menu option, click on the Controls tab, select the ActiveX option button, and scroll down the list to the S's as shown in Figure 2.

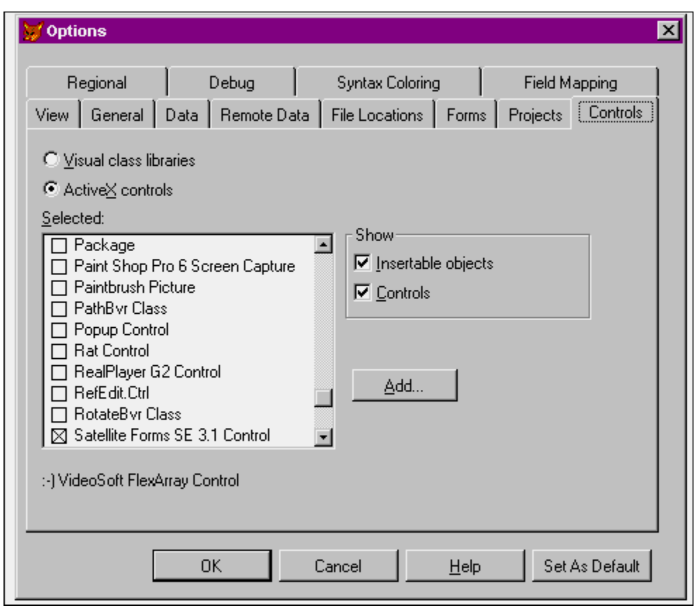


Figure 2. Registering the SatForms ActiveX control with Visual FoxPro.

Check the check box, click on Set As Default command button, and close the dialog.

Now you can access the control via the Form Controls toolbar. Open the Form Controls toolbar, click on the second button from the left (the one with the three books and the small arrow in the lower right corner), and select the ActiveX Controls menu option.

The Form Controls toolbar will change to something like Figure 3. (I've got a funky cursor set running on my machine – that's what shadow is in the figure.)

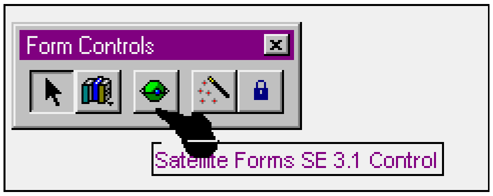


Figure 3. The Form Controls toolbar, showing just ActiveX controls.

The middle button on the toolbar represents the SatForms ActiveX Control. Click on it and then click on your VFP form, just as you do when placing any other control on a form. This control won't ever actually be visible when the form is running, so put it somewhere way far out of the way. (Milwaukee, perhaps?) See Figure 4.

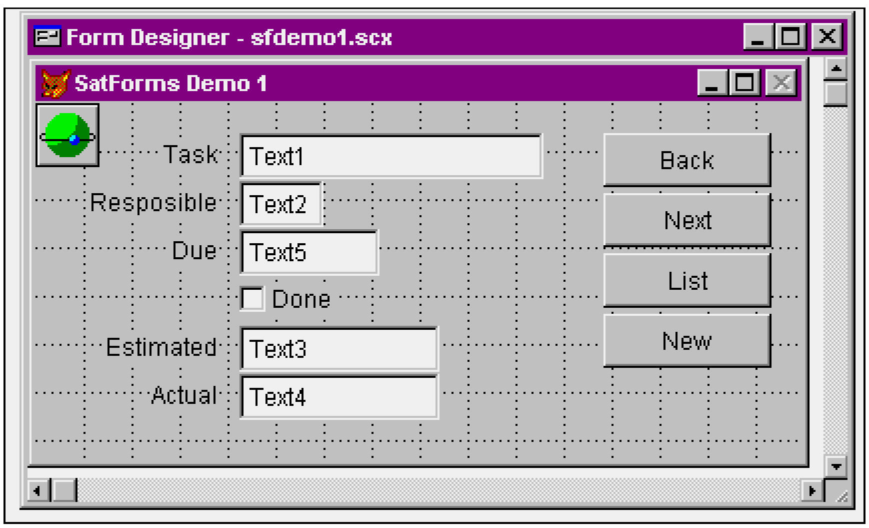


Figure 4. The VFP form with the SatForms ActiveX control.

The SatForms ActiveX control is, by default, named "olecontrol1" – why not change it to something more reasonable, like oleSatForm?

Implementing the SatForms ActiveX control

And now it's time to write some code. (Cool down, guys!) The first thing to do is put the following line in the form's Init() method:

```
thisform.oleSatForm.Object.Enabled = .t.
```

For some reason, the SatForms ActiveX control is disabled when it's dropped on a form. You'll want to use this code to turn it on or nothing else will work.

Next, double-click on the ActiveX control itself, and locate the HotSyncStatus() method. The HotSyncStatus event gets fired many times during the HotSync process – the code that you put in this method will get fired in concert with the firing of the HotSyncStatus event. This is an important point – you don't call HotSyncStatus() in your code - you put code in this method that gets executed automatically when the event is fired.

In other words, your VFP form – with this ActiveX control on it – will be running on your PC. You'll put your handheld in the cradle, press the HotSync button, and the process will start. Part of the HotSync process runs on your PC, of course. The SatForms ActiveX control on your form detects that the HotSync process is running on your PC, and generates HotSync Status events: Kerchunk, kerchunk, kerchunk. (That's what it sounds like inside an IC, by the way.)

When the HotSyncStatus event is generated, it pass StatusCode and Param parameters, so your code in the HotSyncStatus() method needs to look at these parms and do the appropriate thing.

To begin with, we're only interested in the first parameter. So what are the possible values of this parm? The StatusCode parameter can take one of three values: 1, 2 or 3.

1	Start of HotSync
2	End of HotSync
3	HotSync is running

Here's some sample code that you can put in the HotSyncStatus method to see how these parms are passed to the HotSyncStatus() method:

```
* oleSatForm.HotSyncStatus()
do case
case StatusCode = 1
 * start
 debugo "code 1 - start"
case StatusCode = 2
 * start
 debugo "code 2 - end"
case StatusCode = 3
 * running
 debugo "code 3 - running"
othe
 * a mystery
 debugo "code " + tran(StatusCode) + " mystery code"
endcase
```

Once you've put this code in your form, save your work, and run the form. Open your Debug Output window as well, of course. You should be able to navigate through records, make changes, and so on, like usual. Then press the HotSync button on your Palm cradle. You'll get output in the Debug Output like that shown in Figure 5.

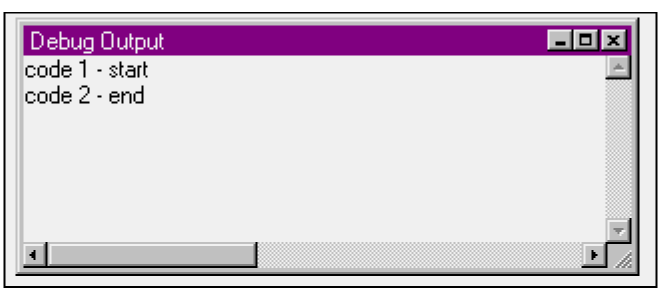


Figure 5. Events generated from the HotSyncStatus() event.

You'll notice that "code 3 – running" didn't display yet – I'll address that shortly. In the meantime, though, you can see that your VFP app is detecting the HotSync event initiated from the handheld cradle. Your VFP app is just sitting there in background, waiting to detect the HotSync.

Now it's time to do some database work.

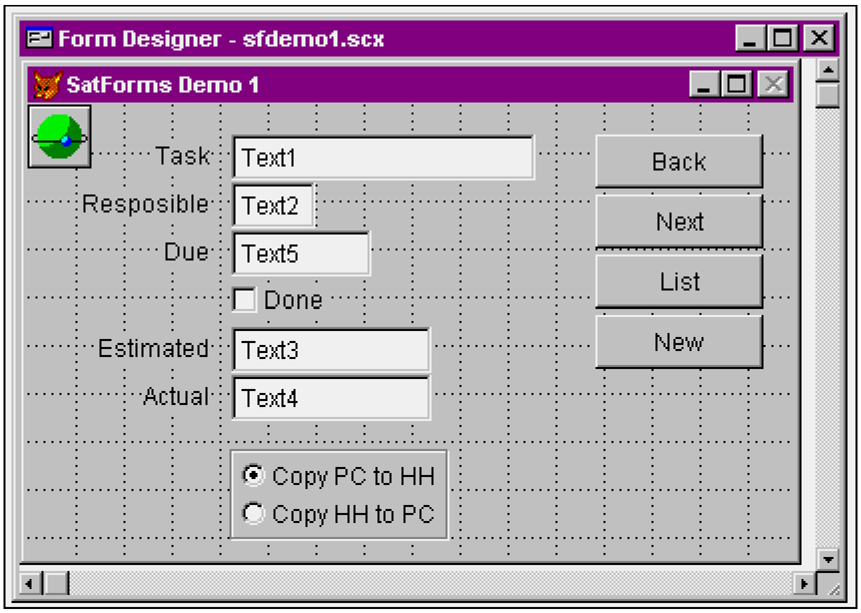
Using the SatForms ActiveX control to move data

The SatForms ActiveX control has a couple more methods that are really interesting. They're called

CopyTableToPalmPilot(<tablename>)
GetTableFromPalmPilot(<tablename>)

and, as you might guess, they're used to move a table to or from the Palm. Note that you move data one table at a time, specifying the name of the table as an argument to the method.

In order to provide a more interesting programming experience, we're going to add the ability for the user to control which method will be used in our example. Put an option group, named opgDirection, on your form like shown in Figure 6.



In your form's Init(), make sure that you initialize the value of the option group to a character string:

```
thisform.opgDirection.value = "Copy PC to HH"
```

Now we're going to add code to the HotSyncStatus() method that will look at the value of the option group and then call either the CopyTable or GetTable method as appropriate. The entire HotSyncStatus() method looks like this:

```
*** ActiveX Control Event ***
LPARAMETERS statuscode, param
do case
case StatusCode = 1
 * start
 debugo "code 1 - start"
 do case
 case thisform.opgDirection.value = "Copy PC to HH"
  debugo "code 1 - copying PC down to HH: SFDEMO1.DBF"
  This.CopyTableToPalmPilot(fullpath("sfdemo1.dbf"))
 case thisform.opgDirection.value = "Copy HH to PC"
  debugo "code 1 - copying HH up to PC: SFDemo1.DBF"
  this.GetTableFromPalmPilot(fullpath("sfdemo1.dbf"))
 othe
  messagebox("Yikes! What is the value of opgDirection? " ;
```

```
  + tran(thisform.opgDirection.value))
 endcase
case StatusCode = 3
 * running
 debugo "code 3 - running. CmdType is: " + tran(this.CmdType)
case StatusCode = 2
 * end
 debugo "code 2 - end"
othe
 debugo "code ? - otherwise: " + tran(StatusCode)
endcase
```

Note that the DO CASE structure that contains the CopyTable and GetTable methods is contained in the Start (StatusCode = 1) code segement – not in the Running (StatusCode = 3) segment as you might expect. Why?

The Running code segment gets called each time a CopyTable or GetTable method completes. You can put your own code in this segment to do “post-transfer” work – such as your own code that synchronizes the intermediate DBF with your own VFP data.

To show you how this works, I added a couple of DEBUGOUT commands in the appropriate places. Let’s see what happens when we run this.

First, run your VFP app, and enter “A” into a record in the VFP table. Then make sure that the “Copy PC to HH” option button is selected. Put your handheld in its cradle, press the HotSync button on the cradle, and wait for the sync messages to complete.

At this point, open up the matching application on your handheld, and look for the same record. You’ll see the value “A”. The Debug Output window on your PC should look like Figure 7.

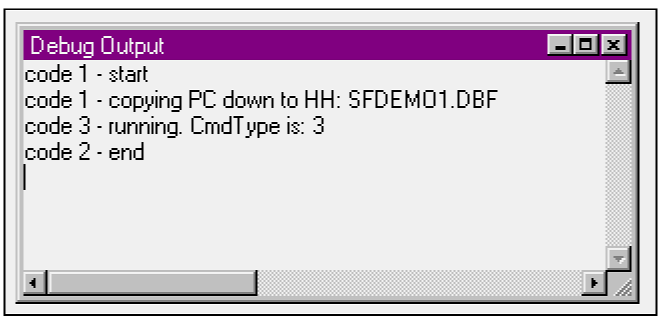


Figure 7. The Debug Output window after issuing a CopyTableToPalmPilot command.

Now, on your Palm, change “A” to “BBB”. Put the handheld back in its cradle, click the “Copy HH to PC” option button in your VFP application on your PC, and then press the Sync button on your handheld’s cradle. Wait for the sync messages to finish, and then find the record on your PC. You’ll see “BBB” has taken the place of “A”. Additionally, your Debug Output window should look like Figure 8.

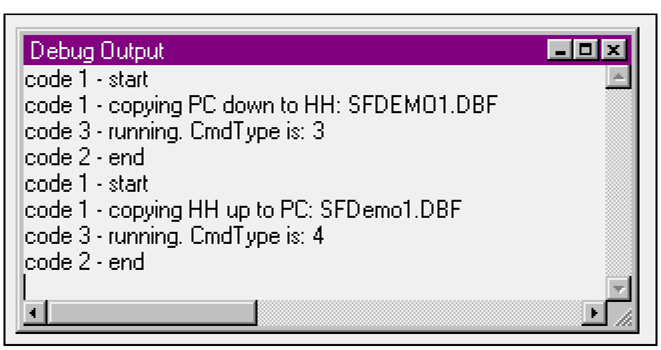


Figure 8. The Debug Output window after issuing a GetTableFromPalmPilot command.

You'll notice that the SatForm ActiveX control's CmdType value varies according to which method is being called:

```
* Property Nameof ActiveX Control: CmdType
* If CmdType is 1, then just finished CopyAppToPalmPilot
* If CmdType is 3, then just finished CopyTableToPalmPilot
* If CmdType is 4, then just finished GetTableFromPalmPilot
```

You can use this information to write reusable code that performs an appropriate function according to which type of data transfer was just performed.

Multiple tables

It'd be a pretty boring world if we just worked with single table applications all the time, wouldn't it? In this next example, we'll move more than one table back and forth, and watch how the process works. Because this is just an example, I didn't bother with hooking this second table into the application – the point is to show you how to transfer oodles of tables.

First, you'll need a second table to work with. The easiest way is to open the SatForms IDE and create a second table. I created SFDemo2.DBF with a single dummy field, and added just one record to it.

Then, in the HotSyncStatus() method, I changed to code like so:

```
do case
 case thisform.opgDirection.value = "Copy PC to HH"
  debugo "code 1 - copying PC down to HH: SFDEMO1.DBF"
  This.CopyTableToPalmPilot(fullpath("sfdemo1.dbf"))
  debugo "code 1 - copying PC down to HH: SFDEMO2.DBF"
  This.CopyTableToPalmPilot(fullpath("sfdemo2.dbf"))
 case thisform.opgDirection.value = "Copy HH to PC"
  debugo "code 1 - copying HH up to PC: SFDemo1.DBF"
  this.GetTableFromPalmPilot(fullpath("sfdemo1.dbf"))
  debugo "code 1 - copying HH up to PC: SFDemo2.DBF"
  this.GetTableFromPalmPilot(fullpath("sfdemo2.dbf"))
 othe
  messagebox("What is the value of opgDirection? " + tran(thisform.opgDirection.value))
 endcase
```

When you run this, the results in the Debug Output look like Figure 9:

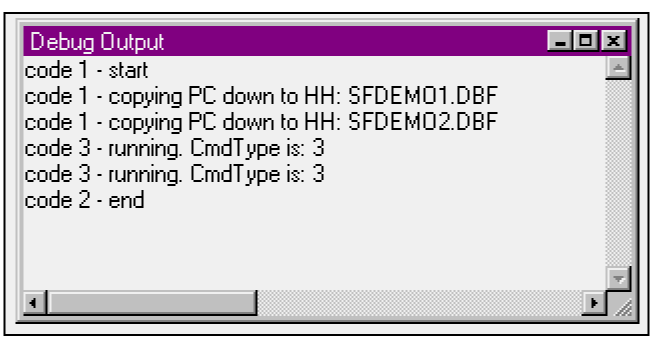


Figure 9. The Debug Output window after copying two tables.

Notice that the CopyTable method was executed twice before the code in the StatusCode = 3 segment was executed. This is intentional, and so it means that if you are relying on running code in the Running segment before another Copy or Get command is executed, you'll need to employ some fancy footwork to make it happen.

Coming Up Next Month

So those are the basics. Next month we'll explore various avenues of approach toward accomplishing our second goal – synchronizing the intermediate DBFs with our master data in VFP.

Whil Hentzen is editor of FoxTalk. whil@hentzenwerke.com