Multi-User and Data Buffering Issues

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
World Wide Web: www.stonefield.com

Overview

In FoxPro 2.x, developers edited records using **`scatter memvar`**, editing the memory variables, and **`gather memvar`**. The purpose of this indirect editing of fields was to protect the record by buffering it. With Visual FoxPro, data buffering is built in, so fields can be edited directly. This session will discuss how data buffering works and explores strategies for selecting which buffering mechanism to use and how to handle multi-user conflicts.

Introduction

If you’ve spent any time at all working with VFP, one of the things you’ve probably learned is that while you can continue to do things the “old” way if you wish, VFP provides you with a better way to perform the same task. How records are edited in forms is a perfect example of this.

Here’s the “old” way I used to write code to edit a record in a data entry screen:

- The screen has **`get`** objects for memory variables with the same names as the table’s fields (for example, M.CUST_ID and M.NAME).

- When the user positions the table to a particular record (for example, using the “Next” button), use **`scatter memvar`** to transfer the record to the memory variables and **`show gets`** to refresh the values shown on the screen. The user cannot edit the variables (they’re either disabled or have **`when`** clauses that evaluate to .F.) because the user is currently in “view” mode.

- When the user chooses the “Edit” button, try to lock the record; display an appropriate message if we can’t. Check the value of each field against its memory variable—if they don’t match, another user must have edited and saved the record since we first displayed it. In that case, display an appropriate message and use **`scatter memvar`** and **`show gets`** again so the user sees the current contents of the record.

- If the fields and memory variables match, either enable the **`get`** objects or make their **`when`** clauses evaluate to .T. so the user can edit the variables.

- When the user chooses the “Save” button, do some validation to ensure everything was entered according to our rules, then **`gather memvar`** to update the record from the memory variables and unlock the record. Disable the **`get`** objects or make their **`when`** clauses evaluate to .F. so the user is once again in “view” mode.

Notice in this scheme we don’t do a direct **`read`** against the record. Instead, we allow the user to edit memory variables and only write those memory variables back to the record if everything went OK. The reason for using this method is it protects the table; we don’t allow any data to be stored unless it passes all the rules. Also notice the record is locked while the user is editing the memory variables. This prevents another user from editing the same record at the same time. It does, however, suffer from the “out to lunch” syndrome—if the user starts the edit, then goes for lunch, the record stays locked, unavailable to other users for editing.

This isn’t the only way to edit records, of course. You could do the lock just before saving the record instead of when the edit mode starts. This minimizes the time the record is locked, allowing other users access to it. This has its own drawback, though: if the user edits the variables and clicks on “Save”, what happens if some other user edited the record in the meantime? Do you overwrite their changes? Do you prevent the record from being saved? This is a design issue you must handle on a case-by-case basis.

The whole purpose of all this effort is to protect the data. If you were writing an application that only you would ever use, you’d probably make it a lot simpler—just **`read`** against the fields in the record directly. This makes the screen act like the “form” equivalent of a **`browse`**, since everything you type goes directly into the record. However, since we can’t trust those pesky users to know what they can and can’t enter, we have to protect the data by building a “firewall” between the user and the table. Creating this “firewall” in FoxPro 2.x took a significant amount of coding.

VFP provides a built-in “firewall” mechanism that gives us the best of both worlds: direct **`read`** against a record while only permitting the data to be written after it passes all the tests. This mechanism is *buffering*.

Buffering

Using memory variables to hold the contents of a record can be considered like creating a buffer of data. The data is transferred from the record to the "buffer" by using **`scatter memvar`** and from the "buffer" to the record with **`gather memvar`**.

Not only can VFP do this type of single record buffering (called *record* or *row buffering*) automatically, it also supports another type of buffering (called *table buffering*) in which multiple records are accessed through a buffer.

Record buffering is normally used when you want to access or update single records at a time. This is common in data entry mechanisms like that described above: the user can display or edit a single record in the form. Table buffering would be the choice for updating several records at a time. A common example of this is an invoice header-detail screen. By using table buffering for the invoice detail table, you can allow the user to edit detail lines as long as they wish, and then save or cancel all the detail records at once.

In addition to two buffering mechanisms, there are two locking mechanisms. The "old" way I described earlier can be considered to be a *pessimistic* locking scheme—the record is locked as soon as the user chooses "Edit", and stays locked until they choose "Save". This ensures no one else can change the record while this user is doing so, which may or may not be a good thing, depending on your application. The other method I described earlier is an *optimistic* locking mechanism—the record is only locked for the brief amount of time it takes to write the record, and is immediately unlocked. This maximizes the availability of the record (this is also known as maximizing *concurrency*) but means we have to handle conflicts that occur if two users edit the record at the same time. As we'll see in a moment, this is actually easy to do in VFP, so optimistic buffering will probably be the mechanism of choice for most applications.

Since records can be automatically buffered, there's no longer a need to use the "manual buffer" mechanism. In other words, now we can **`read`** directly against the fields in the record and not worry about maintaining memory variables for each one. To save the changes, we simply tell VFP to write the buffer to the table, and to cancel the changes, we tell it not to. We'll see how to do that in a moment.

VFP implements buffering by creating a "cursor" whenever a table is opened. The cursor is used to define properties for the table. In the case of local tables, the only property for the cursor is how buffering is performed; views and remote tables have additional properties beyond the scope of this session. These properties are set using the **`cursorsetprop()`** function and examined with **`cursorgetprop()`**. We'll see the use of these functions shortly.

Table buffering has an interesting implementation regarding appended records: as records are added to the buffer, they're assigned a negative record number. **`recno()`** returns -1 for the first appended record, -2 for the second, and so on. You can use **`go`** with a negative number to position the buffer to the appropriate appended record. This has an implication for routines handling record numbers—instead of testing for **`between(lnRecno, 1, reccount())`** to ensure lnRecno is a valid record number, you'll now have to test for **`between(lnRecno, 1, reccount()) or lnRecno < 0`**.

Using Buffering

Buffering is turned off by default, so VFP acts just like FoxPro 2.x in terms of how updates are written to a table. To use buffering, you must specifically turn it on. Buffering is available for both free tables and those attached to a database. Buffering requires that you **`set multilocks on`** since by default it too is set off; you'll get an error message if you forget to do this. You can put **`multilocks = on`** in your CONFIG.FPW or use the *Options* function under the *Tools* pad to save this setting as the default.

Buffering is controlled using **`cursorsetprop('Buffering', <n>, <Alias>)`**. You don't have to specify **`<Alias>`** if you're setting buffering for the current table. **`<n>`** is one of the following values depending on the buffering and locking method you wish to use:

Buffering/Locking Method	**<n>**
no buffering	1
record, pessimistic	2
record, optimistic	3
table, pessimistic	4
table, optimistic	5

For example, to enable optimistic record buffering, use **`cursorsetprop('Buffering', 3)`**. To determine what buffering is currently in use for a table, use **`cursorgetprop('Buffering')`**.

To enable buffering in a form, you could specify **`cursorsetprop()`** for each table in the form's Load method, but the preferred approach is to set the form's BufferMode property to either optimistic or pessimistic (the default is "none"). The form will then automatically use table buffering for tables bound to grids and row buffering for all other tables. If you use a DataEnvironment for the form, you can override the form's BufferMode for a particular table by setting its BufferModeOverride property as desired.

While the user is changing the data in the buffered record (they're in the middle of editing the record), you have access to not only the value they've entered into each field, but also the former value of each field and its current value (the value actually on disk). Two new functions, **`oldval()`** and **`curval()`**, were added for this purpose. Here's how you obtain the appropriate values:

To Get:	**Use:**
the value the user entered (the value in the buffer)	**`<fieldname>`** or **`<alias.fieldname>`**
the value before the user changed anything	**`oldval('<fieldname>')`**
the current value in the record	**`curval('<fieldname>')`**

`curval()` and **`oldval()`** can only be used with optimistic buffering.

You may be wondering how the value returned by **`curval()`** would differ from the one returned by **`oldval()`**. It obviously wouldn't if only a single user is running the application. However, on a network and with optimistic locking, it's possible that after the user started editing the record, another user edited the same record and saved their changes. Here's an example:

Bob brings up record #2 in CONTACTS.DBF and clicks on the "Edit" button:

Field	**Value**	**oldval()**	**curval()**
LAST_NAME	**`Jones`**	**`Jones`**	**`Jones`**
FIRST_NAME	**`Bill`**	**`Bill`**	**`Bill`**

Bob changes the first name to Sam but doesn't save the record yet:

Field	Value	oldval()	curval()
LAST_NAME	`Jones`	`Jones`	`Jones`
FIRST_NAME	`Sam`	`Bill`	`Bill`

Mary brings up record #2 in CONTACTS.DBF, clicks on the "Edit" button, changes the first name to Eric, and saves. At Bill's machine:

Field	Value	oldval()	curval()
LAST_NAME	`Jones`	`Jones`	`Jones`
FIRST_NAME	`Sam`	`Bill`	`Eric`

Notice **`FIRST_NAME`**, **`oldval('FIRST_NAME')`**, and **`curval('FIRST_NAME')`** all return different values. By having access to the original value, the buffered value, and the current value for each field in a record, you can:

- determine which fields the user changed by comparing the buffered value to the original value; and

- detect whether other users on a network made changes to the same record after the edit had started by comparing the original value to the current value.

If you don't care about the old and current values but only wish to detect if a field was edited by the user, use **`getfldstate()`**. This new function returns a numeric value indicating if something about the current record has changed. **`getfldstate()`** is called as follows:

```
getfldstate(<FieldName> | <FieldNumber> [, <Alias> | <WorkArea>])
```

and returns one of the following values:

Value	Description
1	No change
2	The field was edited or the deletion status of the record was changed
3	A record was appended but the field was not edited and the deletion status was not changed.
4	A record was appended and the field was edited or the deletion status of the record was changed.

Changing the deletion status means either deleting or recalling the record. Note deleting and then immediately recalling the record will result in a value of 2 or 4 even though there's no net effect to the record.

If you don't specify an alias or workarea, **`getfldstate()`** operates on the current table. Specify 0 for **`<FieldNumber>`** to return the append or deletion status of the current record. If you specify -1 for **`<FieldNumber>`**, the function will return a character string with the first digit representing the table status and one digit for the status of each field.

In the example mentioned earlier, where Bill edits the second field, **`getfldstate(-1)`** would return “112”. The first digit indicates the record was not appended or deleted, the second that the first field was unchanged, and the third that the second field was changed.

Writing a Buffered Record

Continuing with the previous example, now Bill clicks on the “Save” button. How do we tell VFP to write the buffer to the record? With record buffering, the table is updated when you move the record pointer or issue the new **`tableupdate()`** function. With table buffering, moving the record pointer doesn’t update the table (since the whole point of table buffering is that several records are buffered at once), so the usual way is to issue **`tableupdate()`**. It’s best to use **`tableupdate()`** even for record buffering since you have more control over what happens when.

`tableupdate()` returns .T. if the buffer was successfully written to the record. If the record buffer hasn’t changed (the user didn’t edit any fields, add a record, or change the deleted status for the record), **`tableupdate()`** returns .T. but actually does nothing.

`tableupdate()`can take a few optional parameters:

```
tableupdate(<AllRows>, <Forced>, <Alias> | <Workarea>)
```

The first parameter indicates what records to update: .F. tells it to only update the current record, while .T. means update all records (only effective if table buffering is used). If the second parameter is .T., any changes by another user will be overwritten by the current user’s changes. Unless the third parameter is specified, **`tableupdate()`** will update the current table.

How do you cancel the changes the user made? With the memory variable approach, you’d just **`scatter memvar`** again to restore the memory variables to the values stored on disk. With buffering, use the **`tablerevert()`** function to do the same for the buffer.

Handling Errors

Continuing on with the “Bill and Mary” example, the code executed when Bill clicks the “Save” button uses the **`tableupdate()`** function to try to write the buffer to the record. Remember Mary edited the record and saved her changes as Bill was editing the same record. When Bill clicks on “Save”, **`tableupdate()`** will return .F., meaning it didn’t write the buffer. Why?

VFP will not write the buffer to the record under the following conditions:

- Another user changed and saved the record while this user was editing it (as happened in this example). VFP automatically compares **`oldval()`** and **`curval()`** for each field. If it detects any differences, we have a conflict.

- The user entered a duplicate primary or candidate key value.

- A field or table rule was violated, or a field that doesn’t support null values is null.

- A trigger failed.

- Another user has the record locked. This can be minimized by avoiding manually locking records with **`rlock()`** and using the same buffer locking mechanism for a table in all forms and programs that access it.

- Another user deleted the record.

You must decide what to do when **`tableupdate()`** fails. Also, if your application allows the user to click on the "Next" or "Previous" buttons while editing a record and those functions don't issue a **`tableupdate()`**, you must handle the error that will occur when the automatic save is attempted. In both of these cases, the proper place to handle this is in an error trapping routine.

Error handling has been improved in VFP. The old way to set an error trap (which you can still use in VFP) is to use the **`on error`** command to specify a procedure to execute when an error occurs. This error routine would typically look at **`error()`** and **`message()`** to determine what happened, and take the appropriate action.

VFP now provides an automatic error handling mechanism: the *Error* method. If an Error method exists for an object or form, it will automatically be executed when an error occurs without having to manually set the trap. **`aerror()`** is a new function that helps in figuring out what went wrong. You pass it an array name and it creates or updates the array with the following elements:

Element	Type	Description
1	Numeric	The error number (same as **`error()`**).
2	Character	The error message (same as **`message()`**).
3	Character	The error parameter (for example, a field name) if the error has one (same as **`sys(2018)`**) or .NULL. if not.
4	Numeric or Character	The work area in which the error occurred if appropriate, .NULL. otherwise.
5	Numeric or Character	The trigger that failed (1 for insert, 2 for update, or 3 for delete) if a trigger failed (error 1539), or .NULL. if not.
6	Numeric or Character	.NULL. (used for OLE and ODBC errors).
7	Numeric	.NULL. (used for OLE errors).

For example, **`aerror(laERROR)`** will create or update an array called laERROR.

Here are the common errors that may occur when VFP attempts to write the buffer to the table:

Error #	Error Message	Comment
109	Record is in use by another	
1539	Trigger failed	Check element 5 to determine which trigger failed.
1581	Field does not accept null values	Check element 3 to determine which field was involved.
1582	Field validation rule is violated	Check element 3 to determine which field was involved.
1583	Record validation rule is violated	
1585	Record has been modified by another	
1884	Uniqueness of index violated	Check element 3 to determine which tag was involved.

Handling most of these errors is straightforward: tell the user the problem and leave them in edit mode to correct the problem or cancel. For error #1585 (record has been modified by another), there are several ways you could handle the error:

- You could tell them someone else modified the record and then cancel their edits using **`tablerevert()`**. I suspect most users wouldn't be too thrilled by this approach <g>.

- You can force the update of the record by using **`tableupdate(.F., .T.)`**. This causes the other user's changes to be overwritten by the current user's. This user might be happy, but the other user probably won't be.

- You can display the changes the other user made to the record in another copy of the same form (easy to do with VFP's ability to create multiple instances of the same form). The user can then decide whether the changes the other user made should be kept or not, and you can either use **`tableupdate(.F., .T.)`** to force the update or **`tablerevert()`** to cancel.

- A more intelligent scheme involves determining if we have a "real" conflict or not. By "real", I mean: did both users change the same field or not. If the fields they updated are different, we could tell VFP to just update the field this user changed, leaving the other user's changes intact. An example might be in an order processing system. One user may have edited the description of a product while another user entered an order for the product, thereby decreasing the quantity on hand. These changes aren't mutually exclusive—if we make our table update less granular (that is, we don't update an entire record at a time, just the fields we changed), we can satisfy both users.

  Here's the logic of how that works:

  - Find a field where **`oldval()`** is different than **`curval()`**, meaning this field was edited by another user. If the field's buffered value is the same as **`oldval()`**, this user didn't change the field, so we can prevent overwriting its new value by setting the buffered value to **`curval()`**.

  - Find a field where the buffered value is different than **`oldval()`**. This is a field this user edited. If **`oldval()`** equals **`curval()`**, the other user didn't change this field, so we can safely overwrite it.

  - If we find a field where the buffered value is different than **`oldval()`** but the same as **`curval()`**, both users made the same change. While this may seem unlikely, one example would be when someone sends a change of address notice to a company and somehow two users decide to update the record at the same time. Since the changes were identical, we might be able to overwrite the field. However, in the case of a quantity being updated by the same amount (for example, two orders for the same quantity were entered at the same time), you wouldn't want to overwrite the field, and would consider this to be a "real" conflict.

- If we have a case where the buffered value of a field is different than both **`oldval()`** and **`curval()`**, and **`oldval()`** and **`curval()`** aren't the same either, both users changed the same field but to different values. In this case, we have a "real" conflict. You have to decide how to handle the conflict.

  In the case of inventory quantity on hand or account balances, one possibility is to apply the same change the other user made to the buffered value. For example, if **`oldval()`** is 10 and **`curval()`** is 20, the other user increased the amount by 10. If the buffered value is 5, this user is decreasing the amount by 5. The new buffered value should therefore be **`value + curval() - oldval()`**, or 15.

  In the case of Date fields, business rules and common sense might help. For example, in a patient scheduling program with a field containing the date of a patient's next visit, the earlier of the two dates in conflict is probably the correct one to use, unless it's prior to the current date, in which case the later date is the correct one.

  Other types of fields, especially Character and Memo fields, often can't be resolved without asking the user to make a decision about overwriting the other user's changes or abandoning their own. Allowing the user to see the other user's changes (as mentioned earlier) can help them make this decision.

Here's some code that will do this type of conflict resolution (this code assumes we've already determined the problem is error #1585, the record had been modified by another user):

```
* Check every field to see which ones have a conflict.

llConflict = .F.
for lnI = 1 to fcount()
    lcField      = field(lnI)
    llOtherUser  = oldval(lcField)    <> curval(lcField)
    llThisUser   = evaluate(lcField) <> oldval(lcField)
    llSameChange = evaluate(lcField) == curval(lcField)
    do case

* Another user edited this field but this user didn't, so grab the
* new value.

       case llOtherUser and not llThisUser
          replace (lcField) with curval(lcField)

* Another user didn't edit this field, or they both made the same
* change, so we don't need to do anything.

       case not llOtherUser or llSameChange

* Uh-oh, both users changed this field, but to different values.

       otherwise
          llConflict = .T.
    endcase
next lnI

* If we have a conflict, handle it.
```

```
if llConflict
    lnChoice = messagebox('Another user also changed this ' + ;
       'record. Do you want to overwrite their changes (Yes), ' + ;
       'not overwrite but see their changes (No), or cancel ' + ;
       'your changes (Cancel)?', 3 + 16, 'Problem Saving Record!')
    do case

* Overwrite their changes.

       case lnChoice = 6
          = tableupdate(.F., .T.)

* See the changes: bring up another instance of the form.

       case lnChoice = 7
          do form MYFORM name oName

* Cancel the changes.

       otherwise
          = tablerevert()
    endcase

* No conflict, so force the update.

else
    = tableupdate(.F., .T.)
endif llConflict
```

Writing a Buffered Table

As we saw earlier, **`tableupdate(.T.)`** attempts to write all records in a table buffer to disk. As with the row buffered version, it will return .F. if it couldn't update a record because another user changed it (among other reasons).

The error trapping routine we saw earlier works fine for row buffering, since we're only concerned with a single record at a time. However, with table buffering, we have to look at each record one at a time. Since we might have a mixture of modified and unmodified records in the buffer, how do we know which records will be updated? To make matters more complicated, if **`tableupdate(.T.)`** fails, we don't know which record it failed on; some records may have been saved and there could be more than one record in conflict.

The new **`getnextmodified()`** function will tell us exactly what we need to know: the record number for the next modified record. If it returns 0, there are no more modified records in the buffer. This function accepts two parameters: the first is the record number after which to search for the next modified records, and the second is the alias or workarea to search in. Initially, you should pass 0 as the first parameter so **`getnextmodified()`** finds the first modified record. To find the next one, pass the record number for the current record.

Here's an example of the earlier conflict management routine, modified to handle table buffered changes when **`tableupdate(.T.)`** fails:

```
* Find the first modified record, then process each one we find.

lnChanged = getnextmodified(0)
do while lnChanged <> 0
```

```
* Move to the record and try to lock it.

   go lnChanged
   if rlock()

* Check every field to see which ones have a conflict.

      llConflict = .F.
      for lnI = 1 to fcount()
         lcField      = field(lnI)
         llOtherUser  = oldval(lcField)   <> curval(lcField)
         llThisUser   = evaluate(lcField) <> oldval(lcField)
         llSameChange = evaluate(lcField) == curval(lcField)
         do case

* Another user edited this field but this user didn't, so grab the
* new value.

            case llOtherUser and not llThisUser
               replace (lcField) with curval(lcField)

* Another user didn't edit this field, or they both made the same
* change, so we don't need to do anything.

            case not llOtherUser or llSameChange

* Uh-oh, both users changed this field, but to different values.

            otherwise
               llConflict = .T.
         endcase
      next lnI

* If we have a conflict, handle it. If we don't, unlike the row buffering
* case, we don't do anything now since all records will be written later.

      if llConflict
         lnChoice = messagebox('Another user also changed ' + ;
            'record ' + ltrim(str(lnChanged)) + '. Do you want to ' + ;
            'overwrite their changes (Yes), not overwrite but see ' + ;
            'their changes (No), or cancel your changes (Cancel)?', 3 + 16, ;
            'Problem Saving Record!')
         do case

* Overwrite their changes: we don't actually need to do anything because we'll
* do them all later (this case is only here for clarity).

            case lnChoice = 6

* See the changes: bring up another instance of the form.

            case lnChoice = 7
               do form MYFORM name oName

* Cancel the changes in this record only.

            otherwise
               = tablerevert()
               unlock record lnChanged
         endcase
      endif llConflict

* We couldn't lock the record, so cancel the changes to this record only.

   else
      = messagebox("Sorry, we couldn't save record #" + ltrim(str(lnChanged)))
      = tablerevert()
      unlock record lnChanged
   endif rlock()

* Find the next modified record and process it.

   lnChanged = getnextmodified(lnChanged)
enddo while lnChanged <> 0

* Since we reverted any changes where we found a conflict and the user wanted
* to cancel their own changes, let's force the remainder of the updates.

= tableupdate(.T., .T.)
```

If a table has changes in a table buffer that haven't been written out to disk and you attempt to close the table or change the buffering mode, you'll get an error (#1545): "Table buffer for alias <Alias> contains uncommitted changes".

Transactions

As we've seen, table buffering is a convenient way to buffer a number of changes to a table and then write or abandon those changes all at once. However, there's one flaw with this approach—what happens if one of the records in the buffer is locked or has been edited by another user? In this case, **`tableupdate(.T.)`** will return .F. and the error trapping routine can be called. The problem: some records were saved and some weren't. Now you have a fairly complicated mess on your hands if you need to back out those changes already made.

Here's an example of such a problem: you go to the bank to transfer money from your savings account to your checking account. The account update program reduces your savings account balance by the appropriate amount, then tries to increase your checking account balance by the same amount. The program might look something like this:

```
seek M.ACCOUNT1
replace BALANCE with BALANCE - M.AMOUNT
seek M.ACCOUNT2
replace BALANCE with BALANCE + M.AMOUNT
llSuccess = tableupdate(.T.)
if not llSuccess
   do ERROR_ROUTINE
endif not llSuccess
```

In the meantime, an automated check clearing program has been processing your checking account, and has reduced its balance by the total of several checks. The program detects the conflict and decides to abandon the update by issuing **`tablerevert(.T.)`**. However, since the savings account was successfully updated, its change is no longer in the buffer, and therefore it stays changed. Now the bank has an "out-of-balance" situation that will be difficult to track down, and one very angry customer when you get your bank statement at the end of the month.

Fortunately, VFP provides a mechanism that can resolve this problem: the *transaction*. A transaction is a specific group of changes that must either all be made at once or all abandoned. A transaction is started with the **`begin`**

`transaction` command. Any table changes after this command has been issued, even those made with **`tableupdate()`**, are not written to the disk until an **`end transaction`** command is encountered. Think of a transaction as a "buffer's buffer". The transaction is held until you determine all changes could be made successfully and issue an **`end transaction`**. If the program crashes or the computer is rebooted before **`end transaction`** is encountered, or if your program issues a **`rollback`** command because one of the changes couldn't be made successfully, none of the changes are actually written to disk.

Let's look at the bank update example but this time use a transaction as a "wrapper" for the update:

```
begin transaction
seek M.ACCOUNT1
replace BALANCE with BALANCE - M.AMOUNT
seek M.ACCOUNT2
replace BALANCE with BALANCE + M.AMOUNT
llSuccess = tableupdate(.T.)
if llSuccess
   end transaction
else
   rollback
endif llSuccess
```

If the first account balance was changed but the second couldn't be successfully, llSuccess will be .F., and the **`rollback`** command will prevent the first change from being written to disk. If everything went OK, **`end transaction`** will write out both changes at once.

Here are some other concepts regarding transactions:

- Transactions can only be used with tables attached to databases; free tables need not apply.

- Transactions apply to memo (FPT) and index (CDX) files as well as to the DBF.

- Commands and functions that alter the database, the table, or the table's indexes cannot be used during a transaction. For example, issuing **`alter table`**, **`delete tag`**, **`index on`**, **`tablerevert()`**, or **`close databases`** during a transaction will generate an error. See the VFP documentation for a complete list of restricted commands.

- You can nest transactions up to five levels deep ("The good news: you no longer have just five **`read`** levels. The bad news: ..." <g>). When an inner level transaction is completed, its changes are added to the cache of changes for the next transaction level rather than being written to disk. Only when the final **`end transaction`** is issued are all the changes written out. You can use the **`txnlevel()`** function to determine the current transaction level.

- Unlike other VFP structured programming constructs (such as **`for/next`** or **`scan/endscan`**), **`begin transaction`**, **`end transaction`**, and **`rollback`** don't have to be located in the same program. You could, for example, have a common routine for starting transactions and another one for ending them. Transactions should be kept as short as possible, however, since any records being updated during a transaction are completely unavailable to other users, even just for reading.

- Records automatically locked by VFP during a transaction are automatically unlocked when the transaction is complete. Any locks you set manually are not automatically unlocked; you are responsible for unlocking those records yourself. If you use **`unlock`** during a transaction, the record actually stays locked until the transaction is done, at which time all specified records are unlocked.

- Although transactions give you as much protection as they can, it's still possible a hardware failure or server crash during the **`end transaction`** disk writes could cause data to be lost.

- Transactions only apply to local tables. Transactions for remote tables are controlled using the **`sqlsetprop()`**, **`sqlcommit()`**, and **`sqlrollback()`** commands. Transaction processing with remote tables is beyond the scope of this session.

Here's another look at the "save" routine and the error trapping routine (in the error routine, code in the **`do while`** loop isn't shown since it's the same as the previous version):

```
begin transaction
if tableupdate(.T.)
   end transaction
else
   rollback
   do ERROR_ROUTINE
endif tableupdate(.T.)

procedure ERROR_ROUTINE

* Do setup stuff here, including checking what happened. If we found error
* #1585, do the following code.

lnChanged = getnextmodified(0)
do while lnChanged <> 0

...

enddo while lnChanged <> 0

* Since we reverted any changes where we found a conflict and the user wanted
* to cancel their own changes, let's force the remainder of the updates and
* then unlock all the records we manually locked.

begin transaction
if tableupdate(.T., .T.)
   end transaction

* Some other error occurred now, so rollback the changes and display an
* appropriate error message (you could also try to handle it here if you
* wish).

else
   = aerror(laError)
   rollback
   = messagebox('Error #' + ltrim(str(laError[1])) + ': ' + laError[2] + ;
      ' occurred while saving.')
endif tableupdate(.T., .T.)
```

Other Issues

As we saw earlier, **`getfldstate()`** can be used to determine if anything in the current record has changed. This allows you to create forms that no longer need a "edit" mode; the data in the form is always available for editing. Each field's InteractiveChange event could enable the "Save" and "Cancel" buttons only if the user actually changed something using code similar to:

```
if getfldstate(-1) = replicate('1', fcount() + 1)
   * disable the buttons, since nothing's changed
```

```
else
   * enable the buttons
endif getfldstate(-1) = replicate('1', fcount() + 1)
```

To create a form without an edit mode, you also need to have code that handles the case when the user closes the window, exits the application, or moves to another record (if row buffering is used). The QueryUnload event can help with the first two; this event occurs when the user clicks in the form's close box or quits the application. You could put code in this event that saves the record before closing the form. In the case of moving the record pointer, you'd modify your record navigation routines (first, last, next, previous, find, etc.) to check if any fields were changed (and if so, save the record) before moving the record pointer. You would likely have one common method in the form that does all this checking and saving, and call it whenever required.

A related issue to this is that **`getfldstate()`** might erroneously indicate that nothing has changed when in fact the user changed the value in a field. This can happen if you provide a menu choice or tool bar button to save the record or move the record pointer. VFP only copies the value in a control (such as a Textbox) to the record buffer when the control loses focus. If the user changes the value in a field and then clicks the Next button in the tool bar, the Textbox doesn't lose focus (since tool bars never receive focus), so the new value isn't copied to the record buffer and VFP doesn't know the data has changed. The solution to this problem is to force the current control's value to be copied to the buffer before using **`getfldstate()`** using code similar to the following in the tool bar button's Click method:

```
with _screen.ActiveForm.ActiveControl
   if type('.ControlSource') <> 'U' and not empty(.ControlSource) and ;
      not evaluate(.ControlSource) == .Value
      replace (.ControlSource) with .Value
   endif type('.ControlSource') <> 'U' ...
endwith
```