Integrating Word's Spellcheck with Your VFP Application

Whil Hentzen

I recently received a request to "integrate spell check into our application". When I asked the customer for details past this vague, one phrase description, they simply responded, "Some of our users are pretty bad spellers. So we'd like them to be able to check what they type before they do things like send letters to people outside the organization." With that set of enhanced specifications now in hand, I set off to see what options I had.

This article describes both the journey I went through, choosing features and making choices between alternatives, as well as the details of the code I ended up with.

When adding a standard but potentially complex feature like spellcheck, the first thing we considered was the 'buy or build' fork in the road. Considering my customer's reticence at adding ongoing licensing costs to their application, I first looked at the 'build' option.

The second thought that we all have, as the mantra "Was that written in FoxPro? No, but it could be!" is never far from our minds, is to roll one's own. This venture, like many things, is one of those "How hard could it be?" projects that reminds me of The Devil and Simon Flagg story

```
http://simonsingh.net/books/fermats-last-
theorem/wacky-fermat-stuff/devilish-short-
story/
```

where there's never really an end. The problem in this quest is two-fold.

First, there's the search for an appropriate (and free) spelling dictionary. You can't just grab the Webster or Oxford dictionary off the Web, as you'd still have to parse the words themselves from the pronunciation and definitions of each word. But as much a pain as that would be, that's the easy part.

The hard part is coming up with good suggestions for words that aren't found in the dictionary.

Let's suppose the user typed the word 'piar'. It's relatively easy to come up with suggestions that involve simply switching the letters around, so that 'pair' is offered as a potential correction. But what about similar words? A good mechanism should also provide 'pliar' and 'pier', and few developers have the in-depth knowledge of heuristics to do that type of work. It's simply not that easy.

So after just a few minutes, I decided that I'd let someone else do the work. Thus, my next thought was one of a variety of third-party solutions, such as an Active-X control or a third-party spell-checker such as FoxSpell. Those that were still supported all required royalty or licensing fees, some in the hundreds of dollars per user. That wasn't going to fly with my customer.

The first 'freebie' option was native Visual FoxPro. Earlier versions came with a spellchecker application, spellchk.app, but it's not included with VFP 9. And for those of you who are thinking "What if I just found a copy of an earlier version of VFP and used it from there?", well, no such luck. Spellchk.app was never allowed to be distributable with your applications due to licensing restrictions of the dictionary that came with it.

There were also a number of 100% VFP-code solutions available on the Web for free, some more robust than others, but all relying on third-party dictionaries of one sort or another as well as home-grown heuristics for determining matches for a misspelled word. Support for these ranged from minimal to non-existent, and many had been abandoned years ago. And, worse, for lack of a better term, the solutions that were still available all seemed (sorry for the technical term) 'icky'.

The last option I considered was using office automation to hook into Microsoft Word's spell

checking engine, letting it do all the work of detecting misspellings and suggesting corrections, and simply returning the results to the user.

This third avenue seemed the most promising - since this particular customer was a 100% Microsoft shop, it was a certainty that each user would have a recent version of Microsoft Word installed on their machine. Why not simply take advantage of Word's native spell-checking capability? Leave the dictionary and heuristics to them, and spend my time working on the interface for my customer's users.

Tamar Granor wrote an excellent article describing the technical details involved in accessing Microsoft Word's spellchecking mechanism from VFP.

```
http://www.tomorrowssolutionsllc.com/Articles/
Checking%20Spelling%20in%20VFP.PDF
```

I won't repeat the details of her article, rather, this article describes how I incorporated Tamar's engine into a user interface and how I connected that mechanism with my customers' application, a particularly challenging task given a lack of specifics from my customer.

General Requirements

Although the challenge was initially presented simply, subsequent discussions revealed a few specifics.

First, the spellcheck was not going to be universally available; only selected fields on certain forms would have spellchecking capability available.

Second, as is common for corporations with industry-specific jargon, the user wanted to be able to switch between a standard dictionary and a custom dictionary that contained pre-defined words for the company. In this case, the jargon was primarily medical terms.

Third, the user wanted to be able to add their own words as they came across them.

Finally, the user wanted to be able to specify certain settings, such as ignoring words that were all caps (in this case, those would be abbreviations and acronyms), for the entire spellcheck implementation.

The Engine

Tamar's engine consists of several pieces:

1. Grabbing a string to spellcheck
2. Checking to see if Word was already running, and instantiating Word if it wasn't
3. Checking each word in the string
4. Storing each misspelled word, together with Word's suggestions, to a data structure for presentation to the user

My wrapper around Tamar's engine consisted of two pieces. The first was to obtain a string to check from the application. The second was the presentation of the resulting choices to the user, allowing them to make a choice, and stuffing their choice back into the calling application.

Now, where do these engine pieces go? What does the UI consist of? Let's look.

The User Interface

While designing the UI, I came across several issues for which decisions had to be made.

1. How is the spellcheck launched?

Either it is started up when the form is opened, and then lies in wait and dynamically checks as the user types, or the user explicitly launches the spellcheck by, say, right-clicking on a control that contains a string to be checked.

2. What to check?

Either spellcheck can check one user-specified control on the form or it can do all controls. Interestingly, this is choice can be independent of how the spellcheck is launched. Obviously, if the user explicitly launches the spellcheck on a certain control, that's the control being checked.

On the other hand, if the spellchecker is the 'lies in wait and checks everything' variety, each control can be flagged to be checked or not via a property setting.

3. What to check in a control?

Either the spellchecker will examine every word in the control, or just a user-selected word in the control.

4. If user launched, what action is performed?

When the user right-clicks on a control, what happens? The spellcheck function might be launched immediately, checking the string and returning suggestions if misspellings were found. Or it might display a context menu that gives the user choices, such as

- Spellcheck using standard dictionary
- Spellcheck using custom dictionary
- Configure spellcheck settings

The third menu option would be where the user could change parameters that controlled how the spellcheck worked, either locally or globally.

5. How to display potential misspellings and suggested corrections?

The universal technique used in applications like word processors and browsers is to flag each word not found in the dictionary with a red squiggly underline. However, this is VFP, where this type of red underline mechanism isn't native to the base control set. Thus, a set of custom text and edit controls that could be configured to display their contents in such a manner would be required. Due to the difficulty in locating such a toolset that is current and actively supported as well as avoiding licensing requirements, that's simply not going to happen.

Thus, an alternate interface would be needed, and the choices varied, depending on what was being checked.

Ultimately, there were two choices available. The first was a context menu that displayed potential misspellings for a single word, just like is commonly done in other applications. The other was to launch a form that displayed all misspelled words in a control together with their suggested corrections.

Summing up

That's a lot of alternatives.

This article isn't intended to exhaustively catalog every possible combination of alternatives and demonstrate implementations. Rather, I'll discuss the choices my customer made and how they were implemented.

For this application, we opted to

1. Explicitly launch the spellchecker
2. On a single field
3. Check all the words in the control
4. If words in the string are not found in the dictionary, a 'Spell Check Choices' form is launched
5. This form contains two columns; the first listed the words that were identified as misspelled, the second listed the suggested corrections.

Given these choices, here's how I incorporated the engine into the interface.

Implementation

First, we've got a form with one or more fields to be spell checked, and the user will right-click on the controls to be checked, one by one. If, for each control, there are one or more words in the control's string not found in the dictionary, a form will be displayed that lists those words and their suggested corrections.

Under the hood, I added a property, lAllowSpellcheck, to the base class of each control (text and edit) that might be checked. The instance of a field to be checked will have the property set to true. Then the right-click method will check the property and if true, run the spell check routine.

```
if this.lAllowSpellcheck
  do form xspellcheckchoices with ;
    thisform, this
endif
```

This flag allows the spellchecking to be turned on or off for certain fields as desired.

The call sends two parms to the Spell Check Choices form. The first is a reference to the calling form and the second is a reference to the control being checked. The second is used to grab the value of the string to check, and then together with the first, is used to create a fully qualified reference to the control if the value in the control has to be updated with a correction.

The init() of the spellcheck data form determines if there are misspelled words and returns true or false as appropriate. If false (no misspelled words are found in the string), the form isn't instantiated. Let's look in more detail.

First, the init() receives the parms and assigns them to form properties, and does some interface nudging:

```
lpara loForm, loObject

thisform.oForm = loForm
* object might be txt or edt
thisform.oObject = loObject

* clear the message from the caller
wait clear
* update the Spell Check Choices form caption
* to reflect what's being checked
thisform.Caption = transform(loForm.name) ;
  + " - " + transform(loObject.name)

* position the Spell Check Choices form below
* the control being checked so that the user
* can see changes being made

* add 10 for just a nudge
* add 23 for the title bar height
this.Top = loForm.top+loObject.top ;
  +loObject.height +23+10
this.left = loForm.left+loObject.left+10
```

Now let's look at the pieces of the engine.

1. Grab a string to spellcheck

The value of the control is determined.

```
* grab the string to check
local lcStringToCheck
lcStringToCheck = loObject.value
if empty(lcStringToCheck)
  return .f.
endif
```

Technically, this could be performed earlier, so as to run less code before deciding whether or not to instantiate the form, but the difference is

trivial, and it's easier to keep all the spell check code together.

2. Check to see if Word was already running, and instantiating Word if it wasn't

Since Word can take a bit to start up, even behind the scenes, a Wait Window is a good idea. Then clear it once started.

```
try
  * Word is running
  thisform.oWord = ;
    getobject(,"Word.Application")
catch
  * Word is not running, start it up
  wait window nowait ;
    'Loading spellchecker in background...'
  thisform.oWord = ;
    createobject("Word.Application")
endtry
wait clear
```

3. Spellcheck each word in the string

Once Word is running, it's time to make the magic happen. The CheckSpellingOfString() method is called next. If this method returns true (the words are all correct), the init() returns false and the Spell Check Choices form isn't instantiated.

```
if thisform.CheckSpellingOfString( ;
  lcStringToCheck)
  * lcStringToCheck was clean
  return .f.
else
```

If, on the other hand, there are misspelled words, the CheckSpellingOfString() fills fills a form array property with the results and stuffs the number of misspelled words in the form property iSuggestionCount.

Then the init() fills the listbox from the form array. Before we continue with the user's actions, let's take a quick look at the actual method, CheckSpellingOfString(), that does the spell checking.

4. Store each misspelled word, together with Word's suggestions, to a data structure for presentation to the user

This method is comes straight from Tamar's article, modified just slightly to handle the needs of interfacing with the Spell Check Choices form.

```
* returns .t. if words are all good
lparameters lcStringToCheck
assert vartype(lcStringToCheck) = 'C' ;
  message "SpellCheck: First parameter ;
  (cString) must be character"
if vartype(lcStringToCheck) <> 'C'
  return .f.
endif

local lReturn, iWord
local oSuggestions as Word.SpellingSuggestions
local oSuggestion as Word.SpellingSuggestion
dimension this.aSuggestions[1]
thisform.aSuggestions[1] = ""
thisform.iSuggestioncount = 0

if empty(lcStringToCheck)
  lReturn = .t.
else
  with thisform.oWord
    .Documents.Add()
    lReturn = .T.
    thisform.iNumOfWordsInStringToCheck ;
      = getwordcount(lcStringToCheck)
    thisform.iNumMisspelledWords = 0
    for iWord = 1 to;
        thisform.iNumOfWordsInStringToCheck
      cWord = getwordnum( lcStringToCheck,;
        iWord )
      oSuggestions =;
        .GetSpellingSuggestions( cWord )
      if oSuggestions.Count <> 0
        thisform.iNumMisspelledWords =;
          thisform.iNumMisspelledWords + 1
        lReturn = .F.
        * Parse the list, put into the array
        for each oSuggestion IN oSuggestions
          thisform.iSuggestionCount = ;
            Thisform.iSuggestionCount + 1
          dime This.aSuggestions[ ;
            This.iSuggestionCount, 3]
          thisform.aSuggestions[ ;
            thisform.iSuggestionCount, 1] = ;
            iWord
          thisform.aSuggestions[ ;
            thisform.iSuggestionCount, 2] = ;
            cWord
          thisform.aSuggestions[ ;
            thisform.iSuggestionCount, 3] = ;
            oSuggestion.Name
        endfor
      endif
    endfor
  endwith
endif
return lReturn
```

This routine is pretty straightforward, taking the string passed to it, splitting it into words via VFP's GetWordCount() function, and then using Word's GetSpellingSuggestions() method to stuff suggestions into an object. If there are suggestions (meaning the word was determined to be misspelled), the list is parsed into an array property of the form that is used to populate the listbox. Finally, either true or false is returned, depending on if a word was misspelled.

5. Present choices to user

If CheckSpellingOfString() returns false, the ELSE segment in the init() of the Spell Check Choices form is fired. Code in this segment fills the listbox in the form with the misspelled words and their suggested corrections from the array property.

```
else
  * fill the listbox with two columns
  * first has misspelled word
  * second has suggested corrections
  if thisform.iSuggestionCount > 0
    dimension thisform.lst.aItems[;
      thisform.iSuggestionCount,2]
    for li = 1 to thisform.iSuggestionCount
```

```
      thisform.lst.aItems[li,1] = ;
        transform( ;
        thisform.aSuggestions[li,2]) + ' '
      thisform.lst.aItems[li,2] = ;
        transform(;
        thisform.aSuggestions[li,3])
    next
    thisform.lst.requery()
  endif
endif
```

This concludes the init(). The form is then displayed to the user, with the list of potential misspelled words and the suggested corrections. The user will execute an action in the UI to replace the misspelled word with their choice.

User Interaction

Now back to the user interface. As mentioned earlier, double-clicking on a choice in the listbox will stuff the control in the calling form with the selected value:

```
if this.ListIndex = 0
  messagebox("Please select a row in the
    list first.")
else
  thisform.iHowManyHaveBeenClicked = ;
    thisform.iHowManyHaveBeenClicked + 1

  local lcNaObject, lcWordToFind, ;
    lcWordToReplaceItWith, lcStrToExecute
  lcNaObject = thisform.oForm.name + '.' ;
    + thisform.oObject.name + '.value'
  lcWordToFind =;
    alltrim(this.aItems[this.ListIndex,1])
  lcWordToReplaceItWith =;
    alltrim(this.aItems[this.ListIndex,2])

  lcStrToExecute = lcNaObject ;
    + '=strtran("' + &lcNaObject + '", "' ;
    + lcWordToFind + '", "' ;
    + lcWordToReplaceItWith + '")'
  &lcStrToExecute

  if thisform.iHowManyHaveBeenClicked ;
    = Thisform.iNumMisspelledWords
    * automatically close the form once we've
    * taken care of every word
    thisform.Release()
  else
    * don't close yet
  endif
endif
```

The iHowManyHaveBeenClicked form property keeps track of how many misspelled words have been processed. Once the last word has been double-clicked on, the form automatically closes by itself.

Once the user is done with choosing replacements, they can hit escape to close the form. In the form's KeyPress method:

```
lpara nKeyCode, nShiftAltCtrl
if nKeyCode = 27
  * escape, to close the form when done
  thisform.Release()
endif
dodefault()
```

Returned to the calling form, the original control still has focus, and the contents have been corrected as the user specified.

The source code for this implementation included in the downloaded consists of

- base.vcx - a class library for the form and the controls,
- spellcheckdemo.scx - a form that contains the controls with the misspelled words, and
- spellcheckchoices.scx - the form that displays the misspellings and suggested corrections.

Note that the source code included with this article does not match exactly with the source code snippets in this first half of this article. The ERs discussed in the second half of the article have been incorporated in the source, so the source demonstrates the final product. The code for the ERs discussed in this article are marked with "ER 4" type annotations.

To run the demo,

```
do form SpellCheckDemo
```

The form will display. The demo form has a default value inserted into the textbox with a pair of errors as shown in Figure 1.

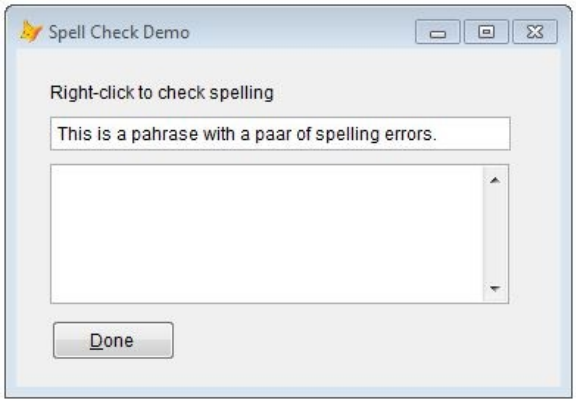

Figure 1. The Spell Check Demo form.

Right-clicking anywhere in the textbox will call the Spell Check Choices dialog, as shown in Figure 2.

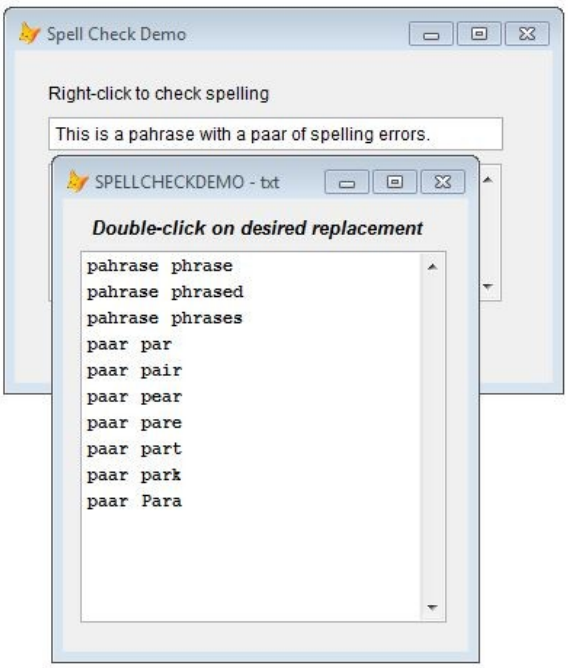

Figure 2. The Spell Check Choices dialog opened when spelling errors are detected.

Double-clicking on a row in the list will substitute the suggested replacement for the misspelled word in the source field, as shown in Figure 3.

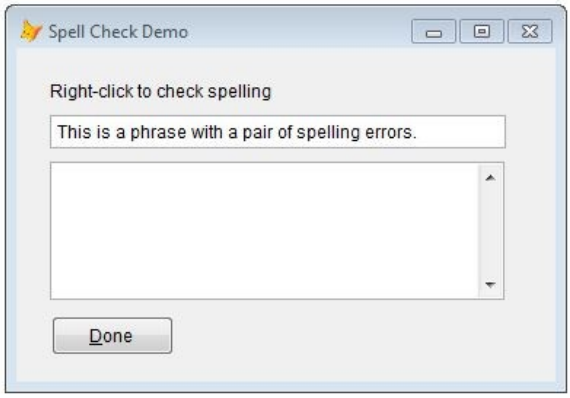

Figure 3. The corrected phrase.

Escaping out of the Spell Check Choices dialog before handling every suggestion will close it. Upon handling the last suggestion, the dialog will automatically close.

Enhancement Requests

As every developer knows, applications never get smaller. The minute you deliver a build, the user comes back with "This is great, but..." and this simple spell checker is no different. Within days of using the tool, my customer came back with a number of enhancement requests.

1. Remove all other suggestions from the list once one has been chosen.

Frankly, this should have been anticipated during design, implementation and testing. A user not paying attention might double-click on a suggestion, and then double-click on another suggestion for the same word. The solution is to remove the rest of the rows in the list box for that word, and then refresh the listbox.

In the listbox's double-click method's 'else' clause, we simply compare the current word to each row in the list and delete the row if they match. We run it in reverse so that the deleted rows don't interfere with the next comparisons.

```
* ER 1
* don't close yet
local liHowManyRows, liHowManyDeleted
liHowManyDeleted = 0
liHowManyRows = alen(this.aItems,1)
for li = liHowManyRows to 1 step -1
  if alltrim(upper(this.aItems[li,1])) ;
    == alltrim(upper(lcWordToFind))
    =adel(this.aItems,li)
    liHowManyDeleted = liHowManyDeleted + 1
  endif
next
dimension this.aItems[liHowManyRows -
liHowManyDeleted,2]
this.Requery()
```

2. Provide a setting to ignore words that are in all caps.

Words that are in all caps can indicate one of three things. Either the typist is 90 years old and is still used to TYPING IN ALL CAPS, or the typist had hit the Caps Lock key accidentally, or the word is an acronym, and should stay in all caps.

In the last case, it'd be nice to have the word automatically ignored. For demonstration purposes in this article, I added a Ignore Words in All Caps checkbox to the Spell Check Demo form and store the value to a form property, lIgnoreWordsInAllCaps, that is used in place of an actual setting in the application. See Figures 4 and 5 to see how the setting affects what is displayed in the Spell Check Choices dialog.

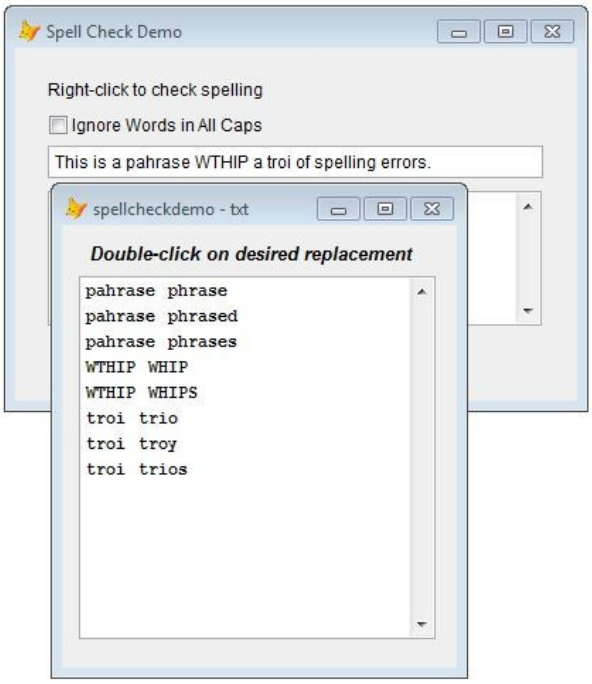

Figure 4. Misspelled words in ALL CAPS are included if the checkbox is unchecked.

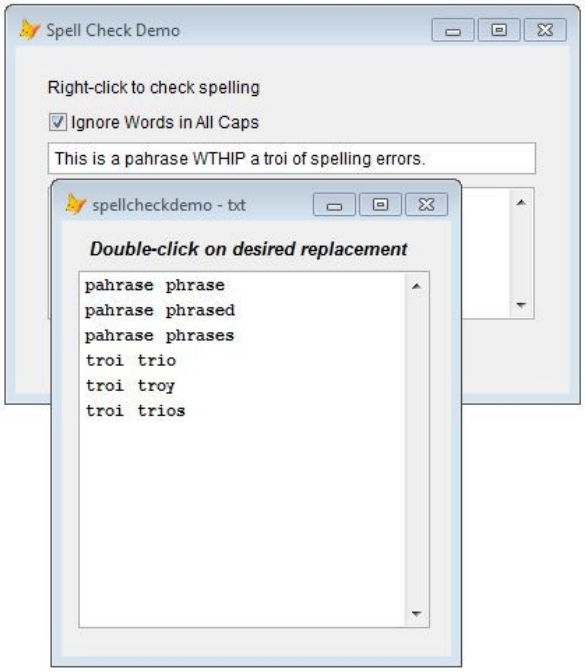

Figure 5. Misspelled words in ALL CAPS are excluded if the checkbox is checked.

The value of the Spell Check Demo form's property is captured by the Spell Check Choices init():

```
* grab the IgnoreWordsInAllCaps property
* from the caller
local llIgnoreWordsInAllCaps
llIgnoreWordsInAllCaps = ;
  loForm.lIgnoreWordsInAllCaps
```

and is then passed to the CheckSpellingOfString() method, like so:

```
if thisform.CheckSpellingOfString( ;
 lcStringToCheck, llIgnoreWordsInAllCaps)
```

The CheckSpellingOfString method in the Spell Check Choices form has then been modified twice, first for the parameters statement:

```
lparameters lcStringToCheck, ;
  llIgnoreWordsInAllCaps
```

and then with a new code segment that determines whether the word is going to be checked for spelling errors or not.

```
* ER 2
* check if word is all caps
if llIgnoreWordsInAllCaps
  * "ignore words in all caps" is true,
  * so determine if this word is all
  * caps, so that we can determine if
  * we need to handle it or not
  if thisform.WordIsAllCaps(cWord)
    * it's all caps, go to next word
    loop
  else
    * word is not all caps, so keep
    * processing this word
  endif
else
  * "ignore words in all caps" is false,
  * so we're going to handle the word
endif
```

The WordIsAllCaps() method looks like this:

```
lparameters lcWord

local liNumChars, llWordIsAllCaps, li
liNumChars = len(alltrim(lcWord))
llWordIsAllCaps = .t.

for li = 1 to liNumChars
  lcCharToCheck = substr(lcWord, li, 1)
  liAsciiValue = asc(lcCharToCheck)
  if liAsciiValue > 64 and liAsciiValue < 91
    * between A/Z
    * this letter is caps
  else
    * as soon as we find a letter that
    * isn't A-Z, we're done
    llWordIsAllCaps = .f.
    exit
  endif
next

return llWordIsAllCaps
```

3. Check for repeated repeated words.

It's easy for a user to get interrupted during data entry and then repeat a word word without realizing it. My customer requested a setting where the field would be checked for any words that were entered twice in a row.

Indicating to the user that the word was repeated was not a trivial issue to deal with, as it didn't fit the existing paradigm of a two column Choices list. We discussed several mechanisms, including automatically deleting the second instance of any word (discarded because there could be situations where the second instance was

intentional) and using a separate interface to handle repeated words (discarded due to increased complexity.)

We decided to include the repeated word in the first column of the Spell Check Choices listbox, but not offer a 'correction' in the second column. Instead, double-clicking on that row in the listbox would remove the second instance from the field.

Again, for demonstration purposes in this article, I added a Flag Repeated Words checkbox to the Spell Check Demo form and store the value to a form property, lFlagRepeatedWords, that is used in place of an actual setting in the application. See Figure 6 to see how the setting affects what is displayed in the Spell Check Choices dialog.

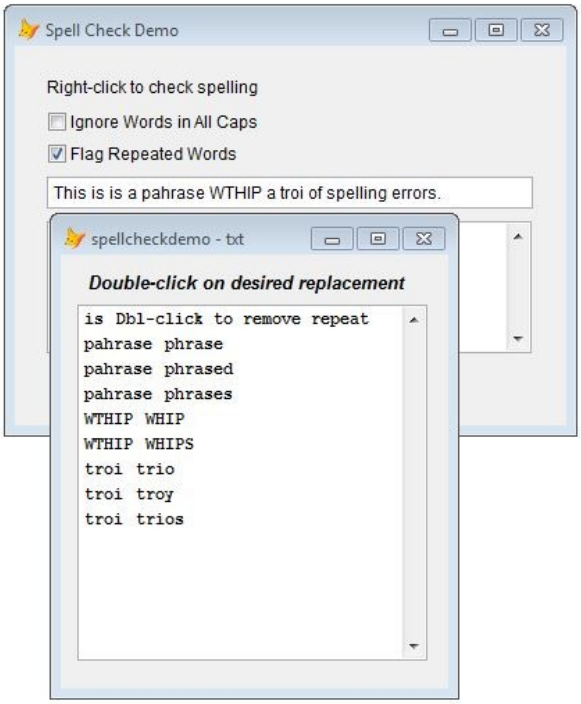

Figure 6. A repeated word is included in the Choices listbox, with a flag indicating that it's repeated.

Implementation was a little trickier, as the way the list in the Spell Check Choices dialog was populated, handled, and cleaned up is more complex.

The value of the Spell Check Demo form's property is captured by the Spell Check Choices init():

```
* grab the IgnoreWordsInAllCaps property
* from the caller
local llFlagRepeatedWords
llFlagRepeatedWords = ;
  loForm.lFlagRepeatedWords
```

and is then is added to the parms passed to the CheckSpellingOfString() method, like so:

```
if thisform.CheckSpellingOfString(;
  lcStringToCheck, llIgnoreWordsInAllCaps,;
  llFlagRepeatedWords)
```

The CheckSpellingOfString() method in the Spell Check Choices form now has an additional chunk of code that does the checking for repeated words. We'll assume that we'll be case sensitive.

We go through the entire string, looking at each word in turn. Once we've grabbed a word, we then examine the following word to see if it's the same. If so, we increment iNumMisspelledWords, the counter for the number of misspelled words in the string (I know, that seems odd, I'll explain why in a moment) and the counter for how many rows are in the list to present to the user. Finally, we add the repeated word to the list of words to present to the user.

Note that the list of suggestions to present to the user uses a form property, dblClickRepeatMessage, to stuff into the second column of the listbox. This string (in this article, "Dbl-click to remove repeat") will be used several times through the code, and instead of hard-coding a literal multiple places, it made sense to put it in one place.

```
* ER 3
* look for repeated words
* assume case sensitive
for liWord = 1 to
thisform.iNumOfWordsInStringToCheck
  lcWord = getwordnum(lcStringToCheck, ;
  liWord )
  * we have a word in the field
  * now roll through the whole field again,
  * and look for this word in the field
  for liWord2 = liWord+1 TO liWord+2
    lcWord2 = getwordnum(lcStringToCheck,;
    liWord2 )
    * if this word in the string matches the
    * word we were looking for
    * stuff into Suggestions array, which
    * will be put into listbox
    if lcWord2 == lcWord
      thisform.iNumMisspelledWords =;
        thisform.iNumMisspelledWords + 1
      thisform.iSuggestionCount =;
        thisform.iSuggestionCount + 1
      dimension This.aSuggestions[ ;
        this.iSuggestionCount, 3]
      thisform.aSuggestions[ ;
        thisform.iSuggestionCount, 1] = liWord
      thisform.aSuggestions[ ;
        thisform.iSuggestionCount, 2] = lcWord
      thisform.aSuggestions[ ;
        thisform.iSuggestionCount, 3] =;
        thisform.DblClickRepeatMessage
    endif
  next
next
```

After this is all done, we'll then do the actual spell checking as described earlier in this article.

Finally, the DblClick method of the listbox needed to be modified, so that the row would be removed from the list if the user double-clicked on it. First, the iNumMisspelledWords variable is used to track how many rows in the list need to be

considered when deciding whether to automatically close the form when the last row has been handled. That's why it is updated for the repeated rows code segment described earlier in this ER.

And, second, the code segment that removed the other choices for the double-clicked words needed to be modified, since we only want to remove that one row, has been modified to include a check for a repeated word scenario:

```
if alltrim(this.aItems[this.ListIndex,2]) ;
  == thisform.cMsgDoubleClickToRemoveRepeat
  * remove the line with the repeated word
  =adel(this.aItems,this.ListIndex)
  liHowManyDeleted = liHowManyDeleted + 1
```

4. Automatically execute spellcheck immediately upon field exit.

First, I added a flag to both the txt and edt classes that the controls on the demo form are based on. This property is called lAutoSpellCheckUponFieldExit and it initialized to .f.

In the LostFocus of the txt and edt classes, I added this code snippet:

```
* ER 4
if this.lAutoSpellCheckUponFieldExit
  wait window nowait 'Checking spelling...'
  do case
  case this.cSpellCheckType = 'FORM'
    do form spellcheckchoices with thisform,
this
  case this.cSpellCheckType = 'POPUP'
    do spellcheckshortcut.mpr WITH thisform,
this
  endcase
endif
dodefault()
```

Note that we don't ALSO check this.lAllowSpellCheck as is done in the right-click method of the txt and edt classes. If this property is set to true, we do the grown-up thing and assume that spell check is allowed.

Then, in the Spell Check Demo form, I added a Auto Spellcheck Upon Field Exit checkbox. See Figure 7. Checking this box causing the spellcheck to automatically fire upon tabbing out of either field.

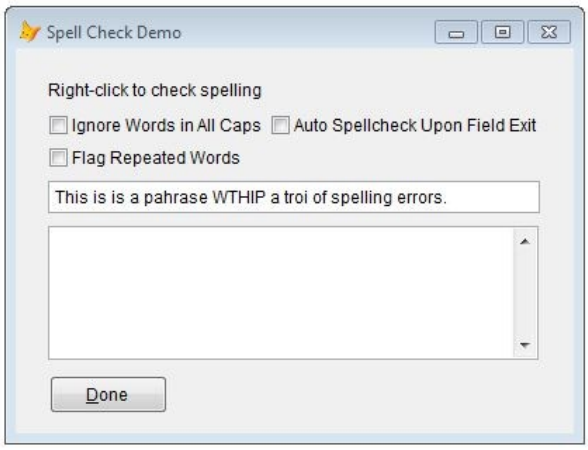

Figure 7. Demonstrating the Auto Spellcheck feature.

In the click() of Auto Spellcheck Upon Field Exit checkbox, set the lAutoSpellCheckUponFieldExit property for each field to the same value as the checkbox. Note that this is a 'all or nothing' mechanism used for demonstration purposes in this article. In my customer's application, this property was explicitly set, field by field, as desired, in the instance of the form.

```
* ER 4
thisform.lAutospellcheckuponfieldexit = ;
  this.value
thisform.txt.lautospellcheckuponfieldexit = ;
  this.Value
thisform.edt.lautospellcheckuponfieldexit = ;
  this.value
```

5. Provide a settings dialog to configure how spellcheck works.

As you could surmise, this was started when the user asked, way back when, "Can we have a setting to ignore words that are in all caps?"

The logical question that immediately arose was "Is this setting user-specific or application-wide?" The app had a goCust object where all settings for this application were stored. These settings came from a table with the structure

```
cUser c(3)
cSetting c(50)
cValue c(20)
```

See Figure 8.

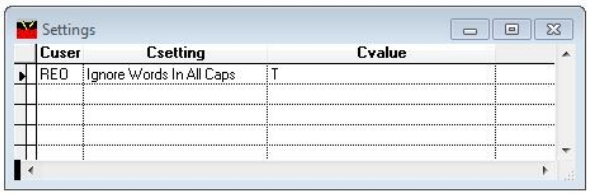

Figure 8. The DBF for the system settings table.

Settings that were applicable to everyone had no value in the user column while settings that

were set on a user-by-user basis had a username included for that row. Upon instantiation of goCust, the setting was pulled from the settings table and stuffed in the goCust object. A sample settings dialog is shown in Figure 9.

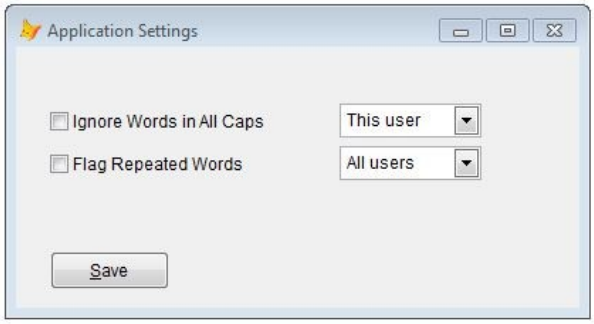

Figure 9. System settings on a local or global basis.

6. Add word to dictionary.

It's not uncommon to have words flagged as misspelled but are actually legitimate words. One-off cases can simply be ignored, but if such a word is encountered regularly, it'd be nice to have the word automatically be ignored upon subsequent spellchecks.

All you clever programmers out there probably thought that the best way to implement this would be to use a right-click method on a misspelled word to add it, but this isn't obvious or necessarily intuitive, nor may it be easily discoverable. And I should know, because that was my first thought as well. Watching a user trying to figure this out on their own (even with a label on the form saying 'right-click to add to customer dictionary') showed me that.

So we settled on including a suggestion in the list below the suggested words, such as "Add Word to Dictionary".

See Figure 10.

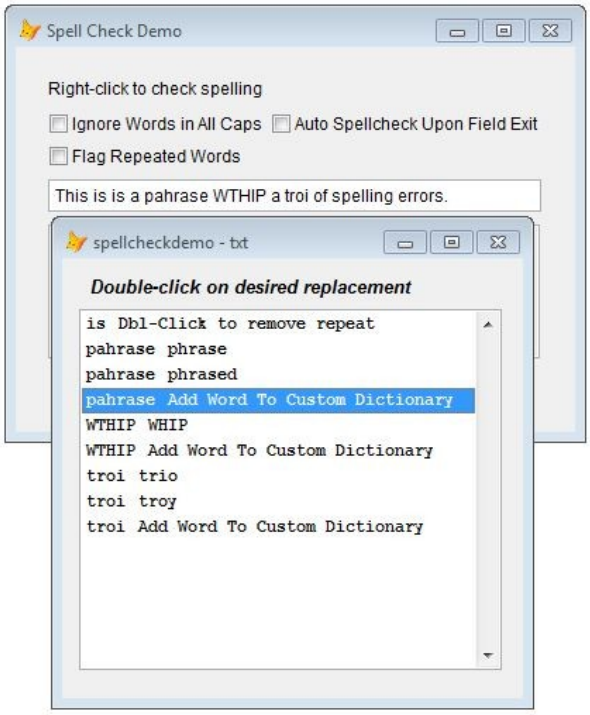

Figure 10. Offering to add misspelled words to a custom dictionary.

Naturally, if the Add Word to Dictionary choice is selected, the list is cleared of suggestions for that word, and then is refreshed per the first enhancement request.

So that's the interface. But there were a pair of issues involved that were initially transparent to the user.

First, does each user want their own custom dictionary, or should each word get added to a global dictionary?

Second, under the hood, should the Word dictionary be used to store newly added words, or should a VFP table be used?

We chose to use a global dictionary, because it was going to be common for multiple users to rely on the same list of custom word choices. Given that choice, we also chose to use a VFP based custom dictionary, because adding to a user's Word dictionary is significantly more complicated than stuffing a VFP dictionary. Add to that the complexity of implementing and maintaining a shared Word dictionary across the network, and it was an easy choice to go with a VFP custom dictionary.

The dictionary itself was pretty simply, just a single column.

```
cWord c(50)
```

See Figure 11.

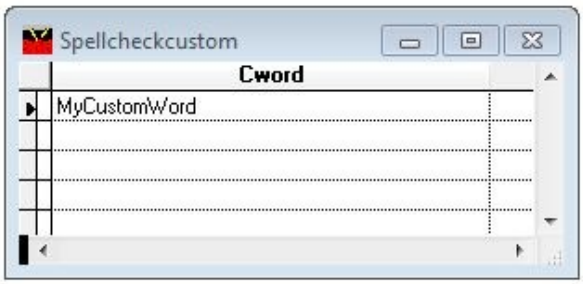

Figure 11. The DBF for the custom dictionary.

Implementing this wasn't too difficult.

First, we needed to add the row to the listbox for each misspelled word. Adding this code segment right after handling the suggested corrections collection in the CheckSpellingOfString() method did the trick:

```
* ER 6
* add one more row for 'Add word to
dictionary'
dimension This.aSuggestions[ ;
  This.iSuggestionCount+1, 3]
thisform.aSuggestions[ ;
  thisform.iSuggestionCount, 1 ] ;
  = liWord
thisform.aSuggestions[ ;
  thisform.iSuggestionCount, 2 ] ;
  = lcWord
thisform.aSuggestions[ ;
  thisform.iSuggestionCount, 3 ] ;
  = thisform.cMsgAddWordToDictionary
```

The other piece of the puzzle was to handle the user's action when they double-clicked on the "Add Word" line in the listbox. This simply required a bit of reworking of the processing logic.

Remember the following construct that handled the double click depending on if the user was clicking on a correction or on the 'Remove Repeated Word' line?

```
if alltrim(this.aItems[this.ListIndex,2]) ;
  == thisform.cMsgDoubleClickToRemoveRepeat
else
endif
```

It now becomes the ELSE clause of a larger construct that handles the Add Word line:

```
if alltrim(this.aItems[this.ListIndex,2]) ;
  == thisform.cMsgAddWordToDictionary
else
  if alltrim(this.aItems[this.ListIndex,2]) ;
    == thisform.cMsgDoubleClickToRemoveRepeat
  else
  endif
endif
```

Interestingly enough, the code that handles emptying the listbox after the user double-clicks doesn't have to be changed at all. The reason is that the offered corrections and the 'Add Word' prompt are all processed in the same way.

7. Use custom dictionary during spell check.

Once the dictionary is in place, it was trivial to incorporate its use. In the CheckSpellingOfString() method, adding the following code segment immediately after the

```
lcWord = getwordnum(lcStringToCheck, liWord )
```

line

```
* ER 7
* spell check using VFP custom ;
* dictionary first
select count(*) from SPELLCHECKCUSTOM ;
  where upper(alltrim(cWord)) ;
  = upper(alltrim(lcWord)) ;
  into array laHowManyMatches
if laHowManyMatches[1] > 0
  loop
endif
```

If the word was found in the custom VFP dictionary, we just looped to the next word in the string. So there you go.

8. Launch all actions from context menu.

Finally, the last request the customer had was to change the initial action executed when the user right-clicked on a field. Instead of automatically executing the spellcheck mechanism (and opening the Spell Check Choices form), they wanted a context menu that displayed multiple choices, as shown in Figure 12.

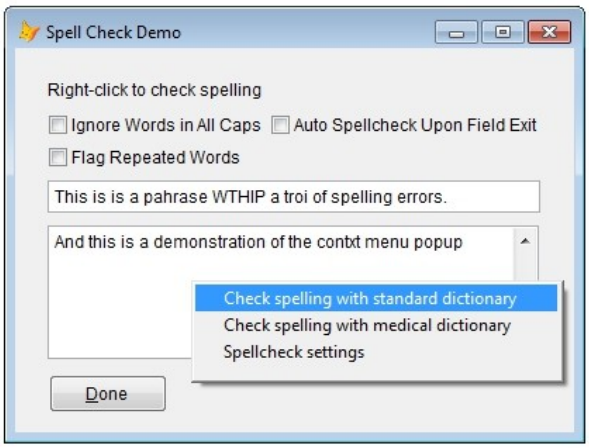

Figure 12. Context menu offering multiple spell-check choices upon right-clicking in a field.

I implemented this option for them by modifying the RightClick() and LostFocus() methods of the text and edit control base classes to include a property that drove whether the behavior would be to immediately run the spellcheck form, or to run a popup first, and then test the property and make the appropriate action.

```
* ER 8
```

```
do case
case this.cSpellCheckType = 'FORM'
  do form spellcheckchoices with ;
    thisform, this
case this.cSpellCheckType = 'POPUP'
  do spellcheckshortcut.mpr with ;
    thisform, this
endcase
```

This way, once the customer discovered that they weren't going to implement the medical dictionary after all, and wanted to revert back to displaying the Spell Check Choices form immediately upon right-clicking or tabbing through the form, they would only have to make changes to two properties.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com