FoxTalk™

A comprehensive monthly guide for users of FoxBASE+® and FoxPro® February 1993

CHEKLIST() Multi-selection List

By Ted Roche

Many developers have created routines to allow the operator multiple choices from a predefined list. For example, Sherri Kennamer created FLDLIST.SCX (found in the FoxPro\Goodies\ Screens directory) to return a list of database fields. Y. Alan Griver's CodeBook demonstrates a more generic dialog in a program called MOVER.PRG. *[FoxTalk will soon bring you a powerful MOVER-style dialog by John Hosier.—Ed.]*

CHEKLIST.PRG was designed with these goals:

- Unlike the MOVER programs, it uses very little screen space on the main entry screen.
- The code and functionality are minimized to create a reusable "black box" routine that uses a small amount of code, disk space, and memory.
- The checklist visual appearance creates a familiar interface for the operator.

FoxPro 2's interface has greatly simplified the design of complex screen objects. You can use a list object within a generated screen to create a multiple-selection checklist. It is a tribute to both the power and the simplicity of FoxPro's Power Tools that the creation of the CHEKLIST.PRG required only about 30 lines of code to be custom written (Figure 1).

The remainder of the commands are created automatically by the Screen Generator from objects designed in the Screen Builder, and can easily be maintained using that interface as well.

In this issue

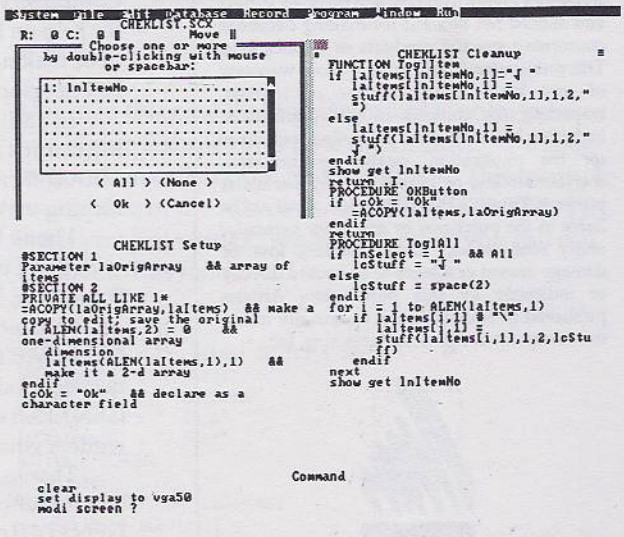

Figure 1. Screen Builder with all CHEKLIST's snippets open and arranged.

Continues on page 3

CHEKLIST()

Continued from page 1

In "A One-line Pushbutton Pop-up" (*FoxTalk*, October 1992), Hank Fay described his design criteria for black box routines. His thought-provoking list presents a standard against which all generic routines should be measured for functionality, speed, independence, and ease of use.

CHEKLIST supplies the necessary functionality to create its window, process actions against the checklist items, and clean up after itself. Since CHEKLIST processes only one external parameter through a private duplicate of the list, its speed is the same as if it were a dedicated, hard-coded routine. CHEKLIST ensures independence from the surrounding application through the use of PRIVATE variables and the preservation and restoration of environment settings that need to be explictly set within CHEKLIST.

Finally, CHEKLIST requires only one parameter, allowing the developer to create the array to be passed to CHEKLIST appropriately.

Creating an array for the list

Before calling the CHEKLIST program, you must create the character array of choices and then process the array to preselect or disable items. An array can be created, using DECLARE/DIMENSION statements, and then filled programmatically. Alternatively, the new SQL - SELECT command can create, dimension, and fill the array from a database in a single statement.

Let's say we have a database GROUPS.DBF, which contains GroupKey, a 10-character field with our checklist options. To create an array, laChoices, containing the list of all the GroupKeys in alphabetical order, use the statement:

```
SELECT Groups.GroupKey ;
  FROM Groups ;
  ORDER by Groups.GroupKey ;
  INTO ARRAY laChoices
```

This is a relatively simple use of an SQL - SELECT statement. A more complex application can create the array and complete much of the processing described below. Using advanced techniques would depend on the developer's need for speed in the final product versus the more challenging coding and debugging required to create them. More complex SQL - SELECTs are described in the box on page 6.

At this point, we've created an array, by using either a SELECT command or DIMENSION and STORE commands, and we need to process it. Processing the array before passing it to CHEKLIST requires us to perform some minor tasks:

- Make room for the checkmarks
- "Check off" any items that should be already selected
- Optionally, disable one or more items

To display an item as "checked off," precede it with a checkmark (√ - ASCII 251) and a space. If an element should not be selected, precede it with two blank spaces. If there are elements you want to display in the array but not to allow the operator to select or deselect, their first character should be a backslash (\), followed by a check and a space or two spaces, as appropriate. FoxPro automatically disables options starting with a backslash, showing them in a different color, and preventing them from being selected.

Invoking CHEKLIST

CHEKLIST can be used in several ways. It can be called as a user-defined function (UDF) within a screen by creating a radio button with the appropriate prompt and defining the VALID clause as an expression CHEKLIST(@ArrayName), where "ArrayName" is your predefined array. Notice the @ symbol, which tells FoxPro to pass the variable by reference. This allows CHEKLIST to operate directly on the array that was passed as a parameter. If you omit the symbol, only the first value of the array will be passed (parameters are passed to UDFs by value as the default) as a character string, and the CHEKLIST function will fail when it attempts to perform array functions on a character string (see the FoxHelp topic on SET UDFPARMS for a detailed explanation). CHEKLIST can also be invoked within code this way:

```
do CHEKLIST with ArrayName
```

Notice in this case that the @ symbol is not used.

Figure 2 shows you how the CHEKLIST dialog looks when called from another screen. This example is available in the source code as SAMPLE.APP/PJX, but if you just want to see what it does, you can call CHEKLIST as follows:

```
DECLARE laChoices[5]
laChoices[1] = "\  Peanut Butter & Jelly"
laChoices[2] = "√ Pizza"
laChoices[3] = "  Chinese"
laChoices[4] = "  Soda"
laChoices[5] = "√ Beer"
ACTIVATE SCREEN
? "Before:"
DISPLAY MEMORY LIKE laChoices
 =CHEKLIST(@laChoices)
? "After:"
DISPLAY MEMORY LIKE laChoices
```

Continues

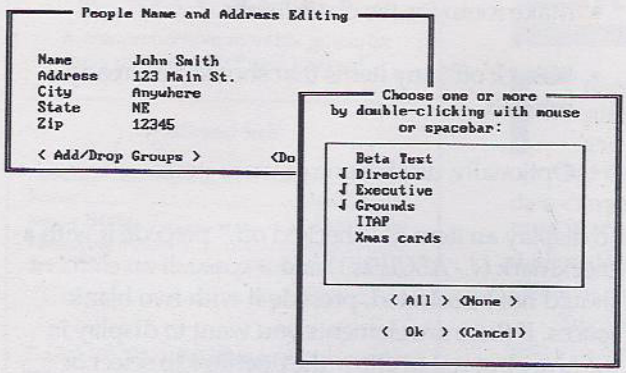

Figure 2. CHEKLIST working with sample data.

When the CHEKLIST function is called, it will display a modal dialog to the operator, with items selected, deselected, and disabled as you have initialized them in the array. The operator selects checklist options by double-clicking on them or double-spacing on the keyboard. The operator can also access the <All> or <None> buttons to toggle all the options (except those disabled) by clicking on the buttons with the mouse or tabbing to them and pressing <Enter> on the keyboard. If <Cancel> is selected, the original array is returned to the calling program. If <OK> is chosen instead, all changes made by the operator are returned in the array. The developer can then review array elements, taking the actions indicated by the operator's choices.

How CHEKLIST works

CHEKLIST.PRG is generated directly from a screen (CHEKLIST.SCX) designed with the Screen Builder. You can MODIFY SCREEN CHEKLIST in the Screen Builder, using source code files that accompany this article. The Screen Layout options selected define the screen as a Window of Type... Dialog. Select Program Generate from the menu and specify the .PRG extension in the GENERATE modal dialog; otherwise you won't be able to use CHEKLIST as a UDF. Select code options to [X] Define and [X] Release Windows, [X] Read Cycle and [X] Modal. Screen snippets required are a Setup clause, a LIST object, an All/None pair of pushbuttons, an OK/CANCEL pushbutton pair, and a set of procedures stored in the Cleanup/ PROCEDURES section. The Setup code is:

```
#SECTION 1
Parameter laOrigArray   && array of items
#SECTION 2   * Make a copy to edit; save the original
private all like l*
=ACOPY(laOrigArray,laItems)
if ALEN(laItems,2) = 0              && one-dimensional array
  dimension ;
  laItems(ALEN(laItems,1),1)  && make it a two-dimensional array
endif
lcOk = "Ok"   && declare as a character field
```

The #SECTION 1 command is a Screen Generator Directive (a Screen Builder preprocessor command) that causes all commands following it (until the end of the snippet or a #SECTION 2 command) to appear at the beginning of the generated code.

Since FoxPro requires that a PARAMETERS command appear as the first executable line of a program, the #SECTION command allows us to move the Setup code from its usual position after the window definition commands. Although the #SECTION 2 command is not required in this case, it is good practice for the Setup clause commands to appear in the normal sequence of the generated code. You may need to depend on that later.

The PRIVATE ALL LIKE L* command ensures that variables used in CHEKLIST will not change the values of identically named variables in calling routines, but rather hide such variables and create a new one for this routine only. The "L*" specifies only those variables beginning with the letter L, which is the standard adhered to in this program for "L"ocal variables. Why not just use PRIVATE ALL and not worry about what the variables are called? PRIVATE ALL should be avoided because of its effect on system variables. System variables (starting with an underscore character) that have not been defined before this routine would be created as Private variables and released upon exiting. That could lead to erratic system behavior.

In the next line, the ACOPY function makes a duplicate array for the List element. Working on a duplicate allows you to return the original array to the calling routine unchanged until the operator commits to the changes.

The next three lines, containing the IF...ENDIF structure, handle the situation where CHEKLIST is passed a single-subscripted array as a parameter. One- and two-dimensional arrays exhibit different behavior, particularly when it comes to referring to specific elements. Instead of coding for the exceptional case of a one-dimensional array, convert a one-dimensional array to a two-dimensional array containing one column. A handy feature of ACOPY() converts one-column two-dimensional arrays back to one-dimensional arrays automatically when the modified array returns in the Cleanup.

Select the list object by picking the List... option from the Screen Builder menu or by pressing <Ctrl-L>. List objects can work with lists from many sources: arrays, popups, database structures, database field values, or directory lists. In CHEKLIST, the list is supplied by the array laItems, the ACOPY() of the array passed to the routine as a parameter. When an item is double-clicked or double-spacebarred, the list's VALID clause calls the routine TogIItem() to toggle the checklist

item on or off. TogIItem() is in the Cleanup/PROCEDURES section:

```
FUNCTION TogIItem
if laItems[lnItemNo,1]="√ "
   laItems[lnItemNo,1] = ;
      stuff(laItems[lnItemNo,1],1,2,"  ")
else
   laItems[lnItemNo,1] = ;
      stuff(laItems[lnItemNo,1],1,2,"√ ")
endif
show get lnItemNo
return .T.
```

The IF...ENDIF structure toggles the element between selected and deselected. The SHOW GET command immediately updates display of the list element. If this command were omitted, the check would not appear on the item toggled until the operator moved to another item.

The All/None pair of pushbuttons (created using the menu option Screen/Pushbutton or <Ctrl-H>) toggles all checklist items on or off by calling the routine TogIAll(). TogIAll() is also located in the Cleanup/PROCEDURES section:

```
PROCEDURE TogIAll
if lnSelect = 1   && All
   lcStuff = "√ "
else
   lcStuff = space(2)
endif
for i = 1 to ALEN(laItems,1)
   if laItems[i,1] # "\"
      laItems[i,1] = stuff(laItems[i,1],1,2,lcStuff)
   endif
next
show get lnItemNo
```

Because the value of the pushbutton was not predefined, the return value, lnSelect, is numeric. The Screen Builder adds a "DEFAULT 1" clause to the generated @...GET code for the pushbutton. This prevents an error if the variable has not been created before the GET statement. However, a pushbutton with a numeric return value can be more difficult to maintain. The @...GET works fine, but if you change the order of the two pushbutton prompts in the Screen Builder pushbutton dialog, the VALID clause will reverse: <All> will now select none, and <None>, all!

The OK/Cancel pushbutton, on the other hand, had a value predefined in the SETUP clause, lcOK = "Ok," which will make the return value a character string. A character return value makes the code easier to understand and avoids the problem altogether.

Be careful! Although you no longer have to worry about the order of prompts, you must not change the text of prompts without making matching changes to the code that processes the return values. The VALID clause OKButton() is run when this terminating pushbutton is selected. OKButton() is also in the Cleanup/PROCEDURES code:

```
PROCEDURE OKButton
if lcOk = "Ok"   && Ok, return the changes
   =ACOPY(laItems,laOrigArray)
endif
return
```

Coding conventions

The VALID clauses for the List, All/None, and Ok/Cancel pushbuttons are defined within their Screen Builder dialogs as (*) Expressions and given names (TogIItem(), TogIAll(), OKButton()). This avoids unpronounceable names the Screen Generator assigns to snippets and makes the program easier to follow.

There is some inconsistency in the use of FUNCTIONs versus PROCEDUREs in the Cleanup code. The first returns a logical True, the second has no return clause at all, and the third just returns, with no return value. What's best? If we asked a dozen developers, we'd get a dozen answers. Code snippets return a .T. value by default, so a return is not required. Many developers adopt a convention where a function returns a value, as the built-in square-root function does. A procedure performs a series of commands. Choose one and stick with it.

Enhancing CHEKLIST()

There is nothing special about using the checkmark character. Just make sure it is changed wherever it appears in the code, or enhance CHEKLIST by substituting a memory variable wherever the checkmark appears in the code. With a memory variable, only a single line of code would have to be changed. If you use CHEKLIST with FoxPro for Windows, a CASE structure testing the system variables _DOS and _WINDOWS would allow a different character in each version. This is particularly important if you use checklist with a font other than the default FoxFont. Most Windows fonts will not support the same upper set of symbols (CHR(128) through CHR(255)). See Figure 3 for an unmodified use of CHEKLIST under FoxPro for Windows. This example uses the simple snippet suggested above to create the data and call the function. Notice the disabled element at the top of the list and its match in the top array element (with backslash) displayed behind the window. The list elements behind the window do not display the check mark properly because the screen font, unlike the window font in the unmodified CHEKLIST, didn't happen to be FoxFont.

Continues

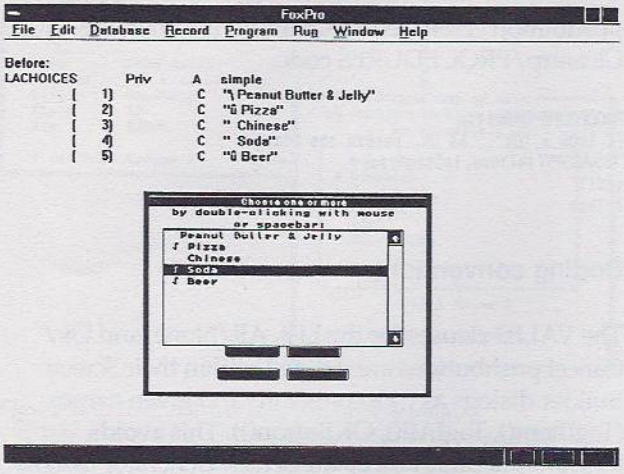

Figure 3. CHEKLIST, under FoxPro for Windows.

The List object will work with either single- or doubled-subscripted arrays. List will only display the first column of elements, so the options must be in the first column of a multicolumn array. Using a two-column array would allow a second column to hold the original value of the first, perhaps as a logical value. That way, you could process only items changed by the operator within CHEKLIST—a big difference in the time to process very large lists.

Other enhancements:

- Passing a title as a parameter
- Passing a logical parameter to control the "All/None" pushbutton display
- Varying the window and popup size with the length of the array prompt

In this example, designed to use minimal code, the operator must double-click options on or off. You could add ON KEY LABEL statements to redefine keys that operate on a single mouseclick or spacebar.

CHEKLIST.PRG easily integrates a multiple selection checklist into your applications by adding a single UDF call in the VALID clause of a button. It is a generic, reusable object, a "black box" handy when a multiple-choice list must be made from an array of items where <All> or <None> options are appropriate. The items might be a fields list for a scope argument on a REPORT FORM or SCAN, a list of cities, or a grocery list. Reusing the same code in multiple places in an application reduces the amount of code to maintain, lowers overall application size, and creates a simpler, consistent interface.

Ted Roche is a senior programmer for The New Hampshire Insurance Group in Manchester, N.H. In six years of analysis, design, and programming, he has completed projects in FoxPro, dBASE, and Clipper for clients in the manufacturing, state-regulatory, and financial-services industries. Ted is exploring full-time employment or long-term contracting in central and southern New Hampshire. He can be reached at 603/746-4017, or on CompuServe at 76400,2503.

More Advanced SQL Usage

As shown on page 3, a single SQL statement can create the array and preformat it with appropriate checkmarks, spaces, and backslashes. If no elements were selected or disabled:

```
SELECT space(2)+Groups.GroupKey ;
   FROM Groups ;
   ORDER by 1 ;
   INTO ARRAY laChoices
```

would create the laChoices array, each element preceded by two spaces.

But let's say a second table, Gruplist, contains records of people and lists they belong on. The people are designated with a 10-character field, PeopleKey, and the groups with a 10-character field, GroupKey. The following commands create the array and preselect items already selected for "SMITH00001":

```
USE gruplist ORDER TAG GroupKey
SET FILTER TO PeopleKey=="SMITH00001"
SELECT ;
     IIF(seek(Groups.GroupKey,"GRUPLIST"),"√ ",space(2)) ;
     + Groups.GroupKey ;
   FROM Groups ;
   ORDER by Groups.GroupKey ;
   INTO ARRAY laChoices
```

In a single SQL statement:

```
SELECT "√ "+GROUPS.GROUPKEY, GROUPS.GROUPKEY;
   FROM GROUPS;
   WHERE GROUPS.GROUPKEY IN;
      (SELECT GRUPLIST.GROUPKEY;
         FROM GRUPLIST;
         WHERE GRUPLIST.PEOPLEKEY = "SMITH00001");
UNION ;
   SELECT "  "+GROUPS.GROUPKEY, GROUPS.GROUPKEY;
      FROM GROUPS;
      WHERE GROUPS.GROUPKEY NOT IN;
      (SELECT GRUPLIST.GROUPKEY;
         FROM GRUPLIST;
         WHERE GRUPLIST.PEOPLEKEY = "SMITH00001");
   ORDER BY 2;
   INTO ARRAY laChoices
```

Why a second column in the second example? The first example used the ORDER BY GROUPKEY clause to sort the elements in their "natural" order. But a checkmark precedes some elements in the second example. If alphabetically sorted, the checkmarked entries would be grouped together at the end of the final array.

Since both methods work, choose the first. It's simpler. It uses seven lines of screen space, the second uses 15. The first creates a one-column array, the second, a wasteful two. But the second method might have a performance advantage in SQL/Rushmore optimization because it is pure SQL. Despite its length, it is only one line of code (notice the semicolons).

These code snippets may require that GRUPLIST.DBF not be open and that an empty select area be found. They are examples, not complete solutions.