**Array and GET tricks**
**(tricks to using Clipper 5.01)**

**Data Based Advisor**
July 01, 1991 Straley, Stephen J.; Tamburrino, Jim

Here are some undocumented tips we discovered from looking at STD.CH:

* You can pass arrays by reference to code blocks
* You can build a message system that's like the one you'd get with @..PROMPT by using a GET

In our January article we stressed the importance of using the standard **Clipper** Header file (STD.CH) for learning, programming, and inspiration. We'll depart from our regular business to bring you some undocumented **Clipper** 5.01 tricks that came to us by looking at STD.CH.

**Magic with arrays**

Two basic rules apply to arrays:

* Arrays are passed by reference from one subroutine to another.
* **Array** subscripts are passed by value.

When multi-dimensional arrays appeared in **Clipper**, some programmers began using them for the GET system. In other words, rather than using a memory variable to mimic the name of a database field, you would use an **array**. This appears to be a preferable method because you can set up a one-to-one relationship between the ordinal position of a field in a database and the subscript position of an element in an **array**. But a problem in using arrays crops up in a GET in the inevitable user-defined function (UDF) used as a look-up/validation routine. Consider this Summer '87 style of code:

```
@ 10,10 GET array[6] VALID LookUp( array[6] )
```

What if users wait a DBEDIT() in the lookUp() function and need to make a selection from that pick-list and have it appear in the GET on the screen? You pass the variable **array**[6] by reference rather than by value. Remember, when a variable is passed by reference, any change to the variable in the subroutine will be reflected once the subroutine has finished. Passing a variable by value simply means only the variable's value is passed and any change to it will not show up. With **Clipper** 5.0, you can pass the **array** element by reference using a code block. Consider the program in Listing 1, which passes an **array** element by reference.

The preprocessor translates the characters appearing in the function call on lines 25 through 27 into the code block: on line 19. For the first GET on line 25, the code block is passed to the function Lookup(). This code block serves two purposes. If evaluated with a call to the EVAL() function without any parameters, the value of the element **array**[1] will be returned. You can see an example of this on line 40.

If, on the other hand, a value is passed to the EVAL() function, along with this code block, the value is stored to the **array** element. Since the code block is created within the same procedure as the **array** on line 21, the operation performed on the **array** element is valid, even though the **array** is declared LOCAL. This is critical in using **array** technology without taking additional symbol space by declaring it PUBLIC or PRIVATE.

When building UDFs to work on a WHEN or VALID clause, always give escape routes. For example, on line 37 the rest of the code executed, so long as the last key pressed wasn't the Up Arrow key.

In addition, try to make functions as generic as possible. Here, the data type of the formal parameter value may be either a code block or "something else." If it's something else, then it's considered a regular variable passed by value. In this case, no EVAL() needs to be performed to obtain its value nor does it need another call to EVAL() to store a value to it. Lines 39 and 40 are, therefore, complicated. They state that if the data type of the variable value is a code block, do an EVAL() on the variable; otherwise, simply return its value.

The returned value then becomes the value of the IF() function within the call to ASCAN(). That value is then searched in the **array** of choices[]. If a match is found, then ASCAN() will return the subscript position of the match. If no match is found, then ASCAN() will return 0, which is EMPTY(). Therefore, if no match is found, a group of selections are PROMPTed to the screen, and once a selection is made, either a call to EVAL() (line 60) is made with that selected item or that selection item is simply stored to variable (line 62). If that variable was passed by reference with a single "@" sign, the variable will have a new value. In either case, passing a parameter or **array** element by reference is possible.

**The good old SET KEY days**

Back in the days of Summer 87, it would have been nice to have the ability to pass a variable by reference that was-currently active in a GET without having to macro expand the third parameter. We worked around the problem by using one of three parameters passed to the subroutine activated by the SET KEY TO command. Now, instead of a work-around, how, about actually passing the variable in the GET as a parameter to the subroutine? Consider the **Clipper** 5.01 code in Listing 2.

In this program, we've provided two tricks. First, we use the WHEN clause to simulate a MESSAGE string-similar to the @..PROMPT command-on individual GET commands. Second, we take the SETKEY() function and pass individual values. By combining the two ideas, you could have context-sensitive hotkeys in individual GETS. As users move from one GET to the next, the WHEN clause will reset the SETKEY() function to call the proper function and pass the proper values.

To see what you're doing, here's the #command in STD.CH which handles the SET KEY TO command:

```
#command SET KEY  TO ;
    =>  SetKey( , { | p, l, v | ( p, l, v ) } )
```

The preprocessor replaces the SET KEY command with the call to the SETKEY() function. is the match marker for the INKEY() value that becomes the first parameter of the function. The second parameter is a code block created from the procedure name with the addition of the three parameters "p", "l", and "v". These parameters are always sent to the code block by the SETKEY() function and contain the PROCNAME(1), PROCLINE(1), and READVAR() functions. In using the SETKEY() function on line 25, we ignored these parameters by not including them between the vertical dashes in the code block. You can add your own parameters to pass to the procedure. Like any function call, these parameters can be passed either by value or reference.

In addition, we use the SET() function on line 27 rather than the SET MESSAGE TO command. Functions and expressions are the preferred direction to program in, not because they're C-like, but because they're more flexible to use. For example, the SET MESSAGE TO command has little meaning outside of being on a command line by itself. On the other hand, the SET() function can be placed in a code block that can be passed as a parameter, stored in an **array**, or even sit in a cargo instance variable for a browsing object. The choices are more open with functions and expressions than with commands.

But don't abandon the commands in the language entirely and carry this concept to the extreme. Try to remember the benefits of readable and maintainable source code, and to use the right structure for a particular task. Since the preprocessor handles the conversion of a command to a function call, you won't be hindered by poor performance at runtime for using commands.

**In conclusion**

These are just some examples of the extended features in **Clipper**. From these we can begin to build a message system for a GET similar to the message system found in the @..PROMPT command.

Looking at **array** technology, we can consider entire object-like structures that contain all the information needed to perform a specific task. For example, instead of passing four parameters of screen coordinates to a subroutine, why not pass an **array** with four elements? Additional information can be stored and passed there as well, including **array** subscripts, objects, columns, code blocks, other arrays, field values, and individual memory variables. Arrays hold new meaning in **Clipper** 5.0, and old problem-solving techniques don't even scratch the surface of these new powers.

Stephen J. Straley, formerly at Nantucket, is President of Sirius Software Development, inc., a software development and consulting firm, and also the author of Programming with **Clipper** 5.0, published by Bantam. Steve can be contacted at (415) 647-8007. Jim Tamburrino is president of LMT Computer Services, Inc., in Marietta, Georgia.