Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

TopSpeed

Clarion5 by TopSpeed

FREE Microsoft Internet Explorer

etc2000 EVENT SPONSOR

September 2000 Index

Creating An MS OutLook-Style Menu In Clarion: Part 2
Microsoft Outlook uses an innovative menu style that's become quite popular. Now Steffen Rasmussen shows how to create the same style of menu in a Clarion application. Part 2.
(Sep 5,2000)

The Clarion Advisor: Better Debugging With DebugView
The Clarion Advisor goes debugging again, this time with a free utility and an easy API call.
(Sep 5,2000)

The Nuts And Bolts Of Passing Parameters: Part 1
Any time you divide a program up into procedures, you need a way for those procedures to communicate. In this two part article, James Cooke explores the many facets of passing parameters.
(Sep 12,2000)

Five Rules for Managing Complexity: Part 3
In the third of this five part series, Tom Ruby explains how to eliminate columns that don't belong in your database.
(Sep 12,2000)

Outlook Menu Templates
Steffen Rasmussen has provided an additional download for his MS Outlook-style menu articles. This new download includes menu templates.
(Sep 12,2000)

Using CHOOSE() To Concatenate Data
If you're used to concatenating strings the traditional Clarion way, you'll want to read what Carl Barnes has to say about concatenating with the CHOOSE() function.
(Sep 15,2000)

The Nuts And Bolts Of Passing Parameters: Part 2
Any time you divide a program up into procedures, you need a way for those procedures to communicate. In this two part article, James Cooke explores the many facets of passing parameters.
(Sep 15,2000)

Five Rules for Managing Complexity: Part 4
Isolate independent multiple relationships.What's that mean? Tom Ruby tells all, in the fourth part of this five part series.
(Sep 15,2000)

Five Rules for Managing Complexity: Part 5

Tom Ruby concludes his series on managing complexity with some surprising thoughts about one-to-one relationships.
(Sep 15,2000)

September 2000 News
Clarion news, notes, and happenings from around the globe.
(Sep 15,2000)

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

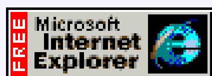
Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

TopSpeed

Clarion5 by TopSpeed



Creating An MS OutLook-Style Menu In Clarion: Part 2
(Sep 5,2000)

The Clarion Advisor: Better Debugging With DebugView
(Sep 5,2000)

Creating an "MS OutLook" Menu In Clarion

by Steffen Rasmussen

Part 2

In the previous article I described how to create the overall structure of an OutLook style menu by populating the different menu selection buttons. Now I'll show you how to quickly finish the menu by completing the Action tab with a procedure call for every button in the menu.

Or so I thought. The application that I'm developing exceeds 100 different procedures combined in different ways. So if I have to create a menu for each possible combination I'm going to end up with so many tabs, buttons and code that in a month or two it would be impossible to make any head or tail of the procedure. Reusing the code is also practically impossible. All in all I had to rethink the whole approach, although I'm terribly behind schedule.

I could just use the menu as a toolbar that never changes. This would be quite easy but I want the menu to show the menu items the user needs at any given time and nothing else. So the menu has to change dynamically depending on which procedure the user has opened.

What I really needed was a reusable menu procedure that preferably could be transformed into a template. But how do I do that?

First of all keep it simple. Unfortunately keeping it simple is often the most difficult part of programming.

I have the overall structure, so the next step is to getting the menu to interact with the different browse and update procedures. The menu selections buttons are dependent upon which procedure is in view. So instead of having the menu containing every possible combination of procedure selections, it makes more sense to have the procedure dynamically change the menu. In this way I can reuse the menu in all my applications with out changing anything in the menu. What I would have to do though, is to create some code that dynamically changes the menu selection buttons.

The Menu Strategy

As the reader probably could guess this menu program has been through many changes during the development and there will probably be a lot more in the future. But at this stage I have to clarify how the menu is going to be used to call the program procedures.

- Initially the only procedures called from the OutLook menu are Browse procedures.
- A Browse procedure should only be called from within one category.
- When a Browse procedure is opened it is not closed before the user closes the application. Note that the maximum allowed open threads is 64, so if it is a big application you will have to make sure that the user does not open any more then the maximum allowed threads.
- All procedure calls are made from the OutLook menu.
- The Update procedures are also called from the OutLook Menu but are initially called from a Browse procedure when a file is Created or edited.
- When a Browse procedure is first opened it cannot be closed again before the program terminates. I'll come back to this later.
- For each group it should be possible to start a procedure even if an other procedure is activated in an other group.

As development progresses so will this ToDo list. But for now I will show you how I have chosen to solve most of these challenges.

Let's get back to the menu.

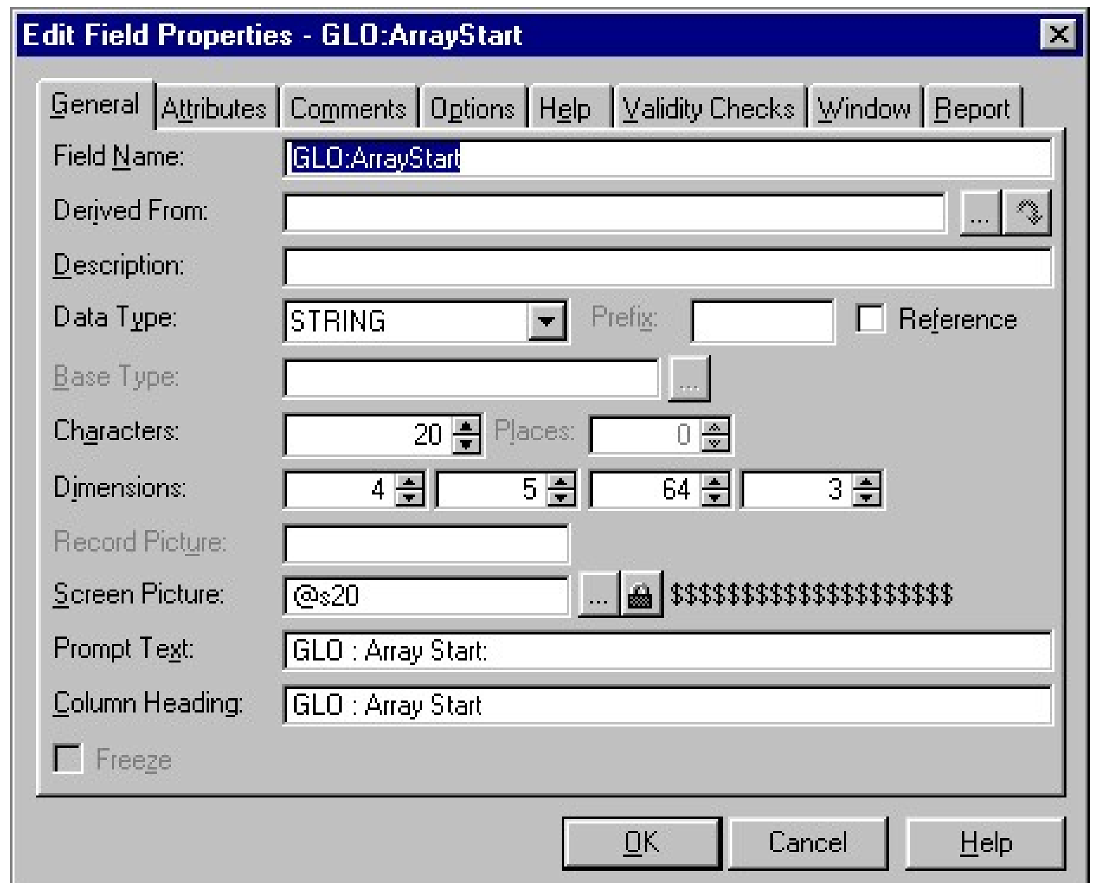
The Menu Functionality

If you have been with me so far you would have finished the overall structure of the menu. Now you just have to program the menu functionality into the different menu buttons.

The menu has four categories and within each category there are five buttons, which to the user will appear as buttons for perhaps 20 or more procedures. Each button will change functionality depending on which Group, Window and procedure button is selected. And to add to this each procedure button will also have to change icon, Button name and procedure name. All this information has to be exchanged between the different procedures and the menu. You could do this by defining a global variable for each set of information that has to be exchanged, but it will just make it too difficult to administrate all these global variables. Another option is to use one global variable containing all this information in an array.

To define a global array select Global and then the Data button. Choose Insert and populate the fields as done in the figure below:

Figure 1. Edit Field Properties for the global array.



Notice the Dimensions fields. This is where the array is created. Figure 1 shows a multidimensional array in four dimensions.

For those of you who haven't made an acquaintance with arrays I'll give a brief explanation. An array is a collection of individual data items, which all together can be treated as a single piece of data. Each data item can be accessed directly and in any order. A one-dimensional array consists of one row of data. A two dimensional array consist of a row and a column of data. A three dimensional array adds an other dimension to the two dimensional array so there are x, y and z coordinates, and so forth.

In Figure 1 there are four dimensions [1, 2, 3, 4] For each dimension you will have to declare a maximum size, which in this case is [4, 5, 64, 3]. This means that the first dimension can consists of maximum four data elements. The second dimension can consists of maximum five data elements, the third of 64 and the fourth three.

The first dimension contains the numbers from 1 to 4, which represents the four Group Buttons. The second dimension contains information about which Button (1-5) is used.

The third dimension contains the window number (1-64) and the fourth dimension contains the buttons icon, the name of the button and the name of the procedure. Apart from this array you will also need a global array to keep track of which button and window is selected:

```
GLO:ArrayButton[4,64,1]
GLO:ArrayWindow[5,1]
```

The `GLO:ArrayButton` consist of [Group number, Window number, Button number] and the `GLO:ArrayWindows`[Button number, Window number]

So why not just use the original Array[4, 5, 64, 1] since it contain all the values? The reason is that this array contains the "finished" menu structure, and in order to use it you have to know what menu item the user selected in order to get the correct array address.

Now you just have to put the array to use by populating the data elements as needed.

Start by populating the global data fields:

```
GLO:WindowThread, Long !Thread number
GLO:GroupNumber, Byte
```

Next in the embedded code for each of `?ButtonGroupOne`, `?ButtonGroupTwo`, `?ButtonGroupThree` and `?ButtonGroupFour` implement the following:

```
Control Events. ?ButtonGroupOne. Accepted:

GLO:GroupNumber = 1 !Button Number selected
DO RoutineProcedureStart !Show active procedure

Control Events. ?ButtonGroupTwo. Accepted:

GLO:GroupNumber = 2 !Button Number selected
DO RoutineProcedureStart !Show active procedure

etc..
```

Notice the second line: `DO RoutineProcedureStart`. Here a routine is called to update all the buttons in the ToolBox, because each time a category is selected it should call the active procedure, which is connected to this group. In the Embedded Source Procedure Routines:

```
RoutineProcedureStart
ROUTINE DO RoutineInitButton !***************** Call
Procedure ***************** IF
GLO:ArrayButton[GLO:GroupNumber,|
GLO:ArrayWindow[GLO:GroupNumber,1]),1]<>0
ProcedureStart(GLO:ArrayStart[GLO:GroupNumber,|
(GLO:ArrayButton [GLO:GroupNumber,|
(GLO:ArrayWindow[GLO:GroupNumber,1]),1]), |
(GLO:ArrayWindow[GLO:GroupNumber,1]),3],25000) !UnHide
"Button selected frame" ?GroupSelectedButton{PROP:HIDE} =
0 ELSE !Hide the "Button selected frame"
?GroupSelectedButton{PROP:HIDE} = 1 END
```

At a glance this could look like gobbledygook, but if you know how to read it, it forms a synthesis. In the first line the procedure is defined, then an other procedure is called (which you created in Part I of this article). In the third line a somewhat cryptic IF statement is started, but if you break it down it is not that difficult. The IF structure is used to determine if any button in the group has been selected. If so, the procedure is called and the `?GroupSelectedButton` is unhidden. If no button is selected, the `?GroupSelectedButton` is hidden. In pseudocode it would look something like:

```
IF ArrayButton[Group, Window, Button] <> no selection
   START(ArrayStart[Group, Button, Window,
Procedure],25000)
   UnHide ?GroupSelectedButton
ELSE
   Hide ?GroupSelectedButton
END
```

The pseudocode doesn't explain the START command but read on and you will begin to understand.

First of all there is a minor detail that has to be set in order. In part one you repositioned all the menu elements except for one, and that is the `?GroupSelectedButton`. This was postponed until now, after the introduction of the array. This code is as previously located in `RoutineInitButton`:

```
!****** Find the Y-coordinate for the selected Button
*******
CASE GLO:ArrayButton[GLO:GroupNumber,|
   (GLO:ArrayWindow[GLO:GroupNumber,1]),1]
OF 1
 GETPOSITION(?ButtonOne,,Y)
OF 2
 GETPOSITION(?ButtonTwo,,Y)

Etc..
```

This code means that within the selected group find the number of the selected button. When the number is found get the Y coordinate of this button (the X coordinate never changes). Use the Y coordinate to position the ?GroupSelectedButton exactly under the selected button:

```
!*********** Position the selection frame ************
!Reposition the "Button selected frame"
SETPOSITION(?GroupSelectedButton,,Y)
```

Dynamically Changing Procedure Calls

Until this point there haven't been any defined procedure calls for the five buttons. Under normal circumstances all that should be done is to change the button properties to make a procedure call. Unfortunately this can't be used when the procedure to be called will change dynamically.

So how do you do that?

In the Clarion language you would normally call a browse with the START command, for example:

```
START(BrowseCustomer,25000)
```

But it isn't possible to directly change the called procedure in this statement dynamically, so what do you do?

For starters take a look at some of the internal works of Clarion. Open the file

BUILTINS.CLW in the \C55\Libsrc directory, which is included in every MAP. You'll see the following procedure type declarations.

```
_PROC(),TYPE
_PROC1(STRING),TYPE
_PROC2(STRING,STRING),TYPE
_PROC3(STRING,STRING,STRING),TYPE
```

If you continue to look down the list in the BUILTINS.CLW you will find:

```
START(_PROC,UNSIGNED=0),SIGNED,PROC,NAME('Cla$START')
START(_PROC1,UNSIGNED=0,STRING),↵
   SIGNED,PROC,NAME('Cla$START1')
START(_PROC2,UNSIGNED=0,STRING,STRING),↵
   SIGNED,PROC,NAME('Cla$START2')
START(_PROC3,UNSIGNED=0,STRING,STRING,STRING),↵
   SIGNED,PROC,NAME('Cla$START3')
```

As you can see there is also four different `START` functions. Each of these forms of `START` takes a procedure as the first parameter, and those procedure prototypes are determined by the `_PROC` procedure types. Logically when programming in Clarion it appears that there is only one `START`, but depending on which parameter list is used the appropriate alias (`Cla$Start,Cla$Start1, Cla$Start2, or Cla$Start3`) will be called.

How can you use this information?

One way is to create your own function. Since the BUILTINS.CLW is included in every `MAP` the only thing you would have to do is create your own `START` alias. In the embedded source Inside the Global Map write the following:

```
MODULE('cls')
  ProcedureStart(long,long),long,proc,name('Cla$START')
END
```

Here you have just created your own function called `ProcedureStart`. This supplies the procedure call used above in `RoutineProcedureStart`. Values from the array are passed to the procedure, like this:

```
ProcedureStart(GLO:ArrayStart[GLO:GroupNumber,|
  (GLO:ArrayButton [GLO:GroupNumber,|
  (GLO:ArrayWindow[GLO:GroupNumber,1]),1]), |
(GLO:ArrayWindow[GLO:GroupNumber,1]),3],25000)
```

How does this then work? Lets look at the `MODULE` structure:

```
MODULE(sourcefile)
 Prototype
END
```

The `MODULE` structure contains the prototypes. The source file could be the name of the file containing the definitions for the different procedures just like the BUILTINS.CLW

which you looked at previously. But in this case the MODULE is being used to create your own procedure definition. This procedure calls an external library. The source file has to have a unique name which in this case is 'cls' but it could be anything you like. Now in the prototype section you duplicate the calling structure of the BUILTINS.CLW in order to make the same kind of calling prototype (START) just with a different name.

So what is the difference? In this prototype the procedure is not predefined as in the BUILTINS.CLW (_PROC, _PROC1, _PROC2, _PROC3) and therefore it is possible to assign an address instead of the name of the procedure as you have to when using the Start command.

Notice the Cla$START; if you instead used Cla$START1 it would have been possible to pass a parameter with the function, in contrast to Cla$START where it isn't possible to pass any parameters. If you want to be able to pass one parameter the module should look like this:

```
MODULE('cls')
  ProcedureStart(long,long,STRING),↵
    long,proc,name('Cla$START1')
END
```

How do you then use this new procedure? Previously you used it in this cryptic array code, but you also have to assign a procedure to call. You would have to populate the menu when it opens:

```
Window Events.OpenWindow (MainMenu)
!********* Group One *********
!Group is selected (Used for updating ToolBox menu)
GLO:GroupNumber = 1
!********** ?ButtonOne **********
GLO:ArrayStart[GLO:GroupNumber,1,|
  (GLO:ArrayWindow[GLO:GroupNumber,1]),1]| ='Customer'
GLO:ArrayStart[GLO:GroupNumber,1,|
  (GLO:ArrayWindow[GLO:GroupNumber,1]),2]|
='Customer.gif'
GLO:ArrayStart[GLO:GroupNumber,1,|
  (GLO:ArrayWindow[GLO:GroupNumber,1]),3]|
  =address(BrowseCustomer)
!********** ?ButtonTwo **********
Etc
!********** ?ButtonThree ********
Etc
!********** ?ButtonFour *********
Etc
!********** ?ButtonFive *********
!********* Group Two *********
!Group is selected (Used for updating ToolBox menu)
GLO:GroupNumber = 2
!********** ?ButtonOne **********
Etc
```

Again quite long and cryptic, but keep in mind that the array just contains ArrayStart [Group,Button,Window,Procedure]. In pseudocode it looks something like:

```
!Where I# = the 4 possible groups
LOOP I#= 1 TO 4 BY 1
  !Where J# = the 5 possible buttons
 LOOP J#= 1 TO 5 BY 1
  ArrayStart[#I, #J, Window, Text] = Button text
  ArrayStart[#I, #J, Window, Icon] = Icon name
  ArrayStart[I#, J#, Window, Procedure] =|
    address(Procedure name)
 END
END
```

The next thing to do is to select the first button in each procedure and then activate the first procedure call:

```
!*** Initial Selected Button ****
!Where I# = the 4 possible groups
LOOP I# = 1 TO 4 BY 1
 IF GLO:ArrayStart[I#,1,(GLO:ArrayWindow[I#,1]),1]|
    <>'' !Text of first Button
     !Button 1 selected
    GLO:ArrayButton[I#,(GLO:ArrayWindow[I#,1]),1] = 1
 ELSE
     !No selection
   GLO:ArrayButton[I#,(GLO:ArrayWindow[I#,1]),1] = 0
 END
END
!***** Open Group at Start****
GLO:GroupNumber = 1
DO RoutineProcedureStart
```

The last thing you would have to do in the MainMenu is to make the five selection buttons work by implementing the following code in each button:

```
Control Events. ?ButtonOne. Accepted
GLO:ArrayButton[GLO:GroupNumber,|
   (GLO:ArrayWindow[GLO:GroupNumber,1]),1] = 1
DO RoutineProcedureStart !Call procedure
```

In the first line the `GLO:ArrayButton` is assigned the button number in this case it is 1 for `?ButtonOne`.

That's it – you're finished… with the MainMenu. Now all you have to do when reusing the code is to change the initial start parameters for the five Buttons and rename the four groups, and you are in business.

Updating The Menu Buttons

After you have finished all the preparations for dynamically changing the menu buttons, you just have to put it to work.

When the program starts the MainMenu buttons are populated and the first browse is visible. When selecting one of the other menu buttons a new browse is selected. This is also the case if you select the same button twice or more. It just keeps opening a new instance of the browse window. To prevent this from happening I have included with this example a Thread Limit template which was created by following the example "Using the Template Wizatron" in the Clarion Wizatron handbook. In each procedure add this template to the procedure extensions. The folder is called Class OutLookMenu.

Each Update procedure does four things:

1. Increment window number when opened.
2. Mark first button in menu as selected (button is depressed)
3. Populate the Main menu with new selections buttons.
4. Decrement window number when closed.

For each update procedure in the `Window Events.OpenWindow`, add this code:

```
!Increment window number
GLO:ArrayWindow[GLO:GroupNumber,1] +=1
GLO:ArrayButton[GLO:GroupNumber,|
 (GLO:ArrayWindow[GLO:GroupNumber,1]),1] = 1
!********** ?ButtonOne **********
GLO:ArrayStart[GLO:GroupNumber,1,|
(GLO:ArrayWindow[GLO:GroupNumber,1]),1]| ='Customer'
GLO:ArrayStart[GLO:GroupNumber,1,|
(GLO:ArrayWindow[GLO:GroupNumber,1]),2]| ='Customer.gif'
GLO:ArrayStart[GLO:GroupNumber,1,|
(GLO:ArrayWindow[GLO:GroupNumber,1]),3]|
=address(UpdateCustomer) !********** ?ButtonTwo
**********
```

And so on for the five buttons, and finish the code off with:

```
!Update the ToolBox menu.
POST(EVENT:RefreshToolBox,,GLO:ToolBoxThread)
```

In the Window Events.CloseWindow:

```
!Decrement window number in Category
GLO:ArrayWindow[GLO:GroupNumber,1] -=1
!Update ToolBox menu.
POST(EVENT:RefreshToolBox,,GLO:ToolBoxThread)
```

So how does this work? Well, each update procedure defines a new menu structure, where the first button is the update procedure and buttons two to five are browse procedures. All the browse procedures, which have some kind of connection to the update procedure, are managed in the MainMenu. Therefore there aren’t any tabs containing browse boxes. To include them in the procedure tree structure you have to select the Procedure button for each update procedure and mark the browse procedures, which are included in the MainMenu.

No matter which button you select within the group the MainMenu won’t change until a new Update procedure is opened or closed. A nice little twist is keeping track of which

button is selected. For example when you select a browse, say button three, and then open the Update procedure, the first button which contains a procedure call to this update procedure is automatically selected. Close the update procedure and now button three is selected and the corresponding browse procedure has focus. All this works fine until you close a browse procedure instead, because the browse procedure does not keep track of the window number. To prevent the user from closing the browse procedure and for that sake resizing, iconisizing and restoring it:

Local Objects. ThisWindow.TakeWindowEvent.CODE.Top of CYCLE/BREAK support

```
OF EVENT:CloseWindow !The user is closing the window
   CYCLE !Prevent users from closing the window.
OF EVENT:Size !The user is resizing the window
   CYCLE !Prevent users from resizing the window.
OF EVENT:Iconize !The user is minimizing the window
   CYCLE !Prevent users from minimizing the window.
OF EVENT:Restore !The user is restoring the window.
   CYCLE !Prevent users from restoring the window.
```

Also if the Browse procedure has a Close button it should be deleted. Just remember that if the close is included in the override control strategies, it should also be deleted there.

Where To Go From Here

As I mentioned in the start the To Do list can be extended quite a bit. Here are some ideas, which I have not explored yet:

- The OutLook menu has to be able to coexist with the existing menu bar. In other words procedures selected in the menu bar also have an impact on the OutLook menu, which has to change its menu structure depending on which procedure is selected from the ordinary menu bar.
- The idea with the menu is to have one instance of any given browse procedure. In the example you have just been through the different browse procedures always had the same record filter, range limit field and the range limit type. Under normal circumstances these values changes in accordance to the calling procedure and should therefore also be applied to the functionality of this OutLook Menu.
- One could also change the code so the fourth group could be used for reports. So when a given window has focus the user can select the reports group and see which report possibilities there is. Personally I haven't thought this idea through, but you would have to keep track of the window in a specific group and which reports it contains in each selection button.
- Add more buttons and make the window, which contains the buttons scroll when necessary. Although I won't recommend adding too many buttons as the user has to look for them. For example with five buttons the user can make the selection at once, but if there are eight buttons or more the user has to do a little more looking.
- Extend the menu programming to exclude groups which are not necessary. This could mean excluding unnamed groups, just like the selection buttons. You would also have to make sure that the repositioning of any following groups is done accordingly.
- Extend the menu programming to include extra groups in the tabs located below the menu. If there aren't any groups to include the tabs should be hidden and the

repositioning of the group is done accordingly. The extra groups just extends the group numbering from 4 to 5, 6, 7…

- Create an OutLook menu template.
- Create an OutLook menu Wizatron.

And so on. You can continue the list and let me know of your progress.

When I started this article I thought that I pretty much had finished the menu development. As it turned out I had just covered the tip of the iceberg, and new ideas changed the development in another direction than I had originally anticipated. That first idea of copying the OutLook menu has grown into a menu that controls the whole application and at any given moment provides the user with exactly the menus needed.

Download the example app

Download an updated app with menu templates

Steffen S. Rasmussen has graduated in Computer Science from Copenhagen Business College. Since then he has worked as a programmer, system technician and network administrator, and is currently IT manager. Clarion is a quite a new language to Steffen since his only been working with it since January 2000. But what better way to learn it than by trying to teach others! Steffen has also set up a web site to collect as many examples of different user interfaces as possible to inspire Clarion developers.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

The Clarion Advisor: Debugging With DebugView

by Jeff Slarve and Dave Harms

Creating An MS OutLook-Style Menu In Clarion: Part 2
(Sep 5,2000)

The Clarion Advisor: Better Debugging With DebugView
(Sep 5,2000)

Debugging is a favorite topic among Clarion developers, and in the past Clarion Magazine has published a number of articles on debugging techniques. (To see these articles, use any Search button to look for the word *debug*.) Probably the most-used method is displaying a message during execution using either the `MESSAGE()` or `STOP()` function. These both have the disadvantage of disrupting the program's events, and if you're trying to solve a user interface problem there's a good chance the debugging process will alter the problem, making things even more difficult.

One way around this is to write debug messages out to a log, and the DebugView utility from Systems Internals makes this easy. Go to http://www.sysinternals.com/ and download a version suitable for the version of Windows you're using. DebugView will capture messages written to the Windows `OutputDebugString` API call and display them in a topmost window.

Listing 1 shows an example of a 32 bit program which uses the `OutputDebugString` call to write several messages to the DebugView window.

Listing 1. A DebugView test program

```
program

   map
     module('')
       OutputDebugString(*CString),raw,pascal,↵
         Name('OutputDebugStringA')
     end
     Debug(String DebugString)
   end

  Code

?    Debug('This is test1')
?    Debug('This is test2')
    message('Yada')

Debug Procedure(String DebugString)
DB  CString(500)
```

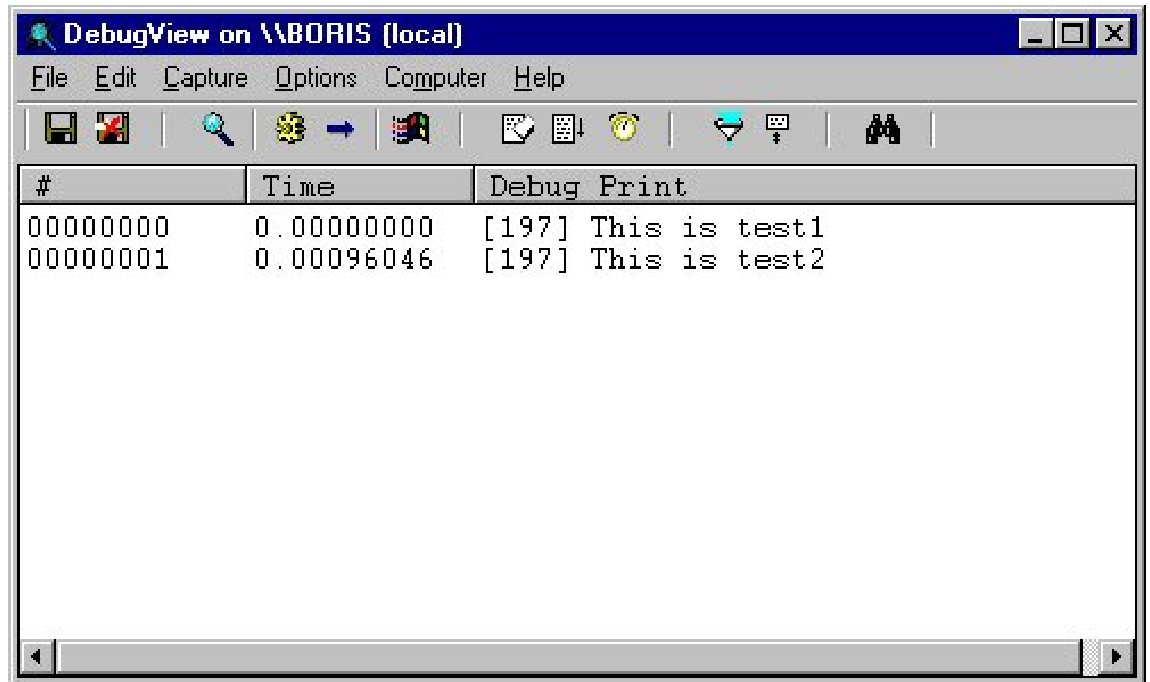
```
Code

DB = Clip(DebugString) & '<13,10,0>'
OutPutDebugString(DB)
```

Note the use of the ? character in column 1 to conditionally compile statements only when debug is on.

Create a project for this source (or use the demo application), compile, run the DbgView.exe, then run the example app. Figure 1 shows the resulting messages displayed in the DebugView window.

Figure 1. The DebugView message window.



With DebugView you can view and record debug session output on your machine or across the Internet, which raises interesting possibilities for beta testing. You can also search and filter debug output, print or log to a file, and format the output. Best of all, DebugView is free! And while you’re at the SysInternals web site be sure to have a look at their other system utilities.

If you’re using ABC, you may also want to try modifying your output string to look like this (assuming "ABC" is the name of the application):

```
? OutPutString = Clip('ABC ' & GlobalErrors.GetProcedureName())|
   & ' - ' & Clip(DBString) & '<13,10,0>'
```

As long as the main EXE contains the lib for all of the DLLs, then all of the DLLs will use the same globalerrors object.

On a related note, if you need to know if you are running under the debugger, you can try putting this function in the map:

```
IsDebuggerPresent(),BOOL,pascal,Name('IsDebuggerPresent')
```

You’ll have to make a lib from kernel32 for this, but `OutputDebugString()` is already in the Win32 API prototypes.

Download the example app

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked Questions

Site Index
Article Index
Author Index
Links To Other Sites

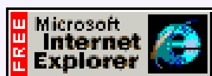
Downloads
Open Source Project
Issues in PDF Format
Free Software

Advertising

Contact Us

TopSpeed

Clarion5 by TopSpeed



The Nuts And Bolts Of Passing Parameters

by James Cooke

The Nuts And Bolts Of Passing Parameters: Part 1
(Sep 12,2000)

Five Rules for Managing Complexity: Part 3
(Sep 12,2000)

Outlook Menu Templates
(Sep 12,2000)

Part 1 of 2

Any time you divide a program up into procedures, you need a way for those procedures to communicate. There are several ways to accomplish this, including the use of global variables, data files, .ini files, module data and parameter passing. The pros and cons of each are dependent on the language, the user's experience and the environment. This article will discuss the basics and benefits of "passing" data across to another procedure.

Why Bother?

The use of global variables is a common alternative for parameter passing, and you might ask why this is not such a good idea. Simply put, global variables clutter up an application. They use up memory and the variables might be unexpectedly modified by other procedures, which makes the procedure using the variable, and by extension the output of that procedure, suspect. Using global variables for this purpose is probably the biggest culprit in producing spaghetti code, and is one of the hallmarks of newbie programmers!

On the other hand, passing parameters lends itself to efficient code reuse and clean encapsulation of data (that is, no contamination of data by external code). It also partially documents the functionality of a procedure - you can look at the data labels being passed to a procedure and the return value(s) and often guess its functionality. Parameter passing is also the standard used by most other programming languages, and understanding how Clarion uses parameter passing will ease later understanding of things like the use of Windows API functions. In addition, these procedures can be shared across multiple applications by creating a DLL or Library, thus extending their functionality to other applications.

There are several techniques in parameter passing, and I will discuss them as follows:

1. Passing a value to a procedure
2. Omittable Parameters
3. Passing Parameters to Threaded Procedures

4. Using Return Values
5. Optional Return Values
6. Passing Values By Reference
7. Passing Objects
8. Local Procedures
9. Passing Complex Structures
10. Passing Reference Parameters To The Windows API
11. Function Libraries

Passing A Value To A Procedure

Passing a value to a procedure essentially means that a local variable is declared in the receiving procedure and is primed to a particular value. Here's an example:

A procedure called `ProcA`, when called, needs to be given three values by the procedure that called it. These values will be of the data types `STRING`, `LONG`, and `BYTE`. To call this procedure and send over this data, the call would be n this format:

```
ProcA('MyText',12345,10)
```

Alternatively, using variable labels instead:

```
ProcA(StringToSend,LongToSend,ByteToSend)
```

On the other side, the called procedure needs to expect the passed data when it is called. To tell Clarion that the procedure `ProcA` will be receiving these variables, or parameters, the procedure's prototype is declared as follows:

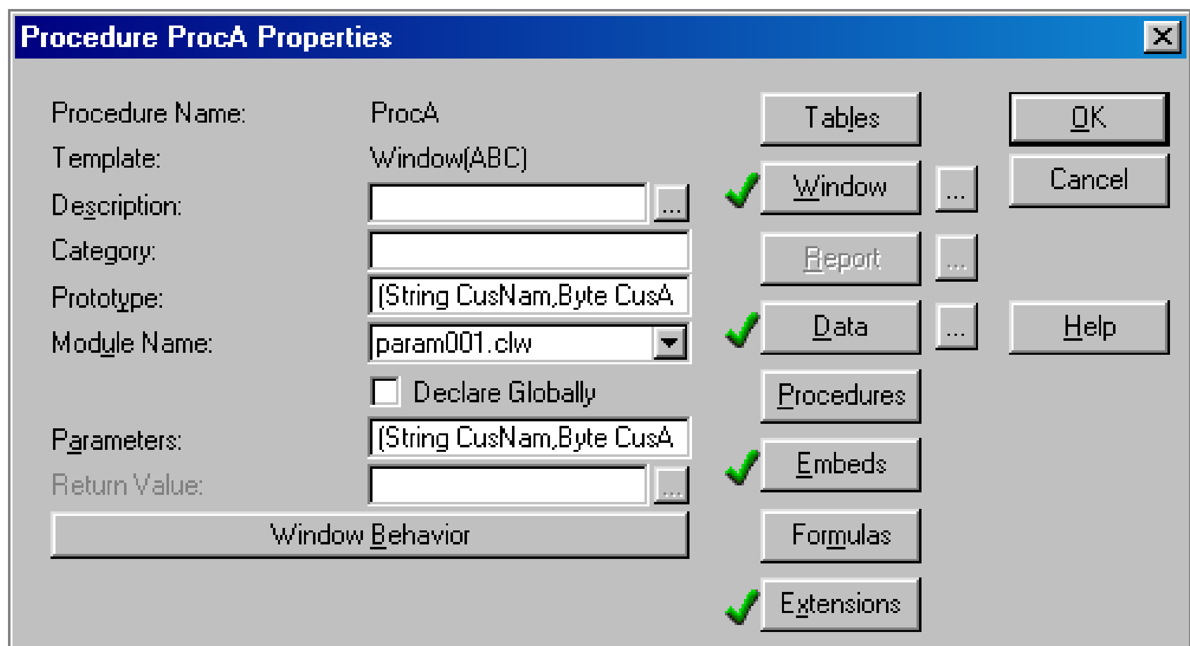
```
ProcA (STRING CusNam,BYTE CusAge,LONG CusNum)
```

The procedure declaration is, in this case, identical:

```
ProcA PROCEDURE (STRING CusNam,BYTE CusAge,LONG CusNum)
```

This means that the procedure will be expecting to receive three variables of type `STRING`, `BYTE` and `LONG`, and the variables will be called `CusNam`, `CusAge`, `CusNum` respectively. Since most programs are not hand coded but exist in an .APP file, it is necessary to get Clarion to generate the above code correctly. To do this, modify the procedure properties, as shown in Figure 1.

Figure 1. Prototyping a simple call in Clarion



In order to access the data, simply use the variables `CusNam`, `CusAge` and `CusNum` as if they were already declared in the Local Data section. So placing a button on the window of `ProcA` with the following code in the Accepted embed will work fine:

```
MESSAGE('Customer number '& CLIP(CusNum) & ' is named ' & |
  CLIP(CusNam) & ' and is ' & CusAge & ' years old!')
```

It is important to note that the data that is received by the procedure `ProcA` is a *copy* of the data that was sent, so modifying the received data will not modify the data in the procedure that called it. Not that it can't be done - it just won't happen using this particular prototype.

To view the implementation of this technique, open the example application and select the toolbar Pass Parameters button, set the variables to pass, and click the button. The procedure will start up and the values in the window will be correct - just as if the variables you were using were global!

That's the first step in uncluttering an application. Now try removing one of the parameters of the function call in the example. For example, try calling this procedure but leave out the CusNum parameter:

```
ProcA(StringToSend, ByteToSend)
```

Compiling this will produce the compiler errors "No matching prototype" and "Too few parameters." But what if one of the parameters is not required? Is it possible to make a function work correctly even if an unneeded parameter is missing? Yes it is!

Omittable Parameters

Omittable parameters allow a procedure to take on a different behavior dependent on how it has been called. This kind of behavior, called polymorphism, provides great flexibility in the language. An example of this is Clarion's `MESSAGE()` function: you can specify a different message box appearance simply by omitting the second or third parameters. Figure 2 shows how the same function can have different results, all by simply adding an extra parameter to the message call. The respective instructions for each popup are as follows:

```
MESSAGE('Hello World!')
MESSAGE('Hello World!','Greetings!')
MESSAGE('Hello World!','Greetings!',Icon:Exclamation)
```

Figure 2. Using the message function passing one, two and three parameters respectively

Comparing the simple implementation of the `MESSAGE()` function with the more complex output indicates that there must be some internal logic in `MESSAGE()` that determines how it should react when a parameter is omitted or included. This same functionality can also be utilized in your own function calls by telling the Clarion compiler that you want to be able to allow the programmer to omit parameters of a procedure. To do that, place the omittable parameters and variable names in <angle brackets>.

Internally, Clarion's MESSAGE() function is prototyped as follows:

```
MESSAGE(STRING,<STRING>,<STRING>,<STRING>,↵
  UNSIGNED=0,BOOL=FALSE), UNSIGNED,PROC
```

This means that Clarion's message procedure *must* receive between one and four STRINGs, an UNSIGNED and a BOOL when it is called. (BUILTINS.CLW still uses the older declaration style of only supplying the data types in the prototype, rather than the more readable type/name pair.)

In the previous example, the parameters passed to `ProcA` were:

```
(STRING StringToSend,BYTE ByteToSend,LONG LongToSend)
```

Firstly, prototype the function as below:

```
ProcB PROCEDURE(STRING CusNam,Byte CusAge,<LONG CusNum>)
```

The procedure declaration is:

```
ProcB PROCEDURE (STRING CusNam,BYTE CusAge,<LONG CusNam>)
```

Now calling the procedure with an omitted parameter will compile successfully - but it still won't do anything! The logic in the body of the function needs to be adjusted to be able to react correctly according to the number of parameters it received. Clarion determines which parameter is missing by means of the `Omitted()` function.

This code will implement the logic:

```
If OMITTED(3)
MESSAGE('Customer Name ' & CLIP(CusNam) & | ' is ' &
CusAge & ' years old!',| 'The Age was not included') Else
MESSAGE('Customer number '& CLIP(CusNum) & | ' is named '
& CLIP(CusNam) & ' and is ' | & CusAge & ' years old!')
End
```

There is a little bit more to this though: Notice that the omittable parameter was the last parameter in the parameter list. What if the omittable parameter were to be the *first* parameter instead? The prototype would be as follows:

```
ProcB PROCEDURE(<STRING CusNam>,BYTE CusAge,LONG CusNum)
```

The procedure declaration would be:

```
ProcB PROCEDURE (<STRING CusNamCusNam>,CusAge,LONG CusNam)
```

To successfully call `ProcB` with two parameters, the call would have to look like Figure 3.

Figure 3. Using a placeholder to mark a missing parameter.

The placeholder comma circled in red indicates to the compiler that the single omitted character was intentional and also to indicate *which* parameter was omitted. This is essential in the case where there may be more than one omittable parameter in the function call.

The two-parameter call to `ProcA` did not need the placeholder comma for two reasons: Because the omittable parameter is the last parameter in the parameter list; and there was only one omittable parameter. If there are multiple omittable parameters and the function call omits one parameter only and does not show any placeholder commas, then the compiler will assume that the last omittable parameter in the parameter list is the missing one and could execute the wrong piece of code. Otherwise, the placeholder comma clears up any confusion as to which parameter is missing and Clarion can execute the right piece of code in the procedure.

One more thing; you probably noticed that the `ProcB` procedure is not a window procedure but a source procedure. I did this because the `Omitted()` function will only work with the current procedure scope. In an ABC application most code is executing as part of a class method. For instance, placing the code in the Accepted embed point of a button will make OMITTED() look for omitted parameters in the ABC `TakeAccepted` method, and not the main `ProcB` procedure. To trap for omitted parameters in an ABC window procedure the `Omitted()` test must be done after the main procedure's *CODE* statement but before the statement `GlobalResponse = ThisWindow.Run()`. (Tip: You will have to use the `Omit` directive to make that happen as there isn't an appropriate embed point.)

Passing Parameters To Threaded Procedures

In the context of this article, starting a procedure in a thread essentially means that the

called procedure does not necessarily retain exclusive focus after it has been called. The calling procedure can regain focus if the user clicks on its window with a mouse, for example. Threading is a very powerful feature in Windows, but until recently it was not possible to pass parameters to a threaded Clarion procedure via the `START` function. This often resulted in the complex and excessive use of global variables to "pass" values.

The only data type that can be passed to a threaded procedure is a string, it must be passed by value, and a maximum of three parameters can be passed to it. No value can be returned to the calling procedure. (Editor's note: Jeff Slarve has devised a workaround for this limitation in `START`. His thread manager allows you to pass any variable or object you like to a `STARTed` procedure.) To prototype a procedure to be called in a thread, declare it the same as a standard procedure, except that it can only have up to three string parameters and cannot return a value:

```
ProcD PROCEDURE(STRING CusNam)
```

The procedure declaration would as per normal:

```
ProcD PROCEDURE (STRING CusNam)
```

To call a procedure under a thread, the call is made in the same way a procedure is usually `STARTed`, with the second parameter the size of the stack allocated to the `STARTed` procedure (and any procedures it calls):

```
Start(ProcD,25000)
```

To pass a string parameter to this procedure, the call is made as follows:

```
Start(ProcD,25000,StringToSend)
```

The `ProcD` procedure will receive the parameter in the same fashion as in an unthreaded call.

It's all very well to be able to pass parameters to functions to make them *do* specific things elsewhere, but what about the *results* of what gets done? It might be acceptable for the procedure to write something to disk or to set a global variable, but relying on that exclusively is rather limiting. It would be very useful for a function to be able to go and do something specific, or evaluate a condition, and optionally pass a specific value back to the calling procedure. This is the purpose of *return values*, and I'll go into that in detail in Part 2.

Download the example application

James Cooke has been using Clarion since 2.1 days and has been a die hard for "the cause" ever since. He and his family recently moved from South Africa to Texas and is currently working in the banking industry. He spends most of his free time basking in the sun by the pool with a good book or succumbing to that hard-to-kick addiction that persistently haunts the Western cosmopolitan neighborhoods - the yard sale.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

TopSpeed

Clarion5 by TopSpeed

FREE Microsoft Internet Explorer

etc2000 EVENT SPONSOR

Five Rules For Managing Complexity

The Nuts And Bolts Of Passing Parameters: Part 1
(Sep 12,2000)

Five Rules for Managing Complexity: Part 3
(Sep 12,2000)

Outlook Menu Templates
(Sep 12,2000)

by Tom Ruby

Part 3

In this series, I've introduced the idea of using rules to reduce the complexity of your work. Software complexity is increasing because the users and clients expect your software to solve increasingly complex problems for them, and if you don't do it, your loyal customers will beat a path to your competitor's door. Face it, the days when the users were delighted with a 2,000 line program with two tables, a browse and a report are far behind.

How do you handle this complexity? The first thing to do is avoid making it worse. Clarion is a powerful tool for solving problems, but let's apply its power to solving the user's problems, not to solving problems we add to them.

So I'll recap Rules 1 and 2, along with the guidelines, and get on with Rule 3.

> *Guideline 1:*
> Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.
>
> *Guideline 2:*
> It is a lot less painful to fix a painful mistake now than it will be later on.
>
> **Rule Number 1:**
> Eliminate repeating fields.
>
> *Guideline 3:*
> A list is resizable, a form is not.
>
> *Guideline 4:*
> Link your tables by a hidden field that is completely meaningless outside the system.
>
> *Guideline 5:*
> Use keys to help the application identify records it is interested in.
>
> **Rule Number 2:**

Eliminate redundant data

If you stick to Rules 1 and 2 when you lay out your data, you will find a lot of the complicated code you're used to writing disappears. There's something else that will disappear; bugs. I mean the type of bug where the users complain, "I changed this here and something went wrong over there…" You like the idea of less headache, fewer bugs and less tangled code? Then I'll go on with Rule Number 3:

Rule Number 3:
Eliminate Columns that don't belong

To explain Rule Number 3, I'll talk a little about Keys. In Clarion, we have the idea of a key and an index rather mixed up because you usually want an index on a key. A key is a column (or field) in a table (or file) which is used to identify rows (or records), while an index is a data structure used to impose an order on the rows without having to actually sort them to make finding records faster. You usually want an index on a key, but you don't have to have one. A key is a logical construct, an index a physical construct.

Now, when I talk about "The Key," or "The Primary Key," what I mean is a column or columns that uniquely identify each row in the table. Even more than that, I mean a column or a value that represents the thing the row of the table records. The other columns of the table describe the thing the KEY represents. Take a look at an example:

Student Table

StudentID	Name	Gender	Teacher	Grade	Section
102	Caleb	M	Wilson	2	1
103	Cameron	M	Wilson	2	1
104	Anthony	M	Wilson	2	1
105	Angel	F	Wilson	2	1
106	Sarah	F	Wilson	2	1
107	Ethan	M	Harn	1	1
108	Mandy	F	Harn	1	1
109	Waylon	M	Wilson	2	1
110	Emily	F	Harn	1	1
111	Seth	M	Harn	1	1
112	Ashley	F	Hard	1	1
112	Drew	M	Wilson	2	1

Obviously, the StudentID is the primary key in the Student table. Remember, the primary key *represents* the student. The other columns *describe* the student. The table also contains the student's name, gender, teacher, grade, and section The section must be which class of the grade they go to. But wait! Three of these columns describe the class, not the student, so they don't belong here. They belong in a class table. So I'll make two tables:

StudentTable	**ClassTable**
StudentID Name Gender ClassID	ClassID Teacher Grade Section

Why is this better? For one thing, moving a student from one class to another is a matter of updating one field, not three. Also, you don't have to worry about things getting out of synch, or the users asking, "If Drew is in 2nd grade why does the dataabase show his teacher as Mrs. Harn, the first grade teacher? Uh oh, Mrs. Bricker will be teaching second grade. With a single table, some user would have had to find and update 7 records, or you would have to write a change teacher process. With a separate class table, only one record needs to be updated.

To clarify this, here's Guideline 5:

> *Guideline 5:*
> The primary key represents the "thing." The rest of the columns describe the thing the primary key represents.

You probably have more information you want to store about the teacher. Perhaps the teacher's phone number, address, or state certificate number. As long as one teacher only teaches one class, you could probably put these in the Class table, which would change it into a teacher table. If you're talking about a bigger school or older kids where a teacher might teach several classes, you would want to make separate teacher tables and class tables. These tables might look like this:

StudentTable	**ClassTable**	**TeacherTable**
StudentID StudentName StudentAddress GradeLevel	ClassID TeacherID RoomNumber Period CourseName	TeacherID TeacherName CertificateNumber TeacherPhoneNumber

So how do you record which class the student is in? It is pretty likely that a student is in more than one class. A class also has more than one student. This is the dreaded Many-To-Many problem. You can't put a list of classes in the Student table because that would violate Rule Number 1, and you can't put a list of students in the class table because that would also violate Rule Number 1. Fortunately, the conundrum is easily solved.

"You don't mean another…" Yes. I mean another table.

AttendsTable
AttendsID StudentID ClassID Grade

Now you can record that Sasha is taking Biology, Algebra 1, Wood Shop, Art, English

and Civics. You also have a place to record what grade she got in each of these classes. Notice that I gave the AttendsTable an AttendsID. This isn't actually needed, because the table's primary key could be StudentID and ClassID. I gave it a single field primary key for two reasons. First, force of habit. Second, if you discover later on you need a list of something from the attends table, maybe an attendance or assignment history, you have a single linking field to use, which is easier to use in a range limit than something like HST:StudentID = ATT:StudentID AND HST:ClassID = ATT:ClassID. When you're building an application, you never know what is going to happen to the requirements for the application, so while you're at it, you might as well simplify the work you'll have to do later on. This is so important that I'll make it a guideline:

> **Guideline 6:**
> You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

You might wonder how you're going to show a student's schedule, or a teacher's class list, with all the fields scattered all over the place. Whenever you make a report or even a browse, the templates make a view structure for you, and a view serves to temporarily present you with an unnormalized picture of your data, much like the original student table.

So, to recap. Here are the first three rules of data normalization:

> **Rule Number 1:**
> Eliminate repeating fields.
>
> **Rule Number 2:**
> Eliminate redundant data
>
> **Rule Number 3:**
> Eliminate Columns that don't belong

And here are the guidelines:

> *Guideline 1:*
> Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.
>
> *Guideline 2:*
> It is a lot less painful to fix a painful mistake now than it will be later on.
>
> *Guideline 3:*
> Link your tables by a hidden field that is completely meaningless outside the system.
>
> *Guideline 4:*
> Use keys to help the application identify records it is interested in.
>
> *Guideline 5:*
> The primary key represents the "thing." The rest of the record describes the thing the primary key represents.
>
> *Guideline 6:*
> You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

For most situations, 3 rules, or "3rd Normal form" is considered "normal enough," but

there are big advantages to understanding Rules 4 and 5. Next time, Rule 4.

Tom Ruby, who is no relation to the man who shot Lee Harvey Oswald, is an independent contractor living in the middle of a hayfield in Central Illinois with his wife Susan and two red-headed sons, Caleb and Ethan. He has been using Clarion for Windows since the summer of '95. Before that, he was a "TopSpeeder" using Modula II, so he has never used the DOS versions of Clarion.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

TopSpeed

Clarion5 by TopSpeed

FREE Microsoft Internet Explorer

etc2000 EVENT SPONSOR

Using CHOOSE() To Concatenate Data

by Carl Barnes

Using CHOOSE() To Concatenate Data
(Sep 15,2000)

The Nuts And Bolts Of Passing Parameters: Part 2
(Sep 15,2000)

Five Rules for Managing Complexity: Part 4
(Sep 15,2000)

Five Rules for Managing Complexity: Part 5
(Sep 15,2000)

September 2000 News
(Sep 15,2000)

Steven Parker's recent article Standard Address Handling showed a lot of classic Clarion code for concatenating together name and address strings to build a `TEXT()` control string without blank lines. This is generally done with a lot of `IF` statements as shown in the code below:

```
NandA = CLIP(MBR:FirstName) & |
  CLIP(' '&MBR:MiddleName) |
  &' '& CLIP(MBR:LastName)
IF MBR:Address1
  NandA = CLIP(NandA) & '<13,10>'|
  & MBR:Address1
END
IF MBR:Address2
  NandA = CLIP(NandA) |
   & '<13,10>' & MBR:Address2
END
IF MBR:City & MBR:State |
 & MBR:ZIPCode
  NandA = CLIP(NandA) & '<13,10>'|
   & CLIP(MBR:City) |
   & CLIP(' '&MBR:State) |
   &' '& MBR:ZIPCode
END
```

An alternative is to use the `CHOOSE()` statement. Most programmers think of `CHOOSE()` as the inverse function to `INLIST` with the syntax `CHOOSE(IndexNumber, Value1, Value2, Value3, ...)`. For example, to return the name of the day of the week:

```
CHOOSE(Date%7+1,'Sun','Mon','Tue','Wed','Thu','Fri','Sat')
```

`CHOOSE()` also has a syntax that allows it to evaluate logical conditions: `CHOOSE(condition ,true-value, false-value )`. The condition is

evaluated, and if it is true `CHOOSE()` returns the first value, otherwise it returns the second value. Using this syntax the above concatenation code can be written as:

```
NandA = CLIP(MBR:FirstName) & CLIP(' '&MBR:MiddleName)|
  & ' '& CLIP(MBR:LastName) |
  & CHOOSE(MBR:Address1='','','<13,10>' |
  & CLIP(MBR:Address1)) |
  & CHOOSE(MBR:Address2='','','<13,10>' |
  & CLIP(MBR:Address2)) |
  & CHOOSE(MBR:City & MBR:State |
  & MBR:ZIPCode='','', '<13,10>' & CLIP(MBR:City) & |
  CLIP(' '&MBR:State) &' '& MBR:ZIPCode)
```

You can see from the code that all of the IF statements are replaced by `CHOOSE()` statements. The first `CHOOSE(MBR:Address1='','','<13,10>'&CLIP(MBR:Address1))` will evaluate the condition "`MBR:Address1=''`" and return blank if the address is blank. If the address is not blank the condition will be false and `CHOOSE()` will return the second value which is a `CR,LF` and address field clipped. Do this for all of the fields and you'll have the string that Steve required.

Choose Your CHOOSE()

A potential bug to watch out for in doing this is to make sure you have a "condition" when using this form of `CHOOSE()` – it has to be a Clarion language statement that returns a true or false value. You should typically write "`IF MBR:Address1`" as the shorthand for "`IF MBR:Address1<>’’`" to test if a value is not blank. (This not only saves keystokes, the compiler will optimize this and write smaller and faster object code. The same is true for "IF ~Variable" instead of "IF Variable=’’".) However, if you wrote "`CHOOSE(MBR:Address1,'t','f’)`" the compiler would evaluate that as an expression using the syntax "`CHOOSE(expression ,value1, value2 [,value3...])`" and not give you the desired result. Since the condition would normally be a string it would always evaluate to the number zero and `CHOOSE()` would always return the false value since zero is not a valid value.

If you would like to avoid typing the equal sign in a condition you can type "`CHOOSE(~MBR:Address1,'blank','not blank’)`". Or to simulate "`IF MBR:Address1`" you can use a double negative "`CHOOSE(~~MBR:Address1,'not blank','blank’)`". Below is the code written again using this syntax. Personally I like the "tilde syntax" since when reading left to right I know immediately I am reading a conditional `CHOOSE()`.

```
NandA = CLIP(MBR:FirstName)&CLIP(' '&MBR:MiddleName)|
  & ' ' & CLIP(MBR:LastName) |
  & CHOOSE(~MBR:Address1,’’,'<13,10>'&CLIP(MBR:Address1)) |
  & CHOOSE(~~MBR:Address2,'<13,10>'&CLIP(MBR:Address2),'') |
  & CHOOSE(~MBR:City & MBR:State & MBR:ZIPCode, '', |
  '<13,10>' & CLIP(MBR:City)  |
  & CLIP(' '&MBR:State) &' '& MBR:ZIPCode,)
```

Another potential bug to watch out for is forgetting the second value (or false value) and typing "`CHOOSE(Condition,'true-value')`". The compiler will not spot the error

and the `CHOOSE()` will always return zero. So if you see a persistent zero in your output you'll want to look for a badly coded `CHOOSE()`.

Performance And Code Size

Many times everything is a compromise. In my opinion I like the `CHOOSE()` syntax over the classic IF syntax. I think the code is faster to write, easier to read and easier to modify. This should result in less bugs. But I do not want compromise performance and have my program run significantly slower or be larger just so the code is easier for me to write. In my analysis the `CHOOSE()` method is faster and a little smaller than the `IF` method. It's not that `CHOOSE()` is faster then `IF`, its eliminating all of the "`NandA = CLIP(NandA)`".

For my performance analysis I took the IF and `CHOOSE()` code examples above and ran them through 500,000 iterations. This was compiled 32-bit using C5. The field `MBR:Address2` was always blank and rest of the fields had reasonable values. I also compared the two ways of writing the `CHOOSE()` conditional and found the "`CHOOSE(~Variable,,)`" to be the fastest and smallest.

No.	Code Method	Time (Sec)	% Faster	Code Bytes	% Smaller
1	Multiple `IF..THEN`	9.75	0%	540	0
2	`Choose(Address='',,)`	7.63	27.8%	522	3.4%
3	`Choose(~Address,,)`	7.54	29.3%	501	7.8%

The fastest code, based on alternative number three above, is shown below. This code has been modified since the performance test above and improves from 29.3% to 45.3% faster. The final `CHOOSE()` has been changed from "`CHOOSE(~MBR:City & MBR:State & MBR:ZIPCode,,)`" to what you see below. Changing from two concatenates and one conditional to three conditionals saves significant time.

```
NandA = CLIP(MBR:FirstName)&CLIP(' '&MBR:MiddleName)&' '|
  & CLIP(MBR:LastName) |
  & CHOOSE(~MBR:Address1,'','<13,10>'|
  & CLIP(MBR:Address1)) |
  & CHOOSE(~MBR:Address2,'','<13,10>'|
  & CLIP(MBR:Address2)) |
  & CHOOSE(~MBR:City AND ~MBR:State AND ~MBR:ZIPCode, '', |
  '<13,10>' & CLIP(MBR:City) |
  & CLIP(' '&MBR:State) &' '& MBR:ZIPCode )
```

One way *not* to write the above code is to move the `CLIP()` outside the `CHOOSE()` statement as shown below. I think this code looks nicer, but this code causes the expensive `CLIP()` to occur even when the field is blank slowing down the above code from 45.3% faster to just 35.4% faster.

```
A Bad Idea: "CHOOSE(~MBR:Address1,'','<13,10>') |
  & CLIP(MBR:Address1) & ".
```

Clipping Tricks

I discovered a concatenating CLIP() trick, shown above in building the "First M Last" name string. The code "CLIP(' '&MBR:MiddleName)" returns an empty string if there is no middle name so the full name does not have a double space in it. If there is a middle name then the space does get in front of it. If you would only like a middle initial then you can choose to do that as shown below:

```
NandA = CLIP(MBR:FirstName) |
 & CHOOSE(~MBR:MiddleName[1],' ',MBR:MiddleName[1]|
  &'. ') & CLIP(MBR:LastName) . . .
```

More CHOOSE() Syntax

CHOOSE() does have one other seldom used syntax and that is CHOOSE(condition). If the condition is true this returns 1, otherwise it returns 0. So it is a shorthand form of CHOOSE(condition,1,0). This has nothing to do with concatenating but I thought I would mention it. Clarion does not have a SGN() function but you could use CHOOSE(MyNumber>0) to get close as long as you do not care about differentiating between zero and negative numbers. A correct SGN(x) function returns –1, 0 or 1 depending on if a number is less than, equal to or greater than zero.

Summary

One final point is that CHOOSE() works with EVALUATE() while IF statements are not permitted (although you can of course bind a function to use statements EVALUATE() doesn't support directly). I hope you'll consider using CHOOSE() instead of nested IFs for building your concatenated strings. I think it makes for more readable code which compiles slightly smaller and performs faster. One other tip to remember is that CLIPs are expensive, especially on long strings.

Carl Barnes is an independent consultant working in the Chicago area. He has been using Clarion since 1990, is a member of Team TopSpeed and a TopSpeed Certified Support Professional. He is the author of the Clarion utilities CW Assistant and Clarion Source Search.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked Questions

Site Index
Article Index
Author Index
Links To Other Sites

Downloads
Open Source Project
Issues in PDF Format
Free Software

Advertising

Contact Us

TopSpeed

Clarion5 by TopSpeed

FREE Microsoft Internet Explorer

etc2000 EVENT SPONSOR

The Nuts And Bolts Of Passing Parameters

by James Cooke

Part 2 of 2

Using CHOOSE() To Concatenate Data
(Sep 15,2000)

The Nuts And Bolts Of Passing Parameters: Part 2
(Sep 15,2000)

Five Rules for Managing Complexity: Part 4
(Sep 15,2000)

Five Rules for Managing Complexity: Part 5
(Sep 15,2000)

September 2000 News
(Sep 15,2000)

In Part 1 of this article, I explained how to get data into a procedure by passing parameters. In many cases, that's only half the battle. You also want to find out something that happened after you send that data into the procedure.

Using Return Values

Return values allow output of a procedure to be treated as a variable. A simple example of Clarion's use of a return function is the Clock() function. Consider the following code:

```
TheTime        LONG
FormattedTime  STRING

   Code
   TheTime=Clock()
   FormattedTime=Format(TheTime,@T1)
   MESSAGE('The time is ' & FormattedTime)
```

> **NOTE:** Historically, in Clarion the keyword PROCEDURE was used when declaring a procedure that didn't return a value, and FUNCTION when the procedure did return a value. The function keyword has been deprecated and is now always treated in Clarion code as a synonym for procedure.

In this example, a variable is directly assigned the result of Clarion's internal Clock() function - as if the Clock() function itself were a variable. The variable is then formatted using the returned value of the Format() function and then displayed using the MESSAGE() function.

This code is powerful but cluttered - it can be simplified by nesting the functions as shown below:

```
MESSAGE('The time is ' & Format(Clock(),@T1))
```

To declare a procedure to return a value prototype the procedure as per normal except a comma and the datatype of the return value is appended at the end of the prototype statement:

```
ProcC PROCEDURE(BYTE CusAge,LONG CusNum) ,STRING, PROC
```

As you can see, the only time prototypes and procedures differ in their format is when a value is returned. The procedure does not declare the return type:

```
ProcC PROCEDURE (BYTE CusAge,LONG CusNam)
```

All the logic in the declared procedure could be the same logic as any other procedure - the only thing that would change is that you need to use the `return()` statement to return the value to the calling procedure. The following code will immediately terminate the function and return the value of `ValueToReturn` to the calling procedure:

```
ValueToReturn = CLIP(CusNum) & '-' & CusAge
Return(ValueToReturn)
```

This function can be called as follows:

```
ReturnedValue=ProcC(LongToSend,ByteToSend)
```

Optional Return Values

Sometimes a programmer might want to call a function but without needing to know the return value. In that case, calling the function on its own would be appropriate:

```
ProcC(LongToSend,ByteToSend)
```

This, however would cause a compiler warning, "*Warning - Calling function as procedure"*

This is because the function will be returning a value, but there is no assignment statement on the receiving side to receive the value the function passes back. The compiler complains but lets it pass. To get rid of the warning message append the procedure prototype with the PROC attribute. This will tell the compiler that it's okay to call a function as if it were a procedure. The example application provides a simple use for this feature in the module ProcC under the menu heading *Return values"*.

Returning a value is great when there is only one item that’s of interest. But what happens when you want to return multiple values? This can be done, but not as might be expected - there is a twist!

Passing Values By Reference

In a normal procedure call, the called procedure receives a copy of whatever data you send it. Any changes the called procedure makes to that data do not affect the data that was used in the procedure call. This method of parameter passing is called called passing parameters by value.

You can also pass parameters by address, which means that any changes to the passed data affect the original data and not a copy.

In order to visualize this it might be useful to think of data in the computer's memory as blocks of data residing on a matrix, as illustrated in Figure 1.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1																	
2																	
3																	
4			J	O	E		S	O	A	P							
5																	
6																	
7																	
8						J	O	H	N		D	O	E				
9																	
10																	

Figure 1. A figurative layout of a computer's memory

Consider a STRING variable of length 10 called Cus:Name which contains the value "JOE SOAP". This data string resides at the coordinates, or address, C4. Clarion lets programmers reference the 10 bytes of data between C4 and L4 by means of the label Cus:Name. Look at this example:

```
Cus:Name = 'JOE BLOGGS'
```

This statement causes the program to get the address and length of `Cus:Name: C4, 10` and change the 10 bytes starting at `C4` to `JOE BLOGGS`**.** This shows that the key to Clarion's ability to make assignment statements work lies in its knowing the address of the variable that needs to be changed. Therefore another way to change `Cus:Name` would be to use its address directly. To do this, a straight assignment statement on the address of the variable will effectively change the value of `Cus:Name` by using its reference address, whether the scope of the data is in range or not. So how can programmers make use of this?

By default, Clarion allows only one value to be returned by a function, which is a problem when multiple values need to be returned. Using the above concept it is possible to pass a procedure the addresses of several local variables, and get the procedure to change the contents of the variables by means of these reference variables, or addresses. The procedure (optionally) still returns only one parameter, but by means of those passed addresses the called procedure is able to "reach back" into the calling procedure and change the respective variables.

To prototype a procedure so that the local variables `StringToSend`, `ByteToSend`, `LongToSend` get modified by the procedure `ProcE`, set up the following function prototype:

```
ProcE PROCEDURE (*STRING ReceivedStringAddress,↵
  *BYTE ReceivedByteAddress,*LONG ReceivedLongAddress), BYTE
```

The procedure declaration is:

```
ProcE PROCEDURE (*STRING ReceivedStringAddress,↵
*BYTE ReceivedByteAddress,*LONG ReceivedLongAddress)
```

The return value will be

```
DataWasModified BYTE
```

The asterisks prepending the datatypes in the prototype declaration tell the compiler to

send over the variables' *addresses* and not the actual values. In other words, looking back at the grid above, the coordinates `C4, 10` would be passed to the procedure instead of the `value "Joe Soap"`. To modify these variables, you can either assign the variables directly or use them as a use variable on a screen.

That's it! Run the example application, click on the "Pass By Reference" toolbox and enter some values in the click the button. Change the three values on the window that pops up and click OK - the popup window will close and all three values in the calling procedure will change - all without using any global variables!

Passing Objects

One of the joys of OOP is the fact that data *and* procedures are treated as an object - in other words as something that can be singularly referenced. By comparison, in procedural code, everything is either a verb or a noun - the verbs are procedures and the nouns are variables. OOP allows programmers to group procedures and variables and reference the whole bundle as a single variable. This means that you can also pass the whole bundle as a parameter to a procedure. An example would be to have a procedure that receives an ABC `FileManager` object, and the function uses the object-specific functionality encapsulated in the object by calling one or many of its methods and / or referencing its properties.

To prototype a procedure to be passed an ABC `FileManager` object set the prototype as follows:

```
ProcF PROCEDURE (FILEMANAGER TheFileManager)
```

The procedure declaration would be:

```
ProcF PROCEDURE (FILEMANAGER TheFileManager)
```

You might have noticed that it appears that the *entire object* is being passed to ProcF because there is no asterisk prepending the FileManager object. Once again, the compiler is smart enough to know that a class is automatically passed by address, so prepending the datatype with an asterisk is not necessary. To make use of the passed object, use the following syntax:

```
ReceivedObject.MethodName(MethodParameter1, MethodParameter1)
```

The following example does nothing spectacular - it serves merely as an example of being able to have a single procedure manage any number of data-file inserts by receiving the respective ABC object as a parameter. It shows, for example, that it is possible to create a central administration point for all calls to the ABC Insert methods, thus enabling other administrative tasks to take place simultaneously such as auditing and error handling.

To pass the `FileManager` object `Access:Customer` to `ProcF`, with the purpose of the `ProcF` procedure being to call the passed object's `Insert()` method and then to increment a global counter type the following in the `ProcF` Code section:

```
!Call the Insert method of the object
If ~TheFileManager.Insert()
  GLO:RecordsAdded += 1 !Do some centralized stuff
End
```

This procedure can be called as follows:

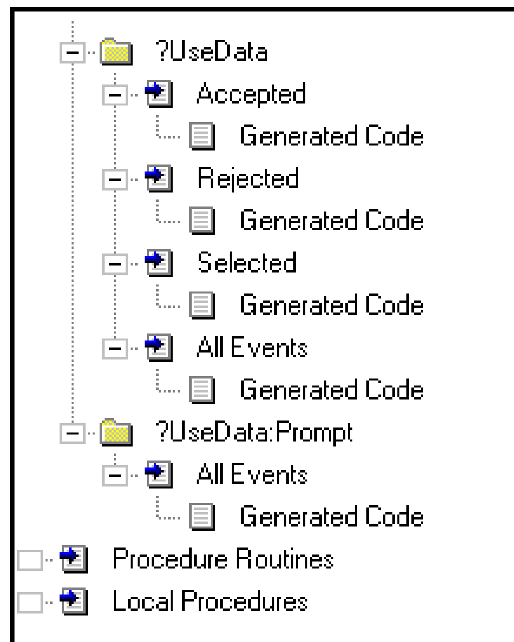
```
ProcF(Access:Customer)
ProcF(Access:Employee)
```

Its all very well being able to be able to use all these forms of procedures and parameters, but there is nothing worse than cluttering up the application tree with a bunch of mini-functions and procedures that may only be used by one or two procedures! One way to do this is to make use of *local procedures.*

Local Procedures

At the bottom of all ABC embed trees there is an embed point called Local Procedures which really tickled my interest when I first saw it.

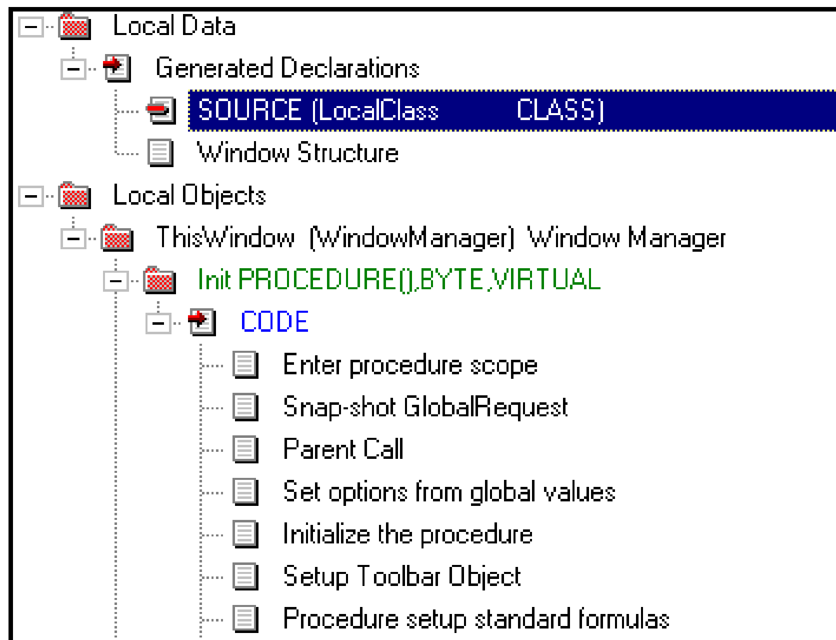
Figure 2. Tthe Local Procedure embed point in the ABC embed tree



To make use of this you can't just go in there and just slap in a procedure and expect to be able to call it, because a procedure needs a definition *and* a prototype declaration. In fact, this embed point is designed for the implementation of a CLASS's procedures. To create a local procedure that receives a LONG, multiplies it by 2 and returns the result to the calling procedure, go to the Local Data|Generated Declarations embed point shown in Figure 6 and type the following:

```
LocalClass CLASS
Func1         Procedure (LONG Received),LONG
END
```

Figure 3. The embed point where the class is declared



`LocalClass.Func1` is a function that will receive a variable, multiply it by two and then pass it back to the calling procedure.

Now go to the embed point Local Procedures as shown in Figure 5 and type the following:

```
LocalClass.Func1 Procedure (LONG Received)

  code
  return(Received * 2)
```

Declaring a class this way ensures that an object will be automatically instantiated on startup of the procedure and destroyed on termination, so memory leaks should not be a problem here. To make use of this function, place the following code in the embed of a button on the screen:

```
MESSAGE(LocalClass.Func1(10))
```

The result will be a message box displaying the value 20. Look at the example application to see this simple but powerful technique in action.

Passing Complex Structures

It should be becoming clear now that Clarion is very versatile when it comes to parameter passing. So, consider a `Queue`, which is a collection of any number of columns and rows that can be referenced, as a single entity, by the queue name.

It should be possible to pass a queue to a procedure just as it is possible to pass other complex objects. Of course, actually passing the physical data in the queue could be quite slow with a large queue, so instead of passing the physical data, the queue will be passed *by reference*. As with classes, queues are always passed by reference.

The only trick is that the queue could have any number of columns in it, making addressing it by reference unpredictable. For all other cases it is necessary to create a new datatype which can be used as a basis both for the procedure prototype and the actual queue itself. To create this new datatype declare the queue structure in the global data section and give it the `TYPE` attribute:

```
TheQueue QUEUE,Type
Field1      STRING(20)
Field2      LONG
End
```

The TYPE attribute means that this structure can be used as a data type on its own, just as STRING, BYTE or LONG are datatypes. Therefore the definition of the queue to be passed will be based on this datatype and not on the QUEUE datatype. Type the following in the data section of the procedure that the queue will be passed *from:*

```
MyQueue TheQueue
```

That declaration may look strange, but it works fine. The fields declared in the TYPE definition, field1 and field2, are implicit in this new data structure. Thus it is possible to address the inherent fields in MyQueue as MyQueue.Field1 and MyQueue.Field2.

To populate this queue, declare the LONG variable Loc:Counter and embed the following in a button of the procedure where MyQueue is declared (which is the procedure that will be passing the queue).

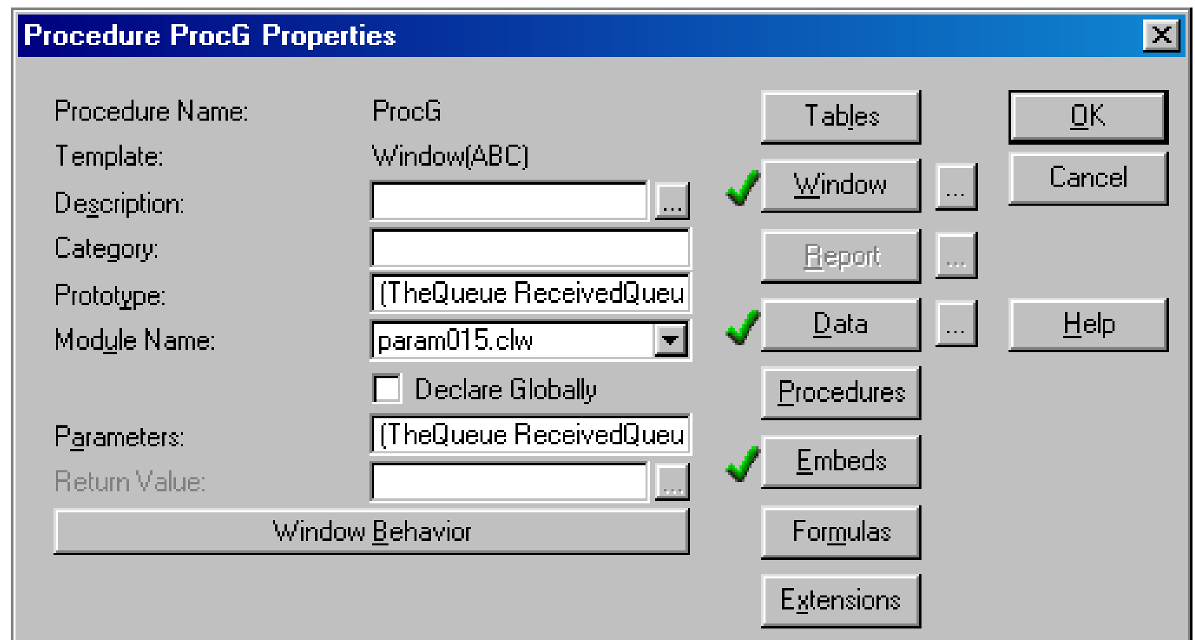
```
Loop Loc:Counter = 1 to 10
  MyQueue.Field1 = random(1,123456)
  MyQueue.Field2 = Loc:Counter
  Add(MyQueue)
End
```

To pass this queue to a procedure, pass MyQueue as if it were an ordinary variable:

```
ProcG(MyQueue)
```

Prototyping a procedure to be able to receive this *typed* variable is simple. It is prototyped the same way as one would prototype a procedure to receive a STRING, except that instead of declaring the datatype as a STRING the datatype would be the label of the *typed* datatype. Figure 4 shows what this would look like using an ABC template procedure:

Figure 4. Setting up a procedure to receive a typed queue



Once `ProcG` has received the queue or, more accurately, the reference to the queue, it may be referenced as an ordinary local queue. It can also be used to populate a listbox, and the procedure can even modify the data – bearing in mind that this is *not* another copy of the data, but the original data from the calling procedure. This means that any modifications to the "local" queue would also modify the data in the calling procedure too. Type the following code in the procedure that received the reference to the queue, and you will see that the data in the calling procedure will get changed too:

```
Loop Loc:Counter=1 to records(ReceivedQueue)
  Get(ReceivedQueue,Loc:Counter)
  ReceivedQueue.Field1 = 'Changed!'
  ReceivedQueue.Field2 = records(ReceivedQueue) - i#
  Put(ReceivedQueue)
End
```

This powerful technique can be used in many different areas, such as running reports directly from a queue or modifying hand coded browse lists directly from update forms.

Passing Reference Parameters To The Windows API

This final example shows a powerful use of reference variables used by WINAPI, which is the set of Windows OS function calls that provides underlying functionality to all Clarion Windows applications.

One of the WinAPI functions is the `FNSPLIT (FileNameSplit)` function, which parses a complete filename into its drive, subdirectory, filename and extension components. All of these values are set by means of passed reference variables. To make use of `FNSPLIT` do the following:

Go to the global embed point *Inside the Global Map* and type the following:

```
MODULE('WINAPI')
fn_split(*CSTRING,*CSTRING,*CSTRING,*CSTRING,*CSTRING),↵
   short,raw,name('_fnsplit')
END
```

In the local data embeds of a procedure declare the following variables:

```
ThePath      CSTRING(255)
TheDrive     CSTRING(255)
TheDir       CSTRING(255)
TheName      CSTRING(255)
TheExt       CSTRING(255)
```

Place all the variables on a screen and embed the following code in a button on the screen:

```
ThePath = 'c:\windows\command\xcopy.exe'
If Fn_Split(ThePath,TheDrive,TheDir,TheName,TheExt).
Display
```

The variables on the screen will immediately be filled with the respective values. To do the reverse, use the `FNMERGE` API call.

Function Libraries

Parameter passing is part of every decent language and helps makes application development rich and flexible. Over time, most projects result in the creation of a library of functions and procedures that get used across multiple applications, and the disciplined use of this library starts to leverage the output of each programmer in a similar way that templates do. The most important thing when managing a repository of such functions and procedures is *documentation*. Unless each programmer knows exactly what is available and how to use them, the functions in the repository will not be used and the whole objective of "normalising" the project's code base will fail.

You'll probably want to put your function libraries in one or more DLLs, which you can then make available to other applications. Take a look at Gordon Smith's article on breaking an application up into dlls to find out more.

A little extra time and effort building up a comprehensive library of business-specific functions for your project will go a long way toward making things smooth running and easier to maintain, and the key to creating a good library is knowing how to pass parameters At the end of the day, that means a finer product!

Download the example application

James Cooke has been using Clarion since 2.1 days and has been a die hard for "the cause" ever since. He and his family recently moved from South Africa to Texas and is currently working in the banking industry. He spends most of his free time basking in the sun by the pool with a good book or succumbing to that hard-to-kick addiction that persistently haunts the Western cosmopolitan neighborhoods - the yard sale.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Reborn Free

CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

TopSpeed

Clarion5 by TopSpeed

FREE Microsoft Internet Explorer

etc2000 EVENT SPONSOR

Five Rules For Managing Complexity

by Tom Ruby

Part 4

Using CHOOSE() To Concatenate Data
(Sep 15,2000)

The Nuts And Bolts Of Passing Parameters: Part 2
(Sep 15,2000)

Five Rules for Managing Complexity: Part 4
(Sep 15,2000)

Five Rules for Managing Complexity: Part 5
(Sep 15,2000)

September 2000 News
(Sep 15,2000)

"Rules! Rules! Rules! What ever do we need all these rules for? Creativity must not be restricted!"

Bugs! Delays! Rewrites! Upset Clients! What do those do for your creativity?

By knowing your science, you'll be released to pursue your art. I've given you the three rules of normalization plus six guidelines to help you manage the complexity of your application. This way you can keep your mind on the user's problem you're trying to solve rather than keeping busy with problems you've created.

Rule Number 1:
Eliminate repeating fields.

Rule Number 2:
Eliminate redundant data

Rule Number 3:
Eliminate Columns that don't belong

Guideline 1:
Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2:
It is a lot less painful to fix a painful mistake now than it will be later on.

Guideline 3:
A list is resizable, a form is not.

Guideline 4:
Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5:
Use keys to help the application identify records it is interested in.

Guideline 6:
The primary key represents the "thing." The rest of the record describes the thing the primary key represents.

Guideline 7:
You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

At this point, you're probably thinking either, "Give me more!" or "Oh no, not again." To tell you the truth, I'm procrastinating myself, and Dave is going to be wondering if I'm ever going to finish the five articles I promised him. Take charge! Grab the bull by the horns! Carpe Diem! So, without further ado:

Rule Number 4:
Isolate independent multiple relationships.

So what does this gobbledygook mean? It's not as bad as it sounds; in fact, it's almost a no-brainer. It means if you have more than one multiple relationship, and they don't have anything to do with each other, put them in separate tables. The best way to understand this, or perhaps the only way I can figure out to explain it, is with an example. Think about the garage example of part two with these tables:

MechanicTable	**MechanicSkillTable**	**SkillTable**
SysIDMechanic NameMechanic SocialSecurity DateBirth RateHourly	SysIDMechanic SysIDSkill Level	SysIDSkill NameSkill

If Herb wanted to keep track of tools owned by each mechanic, rule four tells him not to do it in the MechanicSkillTable, but to make a new ToolTable, which might look like:

ToolTable
SysIDMechanic ToolDescription

Now, this might seem like a no-brainer, but sometimes you can't convince the client that the two different "things" really have nothing to do with each other and they demand to see them in one list. It is pretty obvious that tools and skills don't look anything alike and so should be in separate tables, but suppose the issue is "Tools" and "Uniforms," both of which the mechanic owns? Perhaps Herb has been lead astray by the garage's vocabulary where when they refer to "Tools and Uniforms," they really mean "Property," including tools and uniforms.

Rule 4, in this case, doesn't tell me how to organize the data like Rules 1 through 3 do.

Instead, Rule 4 tells me that Herb didn't quite understand the requirements for the program. Herb knocks himself in the noggin and goes off to make the changes, creating a separate ToolTable. And I feel another guideline coming on:

> Guideline 8:
> **If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully.**

In case you haven't figured it out yet, the 5 rules are the Five Normal Forms. The whole idea of "Data Normalization" is to construct a model of the world out of tables. The better you construct this model, the better your program will be able to deal with the world and the better your program will be able to adapt to changes in its environment.

Next time, Rule Number 5 and the dreaded One-To-One relationship. For now, I'll recap the 5 laws and the 8 Guidelines:

> **Rule Number 1:**
> Eliminate repeating fields.
>
> **Rule Number 2:**
> Eliminate redundant data
>
> **Rule Number 3:**
> Eliminate Columns that don't belong
>
> **Rule Number 4:**
> Isolate independent multiple relationships.
>
> *Guideline 1:*
> Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.
>
> Guideline 2:
> It is a lot less painful to fix a painful mistake now than it will be later on.
>
> Guideline 3:
> A list is resizable, a form is not.
>
> Guideline 4:
> Link your tables by a hidden field that is completely meaningless outside the system.
>
> Guideline 5:
> Use keys to help the application identify records it is interested in.
>
> Guideline 6:
> The primary key represents the "thing." The rest of the record describes the thing the primary key represents.
>
> Guideline 7:
> You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.
>
> Guideline 8:

If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully.

Tom Ruby, who is no relation to the man who shot Lee Harvey Oswald, is an independent contractor living in the middle of a hayfield in Central Illinois with his wife Susan and two red-headed sons, Caleb and Ethan. He has been using Clarion for Windows since the summer of '95. Before that, he was a "TopSpeeder" using Modula II, so he has never used the DOS versions of Clarion.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE

Five Rules For Managing Complexity

by Tom Ruby

Part 3

Using CHOOSE() To Concatenate Data
(Sep 15,2000)

The Nuts And Bolts Of Passing Parameters: Part 2
(Sep 15,2000)

Five Rules for Managing Complexity: Part 4
(Sep 15,2000)

Five Rules for Managing Complexity: Part 5
(Sep 15,2000)

September 2000 News
(Sep 15,2000)

In parts one through four, I explained four of the five rules of Data Normalization. I also gave a number of related guidelines, and showed how to apply them in your Clarion Programs. To summarize, here are the four rules and the eight guidelines:

Rule Number 1:
Eliminate repeating fields.

Rule Number 2:
Eliminate redundant data

Rule Number 3:
Eliminate Columns that don't belong

Rule Number 4:
Isolate independent multiple relationships.

Guideline 1:
Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2:
It is a lot less painful to fix a painful mistake now than it will be later on.

Guideline 3:
A list is resizable, a form is not.

Guideline 4:
Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5:
Use keys to help the application identify records it is interested in.

TopSpeed
Clarion5 by TopSpeed
FREE Microsoft Internet Explorer
etc2000 EVENT SPONSOR

> *Guideline 6:*
> The primary key represents the "thing." The rest of the record describes the thing the primary key represents.

> *Guideline 7:*
> You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

> *Guideline 8:*
> If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully.

Let me start out by shocking you with Guideline 9:

> *Guideline 9:*
> Consider One-To-One relationships harmful.

How does that go again? "Consider One-To-One relationships harmful." A One-To-One relationship means that every row in one table is paired with exactly one row in another table. If the relationship is truly One-To-One, then why do you have two tables? They should be combined into one.

Having said that and having tried to convince you that there is no place for a One-To-One relationship, I'll tell you why you might want to use a One-To-One relationship. Clarion, like many other data management languages, edits a record in memory and saves it to the table when the user presses the Ok button. As a nod toward multi-user activity, Clarion checks to see that nobody else has updated that record before saving it and gives you the message, "This record was changed by another station. Those changes will now be displayed. Use the Ditto Button or Ctrl+' to recall your changes."

In a lot of multi-user systems this is a quite adequate solution to the "last one who saves, wins" problem because it is unlikely two users will be updating the same record at the same time. But suppose you have an online inventory system with dozens of cash registers ringing up tickets and "taking" things out of inventory, and a receiving department with several employees unloading trucks and putting items in inventory. The QuantityOnHand value is a very useful figure for store management, so operators would probably like to see it on the Item form. If you put it in the item record, the chance is pretty good that one of the cash registers or receiving clerks will change the number before the store manager, who is just wanting to correct the description, can hit the Ok button. And the chances are again pretty good that the record will change again before the manager can make his change again, even using the history feature, and hit the Ok button. It could lead to frustrated users.

A frequent solution is not to update the QuantityOnHand during the day and update it during an End-Of-Day operation. But this is an online system, and the user doesn't want to know how many the store had yesterday, but how many it has now.

Do you throw up your hands and announce that Clarion is totally useless for intensive multi-user applications? Some do. I don't. Remember Guideline 8: "If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully." With this in mind, I realize that I just haven't understood the problem fully. You see, there is a field, QuantityOnHand, which is *not* updated on a form, but by some other activity. The other activities happen to be cashiers selling items and receiving clerks unloading trucks. In a sudden burst of insight, probably accompanied by a burned

out light bulb, Guideline 10 pops into mind:

Guideline 10:
Separate automatically updated columns from manually updated columns.

Considering the new Guideline 10, there are likely several columns that are "automatically" updated, and I realize why Guideline 9 doesn't say "NEVER use One-To-One relationships." So I'll make a separate table and update it with a sequence like:

```
!Ainventory being the automagically updated part
LOGOUT(10,Ainventory)
Access:Ainventory.Fetch()
AINV:QuantityOnHand -= QuantitySold
Access:Ainventory.Update()
COMMIT
```

Or in an SQL environment:

```
Ainventory{PROP:SQL} = 'UPDATE Ainventory WHERE
  ItemSysID = xxx SET QuantityOnHand =
  QuantityOnHand - QuantitySold'
```

I could be extremely clever and put a timer on the form to retrieve the automagically updated figure and re display it now and then so the user can "watch" his inventory go up and down. Now the user knows exactly how many packages of "Screaming Yellow Zonkers" she has on hand, except for the one a customer is pushing around in a shopping cart.

The whole point of Guidelines 9 and 10 is to reserve One-To-One relationships to situations where there's a column that is being updated by something other than the form, and to prevent me from having to update two records on the same form.

Rule Number 5, or Fifth Normal Form, is also related to updating the database. In fact all the rules so far are related to updating the database.

Rule Number 5:
Isolate Semantically Related Multiple Relationships.

It's not my fault! Somebody at a university made this rule up. To illustrate Rule Number 5, imagine you're interested in tracking distributors and manufacturers of different parts. Since the manufacturer doesn't want to deal in the quantities you buy, you have to buy from distributors. Each distributor sells the parts from several manufacturers, and the same part may be available from different manufacturers. In a word, it's a mess.

You might think of four tables that look somewhat like this:

Distributor	**Part**
DistributorSysID DistributorName DistributorAddress And all that rot	PartSysID PartName PartDescription
Manufacturer	**DistributorPartManufacturer**

ManufacturerSysID ManufacturerName ManufacturerAddress Bla bla bla	DistributorSysID PartSysID ManufacturerSysID

The fourth table, DistributorPartManufacturer tells you which distributors sell which parts from which manufacturer. Rule Number 5 (or Fifth Normal Form) dictates against this since there are two different relationships, even though they're related. Instead, you should separate this table into two like this:

DistributorManufacturer	**artManufacturer**
DistributorSysID ManufacturerSysID	PartSysID ManufacturerSysID

So what does this gain you besides two more tables? Like the other rules, simplicity in updating. Suppose Acme Widgits starts making three of those handy bolts with the threads offset from the shafts for when the holes don't line up(See Figure 1). Since Acme is handled by four distributors, with the unnormalized table, you need to add 12 records to the big cross reference table to show that these parts now come from the four distributors where you get Acme parts. You also have a fairly complex piece of logic to figure out from the DistributorPartManufacturer table which distributors sell Acme so you can add these records to the table. If nobody else makes these parts, you have to add the three new part records.

With the normalized table, you need to add fewer records, and the code to add these records is simpler. You only need to add three records to the PartManufacturer table to show that Acme now makes the thread offset bolts. If you have lots of update activity, Rule Number 5 can improve your efficiency quite a bit, not to mention reducing complex logic.

To wrap this series up, I'll enumerate the 5 Rules and 10 guidelines:

Rule Number 1:
Eliminate repeating fields.

Rule Number 2:
Eliminate redundant data

Rule Number 3:
Eliminate Columns that don't belong

Rule Number 4:
Isolate independent multiple relationships.

Rule Number 5:
Isolate Semantically Related Multiple Relationships.

Guideline 1:
Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2:
It is a lot less painful to fix a painful mistake now than it will be later on.

Guideline 3:
A list is resizable, a form is not.

Guideline 4:
Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5:
Use keys to help the application identify records it is interested in.

Guideline 6:
The primary key represents the "thing." The rest of the record describes the thing the primary key represents.

Guideline 7:
You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

Guideline 8:
If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully.

Guideline 9:
Consider One-To-One relationships harmful.

Guideline 10:
Separate automatically updated columns from manually updated columns.

One more point. You may have heard it said that Third Normal Form is usually considered "normal enough." Most say it's because the situations in Fourth and Fifth Normal Forms rarely crop up. Actually, these situations appear all over the place, but usually by the time you've thought through the first three normal forms, you've already satisfied the Fourth and Fifth forms.

Finally!

Do you have to follow these five rules of data normalization? No, you don't. I have a hard time understanding why you wouldn't want to make your work easier. Indeed most software is developed the hard way anyhow so just go along with the crowd and give yourself ulcers.

Ok, I *am* being sarcastic. You don't *have* to follow the rules of data normalization, but I know that somehow, every time I've broken them, I've wound up wishing I hadn't.

Tom Ruby, who is no relation to the man who shot Lee Harvey Oswald, is an independent contractor living in the middle of a hayfield in Central Illinois with his wife Susan and two red-headed sons, Caleb and Ethan. He has been using Clarion for Windows since the summer of '95. Before that, he was a "TopSpeeder" using Modula II, so he has never used the DOS versions of Clarion.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

TopSpeed

Clarion 5 by TopSpeed

FREE Microsoft Internet Explorer

etc 2000 EVENT SPONSOR

Clarion News

Using CHOOSE() To Concatenate Data
(Sep 15,2000)

The Nuts And Bolts Of Passing Parameters: Part 2
(Sep 15,2000)

Five Rules for Managing Complexity: Part 4
(Sep 15,2000)

Five Rules for Managing Complexity: Part 5
(Sep 15,2000)

September 2000 News
(Sep 15,2000)

Read The August 2000 News

September 15, 2000

Insight 1.0 Beta 3 Released

The Insight Graphing product is now in Beta 3, with lots of new functionality. Insight will usually cost $299, but will be priced at $199 for the duration of the beta program. Beta users get free upgrades to the gold release, and beyond.

NetTalk 1.0 Beta 8d Released

NetTalk has been tweaked and prodded since the last the last update and is now more compatible with all those 'standard' email servers out there. The normal price for NetTalk is $299, but it's currently on a Special Price of $199 during the beta program. Beta users will automatically get a free upgrade to the gold release, and beyond.

Special Agent Version 1.25 Released

Special Agent 1.25 is out, with 1.26 about to be released. Now supports third party characters and the ability to open an agent in a specific position. Version 1.26 is currently in testing. This is a major release for the international market and includes support for screen translations, and support for alternative speech engines. Special Agent costs $199 and is available through www.ClarionShop.com.

September 12, 2000

solid.software Pricing Changes To USD

All prices on the solid.software web site have changed from EURO to USD. The company’s products will also soon be available at ClarionShop (www.clarionshop.com).

Bug Poster 1.00 Released

Bug Poster 1.00 is an extension for Clarion applications which let users report bugs and enhancement-requests via email directly to the developer. Includes a small DLL and templates.

C5.5 Enterprise And Professional Editions CR2 Available

The second candidate release of C5.5 is now available, in Enterprise and Professional versions. This patch requires Beta 1. New features include a Crystal Reports interface, enhanced RTF, HTML help, and new example programs.

SQL200n Working Documents Posted

Michael Gorman has posted a new set of SQL200n working documents. Lots of other SQL-related stuff on this page as well.

WinWord Previewer Version 1.1

WinWord Previewer 1.1 has been released. New per-report procedure options include preview only,

save file and close, save file and preview, overwrite existing file, append to file, call always or conditionally, and fixed or variable file names. Reports can now be automatically saved as Word documents without any user intervention.

Free Compile Manager Updated
The free CWCM Compile Manager has been updated to version 1.8. Features include better keyboard navigation on tree, fixed redraw problem on WinNT and Win2000, and the ability to make PRJ files.

September 5, 2000

MS FlexGrid Class Update
An updated MS FlexGrid class is now available from Taboga Software. This freeware class allows you to use Microsoft's FlexGrid OCX control for displaying and editing data.

PD 1-Touch Date Tools CR1 Update
The C55 CR1 update of PD 1-Touch Date, Time, and Scheduling Tools is now available. Other ProDomus products written for the C55 Beta versions do not appear to require updates. If you experience any problems, please let Phil Will know.

WisWeb September Announcement
September's newsletter describes the accomplishment of risk and cost reduction, and increase in quality and productivity. In addition, a number of SQL-related papers have been posted to the website.

CopyFlash 2.0 Released
CopyFlash 2.0 has been released New features include: unlimited number of Copy buttons per window; batch copy template for copying a range of records; fields can be excluded from copying; child fields in new record can be primed. CopyFlash works with all versions of Clarion from CW2x to C5.5. Demo available.

Copyright © 1999-2000 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.