WORKING WITH DATAEASE FOR WINDOWS

A La Carte

"The rich features of DfW can help us define a menu system way beyond what is possible with DOS", says Adrian Jones

This article stated life as I worked backwards from another I wrote in the last issue of Dialogue, where I suggested how you could add a data-entry form to a DfW report.

In outline, the idea was to create a generic table that contained DateFrom, DateTo, InvNo and a range of other 'selection' fields. You would then design forms over this table that gave a subset of those fields for use in specific instances, and use a control procedure to first display the appropriate selection form, then run the report based on that selection.

As I thought about it, I realised that this procedure could be made generic if I could find some means to pass over the names of the selection form and the report to be run. And so I came up with the idea of a report menu that displayed the names of the reports to be run, but had in the background the other document names. A button on the record posted the document names to a global variable using the setarray function.

After further thought I realised that I could probably run the entire database from such a set-up, and build a series of related records that I could use as a menu system.

And then I realised that I had, more or less, worked my way back to the original DOS menu system, where you enter records in a form. Indeed, this is also very similar to another DOS adaptations, where the menus are run inside a control procedure and the user selects an item from the list by adding 'yes' to a field value and then hitting F2 to select that item.

So I'm not claiming that this is a totally new idea. But eyes down, as we see how the rich features of DfW can let us refine our menu system way beyond what is possible with DOS.

WORTH THE EFFORT

This article will explore combining a number of tricks and facilities, including relating a table to itself, liberal use of CDFs, and using filters to limit what records are displayed.

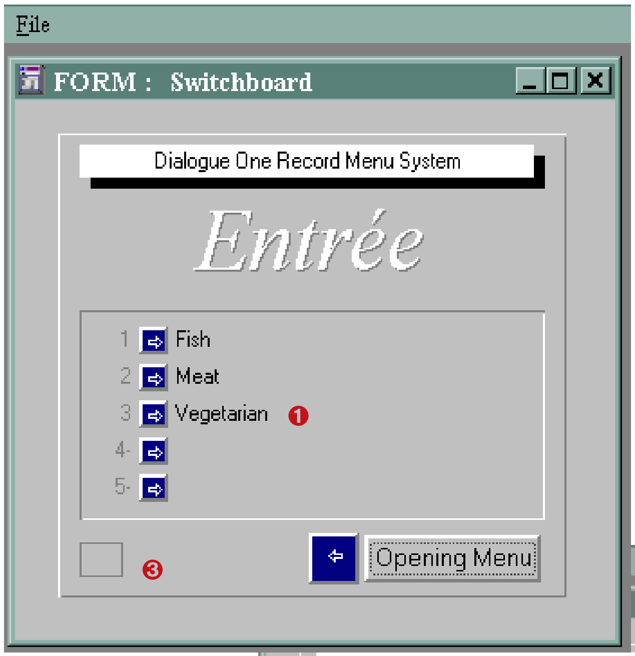
But first of all, why bother?

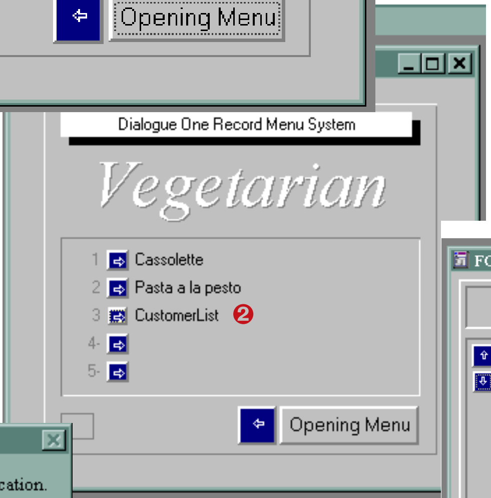
Perhaps individually, each benefit of the ideas in this article is not compelling enough to justify the effort, but combined together, I think they make it worthwhile. Additionally, readers may find other uses for the same techniques.

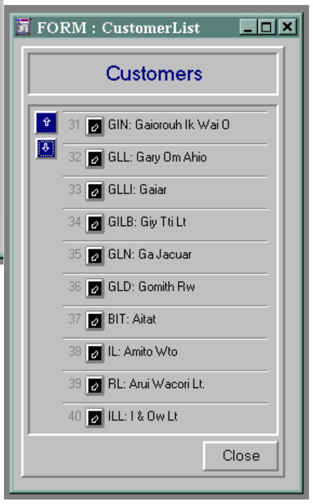
The advantages are:

1. There is only one instance of the menu form on screen at a time, so drilling down through a complicated menu structure won't result in your running out of resources (DfW only lets you have ten documents open at once).
2. Since there is only one document, you can achieve a consistent look and feel. You won't have to recreate the same basic document over and over again.
3. You can have as many items as you like on a menu.
4. You add a new menu option by adding a new record, not by changing the document itself. This eases maintaining and changing the system,

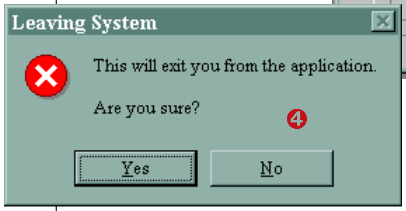
Ready To Order







31	GIN: Gaiorouh Ik Wai O
32	GLL: Gary Om Ahio
33	GLLI: Gaiar
34	GILB: Giy Tti Lt
35	GLN: Ga Jacuar
36	GLD: Gomith Rw
37	BIT: Aitat
38	IL: Amito Wto
39	RL: Arui Wacori Lt.
40	ILL: I & Ow Lt



This will exit you from the application.

Are you sure?

Using my hungry example, the user is assigned an opening menu, defined by a record in the table UserDocs. The menu consists of a set of related records that are 'called' by the opening record, each of which may have another set of related records.

Here, the user has clicked on the blue button next to the Vegetarian option❶, which resets the filter on the main form to be the Vegetarian record, and refreshes accordingly. Note that the first two screen shots are the same form on screen, and that there is only one form open at any time.

Also note that there are no 'real' fields on screen; everything is virtual.

For the third shot, the user has clicked on the option "CustomerList"❷, which refers to a real document in the system. The CustomerList record is flagged as such, and the conditional action of the blue menu selection button opens a document rather than finds another MenuItem record.

❸ This grey box is a virtual field that fires when the form is first opened and array 5 is blank. It gets the ID for the current user's opening menu, and then refreshes the form so that this is the record they see. Additionally, if the person logging in is not registered in the menu system, they will get a "see your administrator" message.

The big blue left arrow button takes the user back to the previous menu. If they try to go back 'before' their opening menu record, they will see this message ❹.

and using the same document in other databases.

5. You can determine what options each user is allowed, again by changing a value in a record.
6. You can define groups of users, and create menu systems based around them.

So what's it all about?

Stripping the concept down to its bones, there are two tables. The first I've called MenuItem. At its most basic, it contains two fields.

DocumentName is a 30 character indexed text field that acts as the menu name, just like the first field does in the DOS system menus form. For the moment, you could also make it unique. (In practice, another field should be used for uniqueness, but I want to keep the system as simple as possible for the purposes of this article.)

CalledByDocName is another 30 character indexed field that stores the name of the menu that calls this item.

We now need a relationship between this table and itself, based on DocumentName and CalledByDocName being equal. My convention is to always put the one side of the relationship in the left-hand column, and the many in the right. I name the left-hand side GetCallingDoc and the right-hand side SeeCalledItems.

WORKING WITH DATAEASE FOR WINDOWS

We then modify the derivation of CalledByDocName to:

```
Lookup GetCallingDoc
DocumentName
```

We also need a second relationship between the table and itself, but this time based on DocumentName being equal on both sides. Name it "N/A" on the left-hand side, and "SameMenuItem" on the right.

Now we can quickly create a form over this table. We will also display the same table, using the relationship name SeeCalledItems, as a subform. For the moment, just display DocumentName as the only field in both the main and subforms. We'll call this new form "Switchboard" (now, where have I heard that name before…).

Add a button to the subform record (which means that the button will appear once for each subform row), with the action 'form open related', and the arguments:

```
SameMenuItem , Switchboard
```

You'll have to add some records to see this in use. See the "Menu of the Day" sidebar for a taster.

Well, that's certainly got me hungry. But before we break for some food, quickly run the form, display the first record, then use the button to jump to the next menu item.

You'll notice that the title bar displays something like: "[FORM: Switchboard via Switchboard via Switchboard]", depending on how deep you go. Basically, you have managed to open three documents, in this case three instances of the same form. Drill down too far and you'll get the error:

```
Form window limit exceeded.
Please close a window.
```

Now, I'd argue that a menu system that goes down to eleven levels is very poor design anyway, but we might go down a few levels, open another document, run a report from that document that opens other documents, etc, and risk running out of resources that way. So the first exercise is to find a way to avoid this constant opening of new documents.

To do this, we're not going to open another document from a menu item. Instead, we will add a filter to the form so that it displays a given record, and change the filter to the identifier for a related record.

First, change the action of the button to 'execute CDF', with the script:

```
Setarray ( 5 , DocumentName )
+ clearselectionfilter ( )
```

(You will need to have installed the functions from the libaries CDFS2 and DFWACTS.)

This posts to a global variable, identified by the number 5, the ID for the subform menu item. But what does the second function do?

Clearselectionfilter is not an obvious choice, I admit. But this is the equivalent of choosing 'all records' from the view menu. This action will remove any user-entered filter, if any, and display the first record for the form.

But it will not 'remove' a designer-entered filter, which is the second half of this trick. In designer view, call up the Query By Model dialog from the document menu. With the first column on the dialog highlighted, click on the button labelled 'Select This Table's Records If:' and enter:

```
MenuItemID = getarray ( 5 )
```

Now run the document – and you'll find that even when you press F3, no records are displayed! Why? Well, we first need to seed that getarray value with something; at the moment it will only display records whose ID is blank – and of course there are none!

For the moment, just add a button to the main form labelled "See Opening Menu", whose action is 'execute CDF', with the arguments:

Menu Of The Day

Test data you might like to use to see the DfW menu system in operation:

DocumentName	CalledByDocName
Opening Menu	[blank]
Hors d'oeuvre	Opening Menu
Entrée	Opening Menu
Desert	Opening Menu
Soup	Hors d'oeuvre
Salad	Hors d'oeuvre
Fish	Entrée
Meat	Entrée
Vegetarian	Entrée
Dover Sole	Fish
John Dory	Fish
Lamb	Meat
Beef	Meat
Pork	Meat
Cassolette	Vegetarian
Pasta a la pesto	Vegetarian
Sorbet	Desert
Summer Pudding	Desert
Sorbet	Hors d'oeuvre

```
setarray ( 5 , “Opening
Menu” ) +
clearselectionfilter ()
```

Switch back to designer view, click on this button, then use the subform button to navigate through the menus. Watch the top line of DfW to confirm that you only have one document open at a time.

BACK TRACK

So far, we can navigate forwards through the menus, but we can’t go backwards. We can only jump straight back to the opening menu.

So we need another button. This is added to the main record part of the form, as we need to return to the item that called this item. Its action is the same as above, except that this time we post the calling item name to the array:

```
Setarray ( 5 ,
CalledByDocName ) +
clearselectionfilter ( )
```

Right now, all we are doing is moving backwards and forwards between menu items; we’re not doing any actual work. Let’s push the boat out a bit futher, and get this system crusing down the river.

We first need to distinguish between records in this MenuItem table that simply call other records, and ones that will call an actual DfW document. Add a yes/no field called CallsDocumentYN to the table, defaulting to’no’.

Now add a record to MenuItem. Type in the name of an existing calling item in CalledByDocName, but make the DocumentName field contain the name of an actual DfW document, and make sure CallsDocumentYN reads ‘yes’.

Change the action of the button on the subform row to:

```
If ( CallsDocumentYN = yes ,
documentopen ( DocumentName )
, setarray ( 5 , DocumentName
) + clearselectionfilter () )
```

The CDF will open in user view (i.e. will run) any document type specified as the document name, so it will display a form, and run a procedure or report. You can regard it as a generic way to run any document in your system.

There is a refinement to add before we go much further. We should stop the button from doing anything should there not actually be an item on this line, so add as a further modification to that button action above:

```
If ( DocumentName = blank ,
blank , If ( CallsDocumentYN
= yes , documentopen (
DocumentName ) , Setarray (
5 , DocumentName ) +
clearselectionfilter () )
```

OVER THE EDGE

We also have to stop the menu system from navigating backwards to a record before the Opening Menu, which at the moment will take us to a blank record. But to do that, I need to also turn my attention to how we'll assign menus to users.

For this, we need another table, which I'll call UserMenuRights. This contains the following fields:

- UserName, text 15, indexed, used to store the name of the user;
- DocumentName, to store the name of the menu item they are allowed to see, which is text 30;
- UserMenuRightID, text 46, unique and indexed, derived as:

```
Jointext ( UserName ,
jointext ( "|" ,
DocumentName ) )
```

We'll need some records in this table. For the moment, we'll assume that this menu system is for your own use. Write and run the following DQL:

```
For MenuItem ;
  Enter a record in
UserMenuRights
    UserName := current user
name ;
    DocumentName := MenuItem
DocumentName .
```

We also need to add another field to MenuItem, which we'll call vUserMenuRightID. This field must be virtual (hence why I start it with the letter 'v'), text 46, and derived:

```
Jointext ( Current "user
name" , jointext ( "|" ,
DocumentName ))
```

Add a relationship between MenuItem and UserMenuRights, based on these last two new fields being equal (yes, a virtual in a relationship!), and named "N/A" on the left-hand side and "SeeUserMenuRights" on the right.

Finally, go to Switchboard in designer view, open up the Query By Model dialog, and add this filter to the SeeCalledItems section:

```
Count of
SeeUserMenuRights > 0
```

To test this in action, first jump straight into designer view and leaf through the menus. Then go to UserMenuRight and change the name of a document, perhaps by adding an 'x' to the beginning (the idea is to create a name that does not exist in MenuItem). Now go back to Switchboard, jump to the opening menu, and check that the item you have changed is now no longer available.

Of course, different users may have different opening menus. We also have to work out how to get the document to display an appropriate opening menu when the user first logs in.

For this we add yet more fields. To MenuItem we add another virtual, called vOpeningDocLink, which is 18 characters long, and derived:

```
Jointext ( "yes" , Current
"user name" )
```

We then add to UserMenuRights a corresponding match field. To make data-entry a bit easier, add a yes/no field called OpeningDocYN, defaulting to 'no', and add OpeningDocLink (text 18, indexed) derived as:

```
Jointext ( OpeningDocYN ,
UserName )
```

Set one of the records as an opening 'document' by checking OpeningDocYN.

Now add a relationship between MenuItem and UserMenuRight, based on the OpeningDocLink and vOpeningDocLink fields being equal, and labelled 'N/A' on the left-hand side and SeeOpeningDoc on the right.

This time we are not adding a filter. Instead, we need a virtual field on the Switchboard form (not on the MenuItem table – well, at least, it isn't needed there) that will fire when the document is first displayed, and then basically shut up. I'll call it vFireCDF (it doesn't really matter what type it is), and it is derived:

```
if ( any SeeOpeningDoc
DocumentName not = blank ,
if ( getarray ( 5 ) = blank
, setarray ( 5 , lookup
"SeeOpeningDoc"
"DocumentName" ) +
clearselectionfilter() ,
blank ) ,
message ( "You are not set
up for this system.

See the system
administrator.", "Sorry..."
, 1 , 0 , 0 ) +
exitdataease() )
```

What this means is:

First check that there is an appropriate opening item for this user. If there is not, display the message and, when they've clicked on the okay button, kick them out of the application with the exitdataease function.

If there is an appropriate opening item and array 5 is blank, grab the name of their opening document, post this to array 5, and clear the selection filter. This will refresh the record on screen, which will now display with the user's opening item.

Thus the screen initially displays blank, but very soon (so you wouldn't notice) jumps to the opening item for this user.

We are almost there! There is one small detail to complete. It is possible that one user might start their menu system further down the tree than another user. In other words, their opening document is actually called by another document, though they in theory should never get to see this. But at the moment, they can navigate backwards 'past' their entry point.

To stop them doing this, change the 'navigate backwards' button to:

```
If ( DocumentName = any
SeeOpeningDoc DocumentName
, If ( message ("This will
exit you from the
application.

Are you sure?" , "Leaving
System" , 3 , 4 , 0 ) = 7 ,
exitdataease () , blank ) ,
setarray ( 5 ,
CalledByDocName ) +
clearselectionfilter ( ) )
```

Which means that should the document on screen be their opening item when they click on the 'back' arrow, a confirmation message asks if they really want to leave the system. If they answer 'yes' (which returns the value 7), they will be exited. Otherwise nothing will happen. If this is not their opening document, they will simply move back a level as before.

We should also change the action of our 'See Opening Menu' button to pick up their opening menu, instead of the currently hard-coded one:

```
setarray ( 5 , lookup
"SeeOpeningDoc"
"DocumentName" ) +
clearselectionfilter ()
```

The finishing touches are to the Switchboard document. Replace all real fields on this document with virtual ones, such as vDocumentName, derived as:

```
Jointext ( "" ,
DocumentName )
```

(Somehow DfW does not work reliably if you simply default to the name of the field you want to copy, hence the jointext.)

You should also make the form prevent data-entry, and edit the menus so that they contain the minimum of functionality.

FURTHER REFINEMENTS

I think that's enough for this issue, since it's already complicated enough! But you might like to contemplate the following for your homework:

- How to call a given document or menu from more than one other menu.
- How to define user groups (and the tables, forms and procedures that you might need to manage such a concept).
- Can you close the menu system altogether when you open a 'real' document – and re-open the menus when you close this document. (Hint: if you close the menu form at any time and then open it again, it will always display the last menu selection you had displayed. Try it!).