vb123.com

Smart Access

Solutions for Microsoft® Access® Developers

Navigating Through Recursion, Part 1

Christopher R. Weber

95 97 2000 2002 DOWNLOAD

Knowing your way around your Access project is important for any developer. In this first of two installments, Christopher Weber takes us through a navigation map generating algorithm he uses to populate a table that describes how the forms and reports in an Access database relate to each other. In next month's issue, Chris will demonstrate recursive query and reporting techniques he uses to generate a tree navigation map of the database.

HAVE you ever taken over a large Access project and been overwhelmed in the first requirements meeting by the plethora of synonyms used to talk about a Client/Customer/Whatever entity? Then there's the corresponding jargon for the Client screen, the Customer form, or the Whatever interface. And everyone in the meeting expects you to know how to navigate through the product you've barely seen.

I've walked into this situation at least three times in the past year: once taking over a project started by another developer, once consulting on a project managed by another developer, and once being added to a team of developers on a very large Access project. Invariably, the project is thrashing to catch up to an arbitrary schedule, there's little or no documentation, and the work needs to be finished yesterday. It's always the same problem ("the Client screen needs to have such and such") and I'm always asking the customer, "Okay, show me how you got there." At some point, I have to sit down and map out all of the navigation routes possible in the project, either on paper or in Visio.

Wouldn't it be nice to have a form like the one in **Figure 1** (on page 4) that shows you all the links between interfaces (forms and reports) in the database? Wouldn't it be nice to have the data generated for you automatically?

Mapping your interfaces isn't a trivial matter and really should be done in the architectural stage of requirements gathering in order to limit the product's

Continues on page 4

January 2003

Volume 11, Number 1

95 Applies to Access 95 | 97 Applies to Access 97 | 2000 Applies to Access 2000 | 2002 Applies to Access 2002

DOWNLOAD

Accompanying files available online at www.smartaccessnewsletter.com

In code, an underscore (_) as the last character of a line indicates that the line has been wrapped for layout purposes. In Access 95 and up you can use the code as it appears, but in Access 2.0 you must recombine the wrapped lines.

Continued from page 1

complexity. A couple of years back, I watched a company that was developing sophisticated banking software go under because no one could predict all the paths the user could take through their product. The owner / domain expert wanted maximum flexibility for the user, and designed the product to jump from almost any interface to any other. The need was clear in his mind, but none of the users could fathom it. The development team, who weren't bankers, was constantly redesigning the application because of recursive calls from one interface to the next. A called B, which called C, which reopened A again, and so on. Though it was the reality in the banking world that these entities were interrelated in a hyper manner, building software to mimic that reality became a cumbersome chore that eventually collapsed on itself. Worst of all, much of this recursion wasn't "discovered" until beta testing. Clearly, this was a classic failure caused by ignoring software engineering principles.

Knowing where the product begins and ends along with everything possible in between is certainly one step toward successful completion of a project. However, there are times when that map isn't known. For example, when an evolutionary prototyping approach is being taken, the design may grow as the project continues. The banking project was using this approach, which certainly got the project going at the beginning, but eventually led it down an evolutionary dead end. At some point, the project should have abandoned that model and taken on a more structured approach. And, as I stated earlier, the map may not exist when you have to take on an existing project.

The Access advantage

Because an Access project typically stores all of its forms and reports in a single .mdb file, and because Access exposes all of the objects and their properties to the developer, the information for tracing each route through a database exists in the .mdb file and can be gleaned to create a map of the system. There are, however, some presumptions in this approach that I must be clear about:

- Navigation is limited to moving between forms and reports (as well it should be). Users aren't directly exposed to queries and tables except as subforms / subreports (in 2000 onward).
- You know the name of the opening interface(s), whether it's a built-in Access switchboard or a custom form.
- Navigation is assumed to be primarily a branching mechanism, organized hierarchically: The user enters at a specific point and then chooses to move to one of a list of forms. Hyper-navigation (where any form can lead to any other form) between interfaces is limited. Any map of a hyper-navigation scheme is arbitrary as to where navigation begins and ends. Though the navigation path could be mapped in multiple views using my technique, they aren't the focus of this discussion.
- Navigation links are created in code using the DoCmd object's OpenForm and OpenReport methods, through custom functions that receive the name of the object to open, and through hyperlinks. No macros are used and calls to forms and reports in public modules (a rare case) aren't accommodated.
- The system may use DoCmd or custom functions to open objects whose names are exposed in a list or some control property. For example, the user can select one or more reports to print from a list box of possibilities, or click on a check box whose Tag property contains the name of the report to preview. You, as developer, must be capable of filling in the blanks where my algorithm cannot trace such designs.
- You don't want to repeat branches of the map that have already been exposed at some lower level, resulting in repetitive or recursive mapping of a branch. However, there are times when this is unavoidable.
- I'm presuming the use of DAO for object coding, and haven't used any Data Access Pages. For purely Access applications, this is typical and most efficient.

This list may seem somewhat limiting, but I've found that it covers the vast majority of navigation routes in the databases I've tested. Indeed, this may be a byproduct of my development style, but I presume that my techniques are typical of most Access developers.

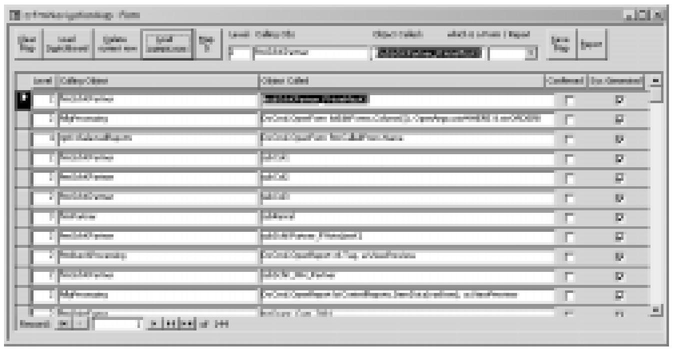


Figure 1. zsfrmNavigationMap showing unconfirmed entries found through dynamically loaded subforms and parameterized OpenForm methods.

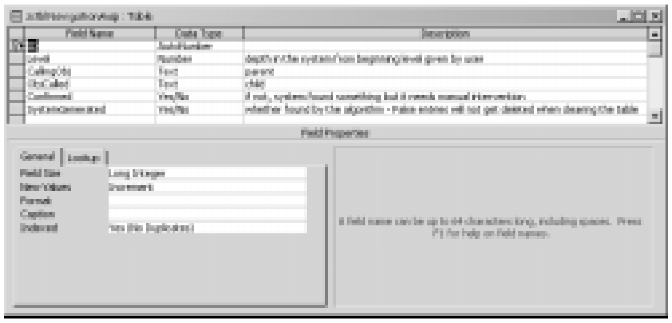
Where do you begin?

The first thing that you need is a way to store navigation link information. The answer is, of course, an Access table with a simple design, as shown in **Figure 2**.

The fields in zstblNavigationMap describe a self-joining hierarchical relationship between two objects: the CallingObj and ObjCalled. For instance, a switchboard (listed in CallingObj) could call a form named frmOrders (which would be listed in ObjCalled). The tables also describe the branching level at which the calling object is found. In my example, the switchboard would be at level 1 and links from frmOrders to other forms or reports would be at level 2, and so on. The Confirmed Boolean field in the table states whether the link found was verified (in this example, whether the ObjCalled was found to be an existing form or report in the database). SystemGenerated is set to true if the algorithm found the linkage. The field is left false for entries the user might have to make manually.

The table has several indexes (see **Figure 3**). The primary key, an autonumber, will increment with each entered record and is useful for examining the order in which the algorithm found the links. A unique index on Level-CallingObj-ObjCalled ensures that the same entry isn't made more than once, and that combinations of CallingObj and ObjCalled found at earlier levels can be used to suppress mapping of the same links found at higher levels. CallingObj and ObjCalled are each indexed because I'll be joining these fields to each other in a self-join query later on.

Lastly, to avoid aberrant entries, the table has a validation rule set in its property sheet that the CallingObj can't be the same as the ObjCalled. This shouldn't happen in the mapping algorithm, but I've seen it pop up. What's more, users shouldn't make such entries (as shown in **Figure 4**).



zstblNavigationMap : Table

Field Name	Data Type	Description
	AutoNumber	
Level	Number	depth in the system/'not beginning level given by user
CallingObj	Text	parent
ObjCalled	Text	child
Confirmed	Yes/No	if not, system found something but it needs manual intervention
SystemGenerated	Yes/No	whether found by the algorithm - False entries will not get deleted when clearing the table

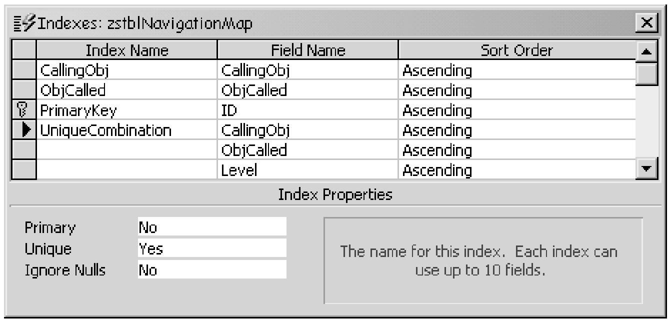
Field Properties

General | Lookup

Field Size	Long Integer
New Values	Increment
Format	
Caption	
Indexed	Yes (No Duplicates)

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Figure 2. zstblNavigationMap in design view.



Indexes: zstblNavigationMap

Index Name	Field Name	Sort Order
CallingObj	CallingObj	Ascending
ObjCalled	ObjCalled	Ascending
PrimaryKey	ID	Ascending
UniqueCombination	CallingObj	Ascending
	ObjCalled	Ascending
	Level	Ascending

Index Properties

Primary	No
Unique	Yes
Ignore Nulls	No

The name for this index. Each index can use up to 10 fields.

Figure 3. zstblNavigationMap Indexes window.

Filling the mapping table

To fill the table with mapping records, you should, theoretically, be able to recurse through the linkages between objects and enter records for each link found. Then, by following the records in Level-CallingObj-ObjCalled order, you can record how each object relates to the other objects. This turns out to be easier said than done.

To begin the process, I'll look at a procedure that traces a Switchboard table generated by the built-in Switchboard Manager menu option under Tools | Database Utilities. The routine cmdLoadSwitchboardTableEntries_Click loops through records in the Switchboard table that have a Command value of 2, 3, or 4 (Open form in Add mode, Open form in Edit mode, or Open report in preview). For each entry, cmdLoadSwitchboardTableEntries_Click calls my routine LogItAndDrillIn with the name of the ObjCalled (in this case, the Argument field in table Switchboard enclosed in double quotes), a pointer to a recordset for editing the table zstblNavigationMap, a Level value of 1, the type of object found (acForm or acReport), and the name of the CallingObj (in this case, "Switchboard"):

```
Private Sub cmdLoadSwitchboardTableEntries_Click()
On Error GoTo ErrorHandler
Dim rst As Recordset
Dim rstSwitchboard As Recordset
Dim intObjectTypeFound As Integer

If genObjectExists("Switchboard Items", acTable) Then
  Set rstSwitchboard = CurrentDb.OpenRecordset( _
    "SELECT Command, Argument FROM " & _
    "[Switchboard Items] WHERE Command in(2,3,4)")
    Set rst = CurrentDb.OpenRecordset( _
           "zstblNavigationMap", dbOpenDynaset)
  With rstSwitchboard
    Do While Not .EOF
      Select Case !Command
        Case 2, 3
          intObjectTypeFound = acForm
        Case 4
          intObjectTypeFound = acReport
      End Select
      Call LogItAndDrillIn("""" & !Argument & _
          """", rst, 1, intObjectTypeFound, "Switchboard")
      .MoveNext
    Loop
  End With
Else
  MsgBox "Switchboard Items table not found.", _
         vbCritical + vbOKOnly, "Process halted..."
End If
subNavigationMap.Form.Requery

Exit_Here:
```

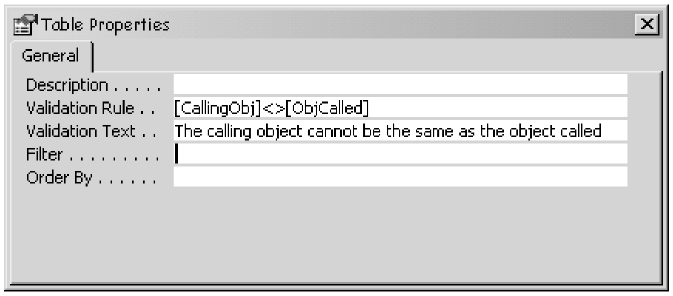


Table Properties

General

Description	
Validation Rule . .	[CallingObj]<>[ObjCalled]
Validation Text . .	The calling object cannot be the same as the object called
Filter	
Order By	

Figure 4. zstblNavigationMap Properties window.

```
  If Not rst Is Nothing Then Set rst = Nothing
  If Not rstSwitchboard Is Nothing Then
        Set rstSwitchboard = Nothing
  End If
Exit Sub

ErrorHandler:
  MsgBox "Error #" & Err.Number
  Resume Exit_Here

End Sub
```

LogItAndDrillIn() attempts to open the ObjCalled and examine it for further linkages. The reason the name of the object being called is passed in using double quotes is because the name will sometimes be embedded in a DoCmd.Open*object* line of code and be found in the first set of double quotes found in the line. I extracted the name in the first few lines of code. If a pair of double quotes can't be found, the reference is trimmed and marked for later manual discovery. An example might look like the following, where list box lstReports allows the user to open one of several available reports kept in a value list or a system table:

```
DoCmd.OpenReport lstReports, View:=acPreview
```

The routine to handle this code looks like this:

```
intFirstQuotePosition = InStr(strReference, Chr(34))
If intFirstQuotePosition Then
  intSecondQuotePosition = _
      InStr(intFirstQuotePosition + 1, strReference, _
      Chr(34))
  strNameOfObjectFound = Mid(strReference, _
    intFirstQuotePosition + 1, intSecondQuotePosition _
    - intFirstQuotePosition - 1)
Else
  strNameOfObjectFound = Trim(strReference)
End If
```

In the next step, LogItAndDrillIn() checks for existing entries in zstblNavigationMap with either the same CallingObj or ObjCalled at a lower level, because you don't want to log an object at a level higher than one already found. If you do, you may see entire sets of branches in the map repeat at various levels and, possibly, recursively call themselves—resulting in an endless hierarchy. Remember that banking software? A hyper structure would cause the same result. However, it may be too late to prevent this from happening by the time the new route is found.

How so? In my example, my Switchboard opened frmOrders. Now frmOrders could have a link to open frmCustomers. The frmOrders–frmCustomers link would be logged at level 2. If Switchboard also has a direct link to frmCustomer, a record with frmCustomers as the ObjCalled will be added for frmCustomers at level 1 (a link from Switchboard) in addition to the link to frmCustomers at level 2 (previously found called from frmOrders). My code prevents that second entry from being made.

This is a design tradeoff that I accepted because I always want to see the earliest call tree for an object, but not necessarily for any later repetitive branchings of the same object. This doesn't prevent seeing frmCustomers called at later levels, because the software checks for the current ObjCalled as the CallingObj at earlier levels. It merely stops the continuation of duplicate branchings of the navigation tree:

```
rst.FindFirst "(CallingObj='" & strNameOfObjectFound _
    & "') AND (Level<" & intFromLevel & ")"
fNotFoundPreviously = rst.NoMatch
If fNotFoundPreviously Then
  rst.FindFirst "(CallingObj='" & _
    pstrNameOfCallingObj & "') AND _
    (Level<" & intFromLevel & ")"
  fNotFoundPreviously = rst.NoMatch
End If
```

Before I log any entries, I check to be sure the ObjCalled exists. This is handled by the genObjectExists routine in the class module, a very handy piece of generic code that you may find useful. I then add a record to zstblNavigationMap through the rst pointer, indicating whether the object can be found in the database and that this is a system-generated entry:

```
If fNotFoundPreviously Then
  fObjExists = genObjectExists(strNameOfObjectFound, _
                   intObjectTypeFound)
  rst.AddNew
  rst!Level = intFromLevel
  rst!CallingObj = pstrNameOfCallingObj
  rst!ObjCalled = strNameOfObjectFound
  rst!Confirmed = fObjExists
  rst!SystemGenerated = True
  rst.Update
```

Now the recursion begins. First of all, remember that I passed in the recordset pointer to LogItAndDrillIn. I pass the reference in because I don't want to create multiple, unresolved pointers to zstblNavigationMap, which would be resource-intensive. It could also, possibly, leave me with records hanging in the midst of an edit should I somehow choose that route while programming. Secondly, I don't continue searching along the path exposed by ObjCalled unless it was passed in with quotes around it *and* it was found to exist in the earlier steps. In other words, I don't want to recursively search on a line like DoCmd.OpenReport lstReports, View:=acPreview where the actual name of the object(s) being called must be discovered manually.

Up to this point, I've been discussing entries in a table used by the switchboard program. The continuation of the search is handled by the routine NavigationMap, which receives the Level + 1, the name of the object found, the same name as the object to examine, and the type of object (acForm or acReport):

```
If intFirstQuotePosition And fObjExists Then
   NavigationMap intFromLevel + 1, _
       strNameOfObjectFound, strNameOfObjectFound, _
       intObjectTypeFound
End If
```

One last thing to discuss before you leave LogItAndDrillIn(): the error handler for error number

3022 (duplicate entry in a unique index). Should another branch call ObjCalled at the same level from the same CallingObj, the unique index would be violated. The unique index stops the entry, so you can just ignore the error and exit the routine:

```
Exit_Here:
  Exit Sub

ErrorHandler:
  Select Case Err
    Case 3022
      'duplicate entry in unique index
    Case Else
      MsgBox "Error #" & Err.Number
  End Select
  Resume Exit_Here

End Sub
```

Navigating through recursion

NavigationMap begins its processing by first verifying that both the CallingObj and ObjCalled exist in the database. This isn't redundant, as manual entries can be traced by NavigationMap(). In fact, if the database doesn't use table Switchboard, or has an alternate entry point, a call can be made through the interface to begin the tracing process:

```
If Not (genObjectExists(pstrNameOfCallingObj, _
          acForm) Or _
   genObjectExists(pstrNameOfCallingObj, acReport)) _
   Then
    MsgBox "Object " & pstrNameOfCallingObj & _
      " NOT found.", vbCritical + vbOKOnly, _
      "Process halted..."
    Exit Sub
End If

If Not genObjectExists(pstrNameOfObjectToExamine, _
                            pintObjectType) Then
  Select Case pintObjectType
    Case acForm
      MsgBox "Form " & pstrNameOfObjectToExamine & _
       " NOT found.", vbCritical + vbOKOnly, _
       "Process halted..."
    Case acReport
      MsgBox "Report " & pstrNameOfObjectToExamine & _
       " NOT found.", vbCritical + vbOKOnly, _
       "Process halted..."
    End Select
  Exit Sub
End If
```

If found, a call to GetObjectToExamine opens the object for examination and then minimizes it:

```
Set rst = CurrentDb.OpenRecordset( _
   "zstblNavigationMap", dbOpenDynaset)
GetObjectToExamine pintObjectType, _
   pstrNameOfObjectToExamine, objFrmRpt
```

If the object has a class module, each non-comment line of code is examined for OpenForm and OpenReport methods. The code also checks for a pair of custom functions, OpenAForm and OpenAReport, that I use to handle error 2051 that occurs when Cancel is set to true in a form's OnOpen event or a Report's NoData event. Both examples can be found in the database in this article's accompanying Download file (available at www.smartaccessnewsletter.com). Leaving the lines that handle this in the NavigationMap() routine won't hurt anything. However, if you have similar implementations that receive the name of the object to open, simply replace the references found here with your own. In either case, if found, the line of code is passed to LogItAndDrillIn where it's parsed for the object name in double quotes, along with the other required parameters:

```
If objFrmRpt.HasModule Then
  Set mdl = objFrmRpt.Module
  With mdl
    For i = 1 To .CountOfLines
      intObjectTypeFound = 0
      If Not Left(Trim(.Lines(i, 1)), 1) = "'" Then
        If InStr(.Lines(i, 1), "OpenForm") Or _
             InStr(.Lines(i, 1), "OpenAForm") Then
          intObjectTypeFound = acForm
        ElseIf InStr(.Lines(i, 1), "OpenReport") Or _
            InStr(.Lines(i, 1), "OpenAReport") Then
          intObjectTypeFound = acReport
        End If
        If intObjectTypeFound Then
          LogItAndDrillIn(.Lines(i, 1), rst, _
            pintFromLevel, intObjectTypeFound, _
            pstrNameOfCallingObj)
        End If
      End If
    Next i
  End With
End If
```

After examining the object's module, each control on the object itself is examined for calls in its event properties. But first, another call to GetObjectToExamine is made to reopen any objects that were inadvertently closed during the recursion process. This shouldn't be possible, but I've seen the DoCmd.Close action at the end of this procedure close objects that were opened several levels earlier in the recursion and were supposedly out of scope. This failsafe code ensures that you continue examining the current object of interest.

Any findings are passed to LogItAndDrillIn(), which continues the process recursively:

```
GetObjectToExamine pintObjectType, _
      pstrNameOfObjectToExamine, objFrmRpt
For Each ctl In objFrmRpt.Controls
  For Each prp In ctl.Properties
    intObjectTypeFound = 0
    If Left(prp.Name, 2) = "On" Then
      If InStr(prp.Value, "=OpenAForm") Then
         intObjectTypeFound = acForm
      ElseIf InStr(prp.Value, "=OpenAReport") Then
        intObjectTypeFound = acReport
      End If
    End If
    If intObjectTypeFound Then
      LogItAndDrillIn prp.Value, rst, pintFromLevel, _
           intObjectTypeFound, pstrNameOfCallingObj
    End If
```

Next, I check for hyperlinked objects whose reference can be found in a control's HyperlinkSubaddress property:

```
If prp.Name = "HyperlinkSubaddress" Then
  If Left(prp.Value, 4) = "Form" Then
    LogItAndDrillIn """" & Right(prp, Len(prp) - 5) _
      & """", rst, pintFromLevel, acForm, _
      pstrNameOfCallingObj
  ElseIf Left(prp, 6) = "Report" Then
```

```
    LogItAndDrillIn("""" & Right(prp, Len(prp) - 7) _
      & """", rst, pintFromLevel, acReport, _
      pstrNameOfCallingObj)
  End If
End If
```

Finally, I handle subforms and reports. However, as of Access 2000, you can't open a subform or subreport while its parent object has it open. Therefore, I must first determine whether the subform's SourceObject exists. If so, I remove the reference in the subform control so that the name of the object can be passed recursively to NavigationMap. I don't, however, call LogItAndDrillIn to record the link from the form to the subform because I don't consider a subform or subreport as a new node on the navigation tree. For instance, if you ask a user what interface they're using, they won't cite the subform, just the main form. So too, neither will I:

```
If ctl.ControlType = acSubform Then
  If Len(ctl.SourceObject) Then
    If intObjectTypeFound = acReport Then
      If genObjectExists(Right(ctl.SourceObject, _
          Len(ctl.SourceObject) - 7), acReport) Then
        strObjectToExamine = Right(ctl.SourceObject, _
                      Len(ctl.SourceObject) - 7)
        ctl.SourceObject = ""
        NavigationMap pintFromLevel, _
          pstrNameOfCallingObj, strObjectToExamine, _
          acReport
      End If
    ElseIf genObjectExists(ctl.SourceObject, acForm) _
            Then
        strObjectToExamine = ctl.SourceObject
        ctl.SourceObject = ""
        NavigationMap pintFromLevel, _
           pstrNameOfCallingObj, strObjectToExamine, _
           acForm
      Else
        'not form or report, could be query/table
      End If
    Else
     'empty subform--set at runtime
      LogItAndDrillIn ctl.Name, rst, pintFromLevel, _
            intObjectTypeFound, pstrNameOfCallingObj
    End If
End If
```

I don't trace into source objects that are tables or queries as is now allowed in Access 2000 and forward. This is because neither have events that will lead to new nodes on the navigation tree. Lastly, I have the cleanup code and error handling:

```
Exit_Here:
On Error Resume Next
  If Not rst Is Nothing Then Set rst = Nothing
  DoCmd.Close pintObjectType, objFrmRpt.Name, acSaveNo
  Exit Sub

ErrorHandler:
  Select Case Err
    Case 7784, 2467
      'subform / subreport source object already open
      'object inadvertently closed in previous call
      Call LogItAndDrillIn "UNABLE TO OPEN " & _
          pstrNameOfObjectToExamine, rst, _
          pintFromLevel, intObjectTypeFound, _
          pstrNameOfCallingObj)
      Err.Clear
      Resume Exit_Here
    Case Else
      MsgBox "Error #" & Err.Number & ": " & _
         Err.Description & " by " & Err.Source, _
         vbOKOnly, "Error in procedure NavigationMap"
      Resume Exit_Here
  End Select
End Sub
```

Once the examination is finished, the object is closed. All of this opening and closing of forms and reports is visible on the screen, which I find both interesting and reassuring (it lets me know that something is going on). I also specifically handle error 7784, which arises when the subform or subreport I wanted to examine is already opened through examination of an object higher up the navigation tree. This is an obscure, but real threat to the process, and I've run into it on a test of a banking application (imagine that!).

Making it easy(er)

The form's zsfrmNavigationMap (you saw it back in Figure 1) and zsfsubNavigationMap expose all the mapping and maintenance methods for my data.

The subform zsfsubNavigationMap displays the records in zstblNavigationMap. The records appear, by default, in primary key order, which is the order in which they're discovered by the algorithm. I found this order useful for debugging and have left it that way. A click on the raised labels allows you to sort by the corresponding data fields using my ExplorerSort function included in a public module in the sample database. You can also use arrow keys to move up and down between records like a datasheet. This is enabled through the use of Form_KeyDown_AsDataSheet routines, found in the subform's module. All fields are editable except SystemGenerated. This allows you to make manual entries, corrections, and edits when needed.

The main form, zsfrmNavigationMap, contains the coded methods for extracting the map and maintaining zstblNavigationMap. The Clear Map button allows you to clear the map table of all records that are system-generated. Should you decide to clear the table and run the algorithm again, non-system-generated entries are *not* deleted, allowing you to avoid repetitive manual rediscovery of those entries the system might not find.

The Load Switchboard button calls cmdLoadSwitchboardTableEntries_Click(), the first procedure you examined. Delete Current Row deletes the current row in the subform. The Load Current Row button copies the entries in the current row of the subform into the text boxes at the top of the main form. Map It passes the parameters in the text boxes to NavigationMap(). So, should you have a custom switchboard or other opening form, you'd begin your examination of the database by entering 1, *formName*, *formName*, and selecting Form as the type of object called. Click Map It and you're on your way.

Continues on page 19

Recursion, Part 1...

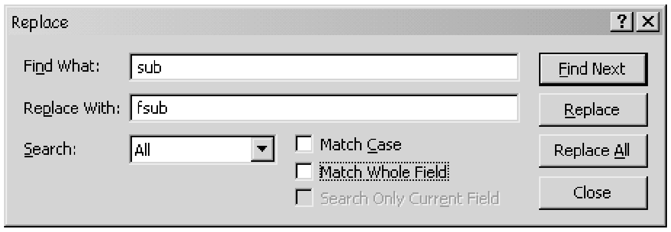
Continued from page 8

After this automated examination, you might have entries that describe subforms that are dynamically loaded in a tab control's OnChange event. Several are shown in Figure 1. Being dynamically loaded, the SourceObject property will be blank and thus, NavigationMap can only pass pstrNameOfObjectToExamine to LogItAndDrillIn. There's also a special circumstance when the subform / subreport source object can already be opened in a previous call. In this case, NavigationMap() passes "UNABLE TO OPEN" & pstrNameOfObjectToExamine to LogItAndDrillIn(). A third type of entry occurs for parameterized calls to OpenForm / OpenReport methods. These you'll have to trace by hand and make manual entries for tracing.

Because I always use the LR naming convention for my subform controls, it's usually easy to determine that a control named "subXXXX" is a container for "fsubXXXX". So, for the subform entries, I use the Find and Replace dialog to replace entries that begin with "UNABLE TO OPEN" with the letter "f", and entries that begin with "sub" with "fsub" (see **Figure 5**). I then press the button Load Current Row for each entry and then the Map It button. The dissociated subform opens and is examined just as if it were found.

The last two buttons at the top right of the main form allow you to save the map you've generated as "zstblNavigationMap" concatenated to Now, and run the report that displays the tree diagram resulting from recursive analysis of the navigation map data (**Figure 6**).

The query for this report and the formatting of the lines that connect the nodes is another story that I'll save for next month's issue.



Find What: sub

Replace With: fsub

Search: All

☐ Match Case
☐ Match Whole Field
☐ Search Only Current Field

Figure 5. Replacing dynamically loaded subform entries with the name of the subform.

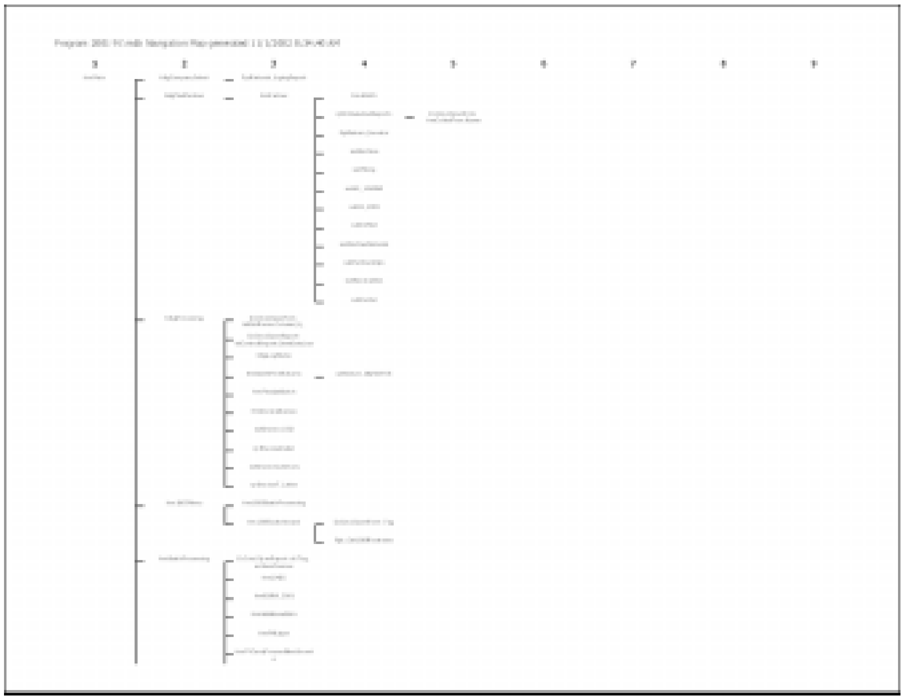


Figure 6. I'll talk about zsrptNavigatonMap next month.

Do try this at home

Remember, nothing's perfect. So far, my Navigation Mapper has done a pretty decent job on every database

I've tested, but I really have no way of testing whether this thing absolutely works every time. Most of the errors I've encountered have been unexpected holes in my logic. Personally, my clients and I are happy just to have the roadmap it creates as a starting point, though usually it's complete.

Until next month, explore the objects in the accompanying database by using the Load Switchboard and Map It buttons. The different problems discussed in this article—including hyperlinks, custom OpenAForm() and OpenAReport() functions, and even a dynamically loaded subform—are included in the sample database. The database has no data behind it, just a series of blank forms and reports, but you'll get the idea.

To use my Navigation Mapper in your own database, copy zstblNavigationMap, zsfrmNavigationMap, and zsfsubNavigationMap to a backup of your file. You'll have to turn on Hidden Objects in the Tools | Options dialog to import and use them. If you used the Switchboard Manager to create your opening form, just press the Load Switchboard button. If you have a custom form, enter 1, *formName*, *formName*, and select Form as the type of object called before clicking the Map It button. Then watch the fun begin. Any unconfirmed entries will need your attention, but hey, it beats going in cold on that new project. ▲

DOWNLOAD **NAVIGATE.ZIP at www.smartaccessnewsletter.com**

Christopher Weber travels throughout the country teaching Access development and programming seminars for The DSW Group in Atlanta, GA. He's been an Access developer since its first release, enjoys working with clients, and heads the DSW Group's Access development and training team. www.access-training.com, .