

I have frequently seen questions about how to get information back from a form called via the DoCmd.OpenForm command and opened in dialog mode. I have had the same problem myself, and after experimenting with numerous constructs, have hit upon a system that has been working for me quite nicely. The short answer is to create a public variable that the called form can write into, but there is a good bit more to it to create a comprehensive system that is clean, simple to use, and flexible. This is how I have done it.

Essentially, it is a systematic method of passing back a string, the same as the OpenArgs parameter. OpenArgs is strictly a string, with the added provision that it may be null. My method does not allow nulls, although it could easily be modified to allow that. The variable could be a variant, which would allow all sorts of things, including nulls, or even a user-defined class, for maximum flexibility. However, for my purposes, it is preferable to stick with strings, the same as the OpenArgs parameter – less possibility for confusion, and the ability to set a null or more complex constructs would not really provide any benefit to the way I am using it.

I use a specific format for these strings, in both the calls to forms, via OpenArgs, and back to the calling form, via my public variables. Those are string variables, called **mdl_PassBack**, always and everywhere, just like OpenArgs always has the same name. Using a string variable rather than a class variable also means it does not need to be initialized.

I pass parameters to the called form by constructing a string into the OpenArgs parameter. It always uses the syntax:

```
Param1=Value1<vbCrLf>
Param2=Value2<vbCrLf>
Param3=Value3<vbCrLf>
Param4<vbCrLf>
Param5=Value5<vbCrLf>
...
ParamN=ValueN
```

much like the classic .INI file format, although I did not add the ability to include comments and blocks, as the .INI does. I only pass a few parameters, so I did not see any point in such complications. I do allow parameters with no value, as in line 4, because sometimes I pass things like 'NewRecord', indicating that the called form should immediately start a new record, and there may not be any values associated with that instruction. Blank lines and spaces are ignored, and the return string from the called form to the calling form is constructed exactly the same way.

Each called form's Load event contains a small loop as the first item in the Load module, which one by one, picks off each line in OpenArgs and feeds the extracted parameters into a Select Case statement. This extraction is done by a small public sub, named Parse_OAPB, set up to handle exactly this format – single lines, containing either a parameter or a parameter=value pair, delimited by vbCrLf. The last line does not need a trailing vbCrLf, but it can have one – it is ignored. The code for Parse_OAPB is at the end of this post.

There is a Case clause for each parameter that the form is able to handle, and a Case Else: Stop command as the last item in the Select Case block, as a safety measure if I screw up and pass a parameter it doesn't expect. Once the OpenArgs string is completely dismantled and parameters handled, the Load event goes on – sometimes just ending, sometimes doing additional prep work based on those parameters. It would of course be possible to do the prep work right in the Select Case block, but I prefer to parse the entire OpenArgs string and set

environment variables as the parameters are identified, then after the parsing is complete, do the things that the parameters direct. It is slightly more code, but the difference in performance is undetectable, and such separation makes the structure much clearer. Each called form has such code in its Load event, which parses the OpenArgs string and sets up various conditions in the form before giving control to the user. An example of such a Load procedure is here:

```
Private Sub Form_Load()  
Dim x$, xl$, xr$, Pridat As Boolean, Upravit As Boolean, KodHledat&  
Set gbl_frmClovek = Me  
If Not IsNull(OpenArgs) Then  
' Loop to dismantle OpenArgs  
  x = OpenArgs  
  Do Until x = ""  
    Parse_OAPB xl, xr, x  
    Select Case xl  
      Case "Add": Pridat = True  
      Case "Edit": Upravit = True: KodHledat = Val(xr)  
      Case "CallingForm": lcl_CallingForm = xr: Forms(lcl_CallingForm).mdl_Passback = ""  
      Case Else: Stop  
    End Select  
  Loop  
' Set up form according to OpenArgs parameters  
  Select Case True  
    Case Pridat  
      With gbl_frmClovekEdit  
        .tglPridat = True  
        .tglPridat_AfterUpdate  
      End With  
    Case Upravit  
      gbl_frmClovekDS.Recordset.FindFirst "KodClovek = " & CStr(KodHledat)  
    Case Else: Stop  
  End Select  
End If  
End Sub
```

The second half of the routine does what the passed parameters dictate. Here it is a Select Case True structure. It could also be an If Then Elseif Then Elseif Then... End If series, or serial If Then or If Then End If statements, if the parameters are not mutually exclusive, or anything, really. What any particular called form does after parsing the parameters out of the OpenArgs string is not actually relevant to the structure I have developed, but I included this example just for show.

I eventually realized that everything I wanted to pass back could also be expressed this way, making life quite simple. To this end, I define public variables: mdl_PassBack\$. I define global variables with the prefix gbl_, public module variables with mdl_, and private module variables with lcl_. Variables inside procedures I simply name by whatever they're doing, but this prefix convention serves me well for keeping track of where a variable with wider scope is defined and what could possibly affect it.

This public mdl_PassBack string variable is defined in every form that wants some information back from a form it calls. One of the parameters I pass to the called form is the name of the calling form, for example:

```
DoCmd.OpenForm FormName:="MyPopupForm", OpenArgs:="CallingForm=" & Me.Name &
vbCrLf & "PublicationAutoID=" & CStr(txtPublicationAutoID)
```

would tell MyPopupForm that it should somehow address the record specified by PublicationAutoID, and that it had been called by the form whose name is paired with the CallingForm parameter.

```
DoCmd.OpenForm FormName:="MyPopupForm", OpenArgs:="CallingForm=" & Me.Name &
vbCrLf & "NewPublication"
```

would indicate that the called form should immediately start a blank new record, and again, that it had been called by the form whose name is paired with the CallingForm parameter.

Each called form has a module-wide private variable defined to hold the calling form name:

```
Private lcl_CallingForm$
```

and one of the Case statements in the initial parsing loop will be:

```
Case "CallingForm": lcl_CallingForm = xr: Forms(lcl_CallingForm).mdl_Passback = ""
```

This stores the name of the calling form in the module-wide private variable, and clears out the calling form's mdl_Passback variable, to remove any ballast from possible previous calls. It also has the debugging benefit of immediately crashing the called form if I forget to declare the mdl_Passback variable in the calling form. Before the called form closes, or actually, any time during its life, it will execute the command:

```
Forms(lcl_CallingForm).mdl_Passback = "...some information..."
```

This refers back to the calling form by the name it passed in OpenArgs. Obviously, this does not need to be in only one place. Many of my forms have this action in the form's Close event, but this is not necessary. A button that the user clicks might be to select a specific record, and the code in that button event would load the mdl_Passback variable with the ID of that record, using something like mdl_Passback = "SelectedRecordID=" & CStr(PublikaceAutoID) and immediately close the form. Simply closing the form would leave the mdl_Passback variable blank, and the calling form would know that the user had left the form without doing anything.

Changes to data I handle using custom events and WithEvents variables, to broadcast to anyone who cares that the contents of a table have changed, but this construct I built to inform about a user's actions or choices while a dialog form is open, not about changes to data. The calling form's public module variable always being named that same name makes remembering how to return information easy. Usually in the Close event (but could be elsewhere), a string is assembled that contains all the information that the called form wants to send back to the caller. The returned information is again a string assembled of one or more lines, using the Parameter=Value or just Parameter syntax, separated by vbCrLf, just like what arrives in OpenArgs. This lets me use the same construct that I use in the Load event procedures to pick apart the returned string - a loop containing calls to the Parse_OAPB routine. There is no requirement that the information string be constructed that way, or that I use that Parse_OAPB routine for both tasks, but sticking to this standard has made my life much simpler than before, when I had an unbelievable mess - each OpenArgs assembly had whatever had occurred to me at the time I wrote it, and every change made for a lot of work in matching the construction in the caller to the handling in the called, parameters sometimes separated by commas, sometimes

semicolons, sometimes colons, order of parameters crucial, and no easy way to vary parameters by who the caller is and what the caller wanted, nor any easy way to pass back information to the caller. This new method allows any number of parameters, in any order, all in the same layout, and in both directions. Often the information passed back is trivial, and could easily be handled with less code, but again, this standardization makes things very clear, easy to locate when editing, and allows for very simple expansion of capabilities should the requirements grow more complex. To handle a new parameter, it is one new Case clause, into a Select Case block of code already in place. I have to write the code to deal with the new parameter, but I would have to do that anyway. It is obviously a bit slower, but once again, undetectable by a human. Compared to the delay of a user clicking a control and waiting for a form to open or close, the additional computation time required by this extra code is less than trivial.

This construct might not be adequate to service more complex communication requirements, but it has worked fine for everything I've needed so far, and I somewhat suspect that more complex requirements would likely arise from an improper isolation of tasks. Proper modularity and encapsulation require minimal passing of information between modules. Needing to pass a lot probably indicates that a restructuring of task allotments is a good idea, or at least worth considering.

Here is the Parse_OAPB routine. All three parameters must be passed ByRef, since they are all modified here and returned to the caller. It picks off the top line, separates it into left and right, if there is an equal sign, or just left, if there is not. The trailing string gets whatever is left after picking off the top line. When the last line has been parsed, an empty string (NOT a Null) is returned in the Trailing parameter. An empty line is discarded without action, and the next one is picked up. All spaces are simply discarded. Since it does not interact with the user, there is no error handling – it is up to me to properly test everything I pass it.

In the Load events of called forms, the OpenArgs information must first be moved to a string variable, since OpenArgs is a read-only parameter - Parse_OAPB would not be able to modify it. When dismantling the information brought back in mdl_PassBack, I can use that variable directly as the third parameter in the calls to Parse_OAPB, since it is a user-defined variable. It also has the handy side effect of clearing out that variable, so that it is ready for the next such call. The called form's Load routine clears it anyway, but this ensures that information does not linger where it is no longer needed, possibly triggering something unwanted.

```
Public Sub Parse_OAPB(ByRef OAL$, ByRef OAR$, ByRef OAT$)
' Parse (O)pen(A)rgs(P)ass(B)ack
' OpenArgs Left, OpenArgs Right, OpenArgs Trailing
Dim i&
OAT = Replace$(OAT, " ", "")
Do
    i = InStr(OAT, vbCrLf)
    If i = 0 Then
        OAL = OAT: OAT = ""
    Else
        OAL = Left$(OAT, i - 1): OAT = Mid$(OAT, i + 2)
    End If
Loop Until (OAL <> "") Or (OAT = "")
```

```
i = InStr(OAL, "=")
If i = 0 Then
    OAR = ""
Else
    OAR = Mid$(OAL, i + 1): OAL = Left$(OAL, i - 1)
End If
End Sub
```