Add an Audit Trail to your Access Database

Published: 22 April 2013
Author: Martin Green
Screenshots: Access 2010, Windows 7
For Access Versions: 2003, 2007, 2010

About This Tutorial

My Access tutorials are designed to help people learn about building databases and how to add functionality to their databases with VBA programming. If you are just looking for some code to copy then skip down to the relevant code listings and help yourself to whatever you need. But if you want to learn something so that maybe next time you can figure it out for yourself, I urge you to read the accompanying notes that explain how I have come to the decisions necessary to fulfil the task at hand. I believe that in addition to telling people what to do, it helps to explain why they are doing it so my code listings are accompanied by explanations of what the code does, how it works and why it is necessary. I have also included two ready-made sample databases using the code shown here which you can download using the link at the bottom of this page.

Why Use an Audit Trail?

If you need to keep track of changes to your data then you need an Audit Trail. A comprehensive Audit Trail records not only what changes are made but when and by whom. Perhaps you are interested only in data that is changed, or perhaps you also want to know about data that has been added or deleted. This is potentially quite a daunting task. You may have many tables, each with a multitude of fields. In a busy database this could result in a very large number of actions being logged so, whatever method you choose, your Audit Trail has to be accurate, reliable, and it must not interfere with the smooth running of the database.

In this tutorial I describe two variations of a simple Audit Trail tool. In one, only edits to the data are recorded, with the option to ignore the addition of new records. In the other the approach is more comprehensive with edits, additions and deletions recorded. Both record the date and time of the action and the identity of the user responsible.

Valuable as it is, it's important to remember that an Audit Trail alone isn't the complete answer to your data security issues. You should make regular backups of your database files and make full use of available security provisions.

The Requirements

Here's what I want from an Audit Trail:

- I want my Audit Trail to keep a record of changes to my data. When a user edits a record I want to know which fields were changed, what the values were before the change was made and what they were changed to. I also want to know who made the changes and when they did it.
- I might want the option to ignore the addition of new records since that might involve duplication of data. After all, if I want to know what data was added I just have to look at the database. And I can record the user ID along with a timestamp when records are added.
- I would like the facility to specify which fields are included in the audit in case I decide that it is not necessary to record changes to every field.
- I might also want the option to know about the deletion of records. This may involve storing the details of the deleted record but, if I have incremental backups of my data, I will already have this information to hand.
- For ease of implementation the Audit Trail should be easily applied to various different datasets without having to re-write the code to suit each one.

The Provisos

I'm going to use VBA to build my Audit Trail tool and since code cannot be attached to a table I can't keep track of changes made directly on the tables themselves. This means that I need to prevent users from working in tables. In my opinion this is good practice anyway and I (almost) never allow users to see the tables in the databases that I build. If you think that users might try to bypass the Audit Trail then prevent them from working in the tables by either hiding them or not showing the Navigation Pane. More persistent users can be thwarted by using code to disable shortcut keys so that they can not open the Navigation Pane. Similarly, data added, changed or deleted by other methods such as queries might need to be tracked so, if necessary, all manipulation of data should be achieved through your user interface (via forms in other words) so that you can retain control of your data and make sure that everything you want recorded is.

I am not going to cover all the ins and outs of database security in this tutorial but, if you do nothing else, hide the Navigation Pane and protect your code with a password. Build a Switchboard or Home Page to give your users easy access to all the things you want to allow them to see.

The Plan

Capturing the Changes

Now that I have decided that my users will work in forms I have a number of easy ways to track what they do. When figuring out my approach to this task my first thought was to make use of the *Dirty* property. The form itself and controls that display data have a *Dirty* property that you can inspect with code to find out whether the form or an individual control contains unsaved changes. There is also an *OnDirty* event that fires when a change is made to the data. But I decided that, although useful in another context, I did not need to make use of these.

I am only interested in changes that are saved, so the time for me to detect any changes is at the point of saving. The form has a *BeforeUpdate* event which fires immediately before a record is saved. At this point the data on the form is just about to be written to the underlying table so two versions of the record exist, the old version which resides in the table and the new version which resides on the form. In addition to their *Value* property form controls that display data have an *OldValue* property. The *Value* property represents what you can see on the form, which may be the original value if no change was made. The *OldValue* property represents the original value before any change was made. By comparing the *Value* with the *OldValue* I can determine whether change was made and, if so, what the values were before and after the change.

If changes have been made, a form automatically saves its data when the focus is moved to a different record or when the form is closed. In addition, the user may choose to save the data manually by means of one of the built-in commands or by clicking the record selector. Regardless of the method by which the data is saved I can be confident that, provided the changes were made using a form, I can trap the event using *BeforeUpdate*.

Deciding How to Save the Data

For simplicity I want to have a single Audit Trail table. The way the data is saved is determined by how flexible the procedure needs to be and how the Audit Trail data is going to be used once it has been collected. If the Audit Trail was only ever going to apply to one recordset then perhaps each field in the recordset could be represented individually in the Audit Trail table. This could potentially cause problems since each field in the recordset would require two fields in the Audit Trail table, one for the old value and one for the new value. This could make for a lot of fields and also make interrogating the data quite complicated.

Instead I decided that I needed just three fields: one for the name of the field that was changed; one for the old value; and one for the new value. This means that if, for example, changes were made to five fields in a record, five rows would be added to the Audit Trail table, one for each changed field. I would have to make sure that each row had the same timestamp so that it was clear that all those fields were changed as part of the same operation.

Identifying the Recordset

I don't want to have several different Audit Trails, one for each table, so I need some way to identify which recordset the recorded changes belong to. I could use the form's *RecordSource* property but I decided it would be simpler to record instead the name of the form, which I can read from its *Name* property. In a multi-user environment where there may be numerous different forms working on the same recordset this might also provide additional information about how or by whom the data was changed.

Identifying the Record

It is important to note which record was changed. The best way to do this is to identify the record by its *Primary Key* field. I (almost) always use an *AutoNumber* for the Primary Key so this was the obvious choice.

Identifying the User

In an Access 2003 database (including databases created in later versions but saved in the **.mdb* format) it is possible to use the user-level security tool to set up individual user accounts and user names. In this case *Application.CurrentUser* could be used to return the name of the current user account. However, if user-level security was not set up or if the database is in Access 2007-2010 format (**.accdb*) *Application.CurrentUser* simply returns "Admin". Fortunately, there are alternatives.

My method of choice is to use the *Environ* function *Environ("USERNAME")* which returns the ID of the current user as identified by Windows, normally a person's Windows log-in name. There is some discussion about how secure this method is since this information can be "spoofed" by someone trying to mask their identity (and no, I'm not going to tell you how to do that!). If you think there is a possibility of this happening then you have the option to talk directly to the Windows API. That is outside my personal skill set but there are plenty of methods published, most of them requiring large amounts of code. The simplest way I have found is to use *CreateObject("WScript.Network").UserName*. You can try the various choices and select the one most suitable for your own particular setup.

Capturing Date and Time

The *Now()* function returns the date and time accurate to the second, which is suitable for auditing purposes.

Specifying Which Fields to Check

Again, to make the process as flexible as possible I want to be able to designate specific fields to check. On the form these may be represented by Text Boxes, Combo Boxes, Check Boxes or any of the other types of bound controls. All form controls have a *Tag* property that can accept any text or numeric value. By assigning a specific value to the tag property of the controls I want to record (I am going to use the word "Audit") I can specify which controls get checked. I can also avoid code errors by trying to read a value from the wrong sort of control such as a Label.

So, now we are ready to get started...

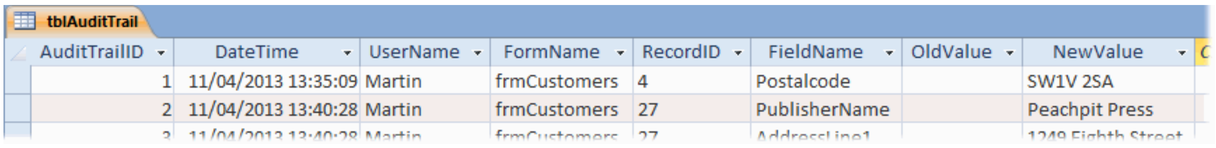
To build an Audit Trail tool that simply records edits, with the option to include or exclude new records you should start here (see: *Build the Audit Trail Table*). If you want you Audit Trail tool to record edits, new records and deletions, you will find a second version of the tool described lower down the page (see *An Alternative Audit Trail Routine for Recording Additions, Edits and Deletes* on page 7).

Build the Audit Trail Table

The first task is to build the table that will receive the records of changes to the database. As I explained earlier, it needs to record the date and time of the change, the identity of the user responsible, and details of the change itself. You'll see that I don't like using spaces in my field names. If you use different field names from the ones shown make sure you make the corresponding changes to the code. Here is a list of the required fields and data types:

Field Name	Data Type
AuditTrailID	AutoNumber (Primary Key)
DateTime	Date/Time
UserName	Text
FormName	Text
RecordID	Text
FieldName	Text
OldValue	Text
NewValue	Text

Note that all the field data types (apart from the Primary Key and the *DateTime* fields) are specified as text. This is to allow the maximum amount of flexibility in the tool since whilst I know that the *UserName*, *FormName* and *FieldName* data will be text, the *RecordID*, *OldValue* and *NewValue* fields might have different data types. The safest way therefore is to store all data as text. It does mean, however, that when interrogating the data you should take account of the fact that you might have dates or numbers stored as text. I have named my table *tblAuditTrail* (*Fig. 1*).



tblAuditTrail

AuditTrailID	DateTime	UserName	FormName	RecordID	FieldName	OldValue	NewValue
1	11/04/2013 13:35:09	Martin	frmCustomers	4	Postalcode		SW1V 2SA
2	11/04/2013 13:40:28	Martin	frmCustomers	27	PublisherName		Peachpit Press
3	11/04/2013 13:40:28	Martin	frmCustomers	27	AddressLine1		1249 Eighth Street

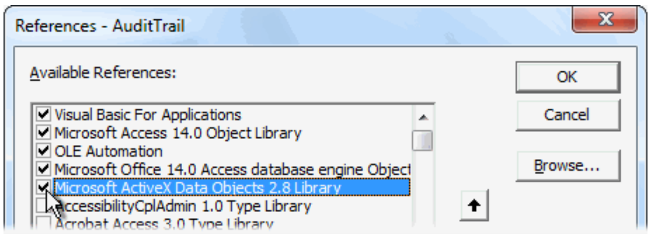
Fig. 1 The Simple Audit Trail table.

Write the Code that Records the Changes

The next task is to create the code that will record the changes to your data. There are a couple of things to do first. Because I have designed the code to be used by any of my forms it does not reside in the form's own code module. Instead it will be placed in a standard VBA module. This means that it will only have to be created once and any form will be able to make use of it.

If you are not familiar with writing VBA code here's what you have to do. From anywhere in your database use the keyboard shortcut **[Alt]+[F11]** to open the Visual Basic Editor. From the **Insert** menu choose **Module** to create a new code module. You should see it appear in the **Project Explorer** window at the left of the screen. The module has been automatically named *Module1*. I like to give my modules meaningful names so I have renamed mine to *basAudit* so I can easily find my code later. This isn't really important but if you want to do the same you can enter the new name by selecting the module and changing its name in the **Properties Window**. (NOTE: If you can't see either the Project Explorer or the Properties Window you can switch them on from the **View** menu.)

Next you have to set a reference to ADO (ActiveX Data Objects) because some of the code will be using this subset of the VBA language. Setting a reference to the ADO code library will allow Access to understand that part of your code. A reference to ADO was set by default in Access 2003 but not in Access 2007 or 2010. Check to see if the reference is set by opening the **Tools** menu and choosing **References**. In the **References** dialog box there will be several items with a tick against them at the top of the list (*Fig. 2*). If there isn't one named *Microsoft ActiveX Data Objects 2.8 Library* (the number may differ slightly e.g. *2.1* in Access 2003) then scroll down the list to find it. Place a tick in the box then click **OK** to set the reference.



References - AuditTrail

Available References:

- [✓] Visual Basic For Applications
- [✓] Microsoft Access 14.0 Object Library
- [✓] OLE Automation
- [✓] Microsoft Office 14.0 Access database engine Object
- [✓] Microsoft ActiveX Data Objects 2.8 Library
- [ ] AccessibilityCplAdmin 1.0 Type Library
- [ ] Acrobat Access 3.0 Type Library

Fig. 2 Set a reference to the ADO Library.

Double-click the module to open it in the main code editing window then add the code shown below (*Listing 1*). You can type the code yourself or copy the text above and paste it into the code module. Make sure that, if you have changed any field names, or given your table a different name, you modify the code accordingly.

Listing 1: A procedure to write data changes to the Audit Trail table

```
Sub AuditChanges(IDField As String)
    On Error GoTo AuditChanges_Err
    Dim cnn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim ctl As Control
    Dim datTimeCheck As Date
    Dim strUserID As String
    Set cnn = CurrentProject.Connection
    Set rst = New ADODB.Recordset
    rst.Open "SELECT * FROM tblAuditTrail", cnn, adOpenDynamic, adLockOptimistic
    datTimeCheck = Now()
    strUserID = Environ("USERNAME")
    For Each ctl In Screen.ActiveForm.Controls
        If ctl.Tag = "Audit" Then
            If Nz(ctl.Value) <> Nz(ctl.OldValue) Then
                With rst
                    .AddNew
                    ![DateTime] = datTimeCheck
                    ![UserName] = strUserID
                    ![FormName] = Screen.ActiveForm.Name
                    ![RecordID] = Screen.ActiveForm.Controls(IDField).Value
                    ![FieldName] = ctl.ControlSource
                    ![OldValue] = ctl.OldValue
                    ![NewValue] = ctl.Value
                    .Update
                End With
            End If
        End If
    Next ctl
AuditChanges_Exit:
    On Error Resume Next
    rst.Close
    cnn.Close
    Set rst = Nothing
    Set cnn = Nothing
    Exit Sub
AuditChanges_Err:
    MsgBox Err.Description, vbCritical, "ERROR!"
    Resume AuditChanges_Exit
End Sub
```

HOW THE CODE WORKS
The AuditChanges code is a self-contained procedure that can be "called" from any form. Along with the name of the procedure I have included a parameter which I have named *IDField*. When calling the procedure you will need to supply a value for this parameter which will be the name of the field that identifies the current record. The first line and the last 3 lines comprise an error handler, telling Access what to do if something goes wrong. The first two *Dim* statements declare the ADO variables that represent a recordset and a database connection. Three more *Dim* statements declare variables representing the controls on the form, the timestamp and the ID of the current user. The two *Set* statements assign values to the ADO variables, the first representing the current database and the second a new recordset. Now things start happening! The next command opens a recordset, described by an SQL statement, that represents the Audit Trail table. At this point a note is made of the time, which is stored in the *datTimeCheck* variable, and of the user ID which is stored in the *strUserID* variable. The block of code starting *For* and ending with *Next* comprises a code loop which visits each control on the currently active form (this means I don't have to specify a form by name). Then comes an *If* statement that checks the control's *Tag* property and only proceeds if the *Tag* reads "Audit". This is important because there are various controls, such as Labels, that don't carry data and these have to be ignored. Assuming the control has been designated as one to be checked, a second *If* statement then compares its *Value* property with its *OldValue* property. Note that I have used the Nz (Null-to-Zero) function because Nulls can cause problems here. If the values are different the a block of ADO code enclosed in a *With* statement adds a new record to the recordset, writes the necessary information into each field, then saves the record before moving to the next control. If the two values are the same the code simply moves straight to the next control. When all the controls have been checked the code performs its exit routine, closing the recordset and the database connection and un-setting their variables, before finally exiting.

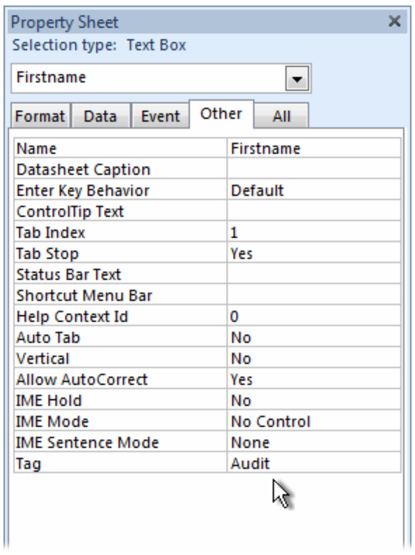
Check the code carefully for spelling errors then open the **Debug** menu and choose **Compile...** This checks the code for errors. If you get an error message then find and fix the problem before compiling again. If everything is OK open the **File** menu and choose **Save** or click the **Save** button to save your code before proceeding to the next step.

Prepare the Forms and Write their Code

There are two tasks to perform on each form whose changes you want to be audited. The controls to be checked must be designated and the code that will call the *AuditChanges* procedure added.

Designate the Controls to be Checked

Open the form in design view then, for each control that you want included in the audit, enter the word *Audit* in its **Tag** property. This is the last item on the **Other** tab of the control's Property Sheet (*Fig. 3*).

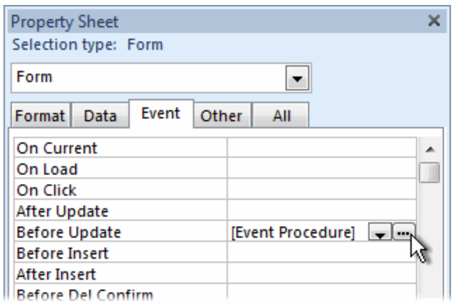


Name	Firstname
Datasheet Caption	
Enter Key Behavior	Default
ControlTip Text	
Tab Index	1
Tab Stop	Yes
Status Bar Text	
Shortcut Menu Bar	
Help Context Id	0
Auto Tab	No
Vertical	No
Allow AutoCorrect	Yes
IME Hold	No
IME Mode	No Control
IME Sentence Mode	None
Tag	Audit

Fig. 3 Set the Tag value in the control's Property Sheet.

Add Code to the Form's BeforeUpdate *Event*

Select the form itself by either clicking the small box where the rulers meet in the upper left corner of the form design window, or by choosing **Form** from the drop-down list at the top of the Property Sheet. On the **Event** tab double-click the box next to **Before Update** so that the text *[Event Procedure]* appears () then click the **Build** button (**[...]**) to open the form's code window.



On Current	
On Load	
On Click	
After Update	
Before Update	[Event Procedure]
Before Insert	
After Insert	
Before Del Confirm	

Fig. 4 Create an event procedure in the form's Property Sheet.

The Visual Basic Editor opens showing the form's code module with and empty *BeforeUpdate* event procedure ready for you to add the necessary code. You have two options here. If you want to audit all changes, including the addition of new records, enter the first code statement shown below (*Listing 2*). You need add only the second of the three lines shown here, don't duplicate the *Private Sub* and *End Sub* statements. IMPORTANT: The example shows *"EmployeeID"* as the parameter value. Change this to the name of the field that identifies the current record, usually the Primary Key field although you can use any field that uniquely identifies the record.

Listing 2: A procedure to call the AuditChanges routine (including new records)

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    Call AuditChanges("EmployeeID")
End Sub
```

If you want to exclude the addition of new records from the audit trail use this code statement instead (*Listing 3*). Again, remember to change the parameter name from *"CustomerID"* to the name of a field that uniquely identifies the current record.

Listing 3: A procedure to call the AuditChanges routine (excluding new records)

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    If Not Me.NewRecord Then Call AuditChanges("CustomerID")
End Sub
```

Check, compile and save your code as described earlier, and you are ready to test your new Audit Trail.

HOW THE CODE WORKS
In the first example (*Listing 2*) This simple statement uses the *Call* keyword to instruct Access to run the *AuditChanges* procedure. Some programmers omit the word *Call*, the instruction will still be understood, but I prefer to use it since it makes it clear to someone reading the code that an external macro is being run. The parameter value is supplied so that the current record can be identified. The second example (*Listing 3*) is modified to include an If Statement that calls the *AuditChanges* macro only if the current record is not a new one.

Job done! The Audit Trail has successfully been added to the database and is ready for use.

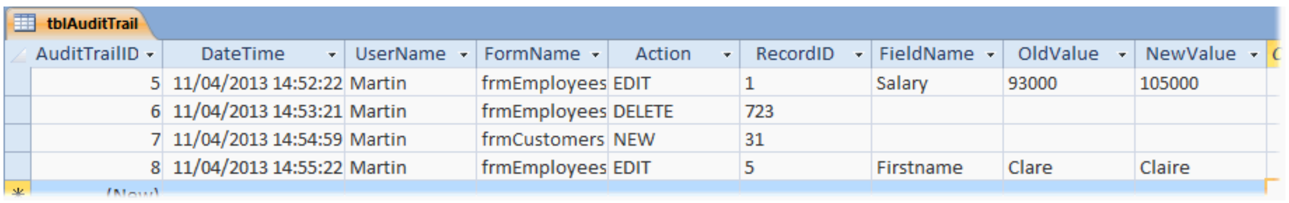
An Alternative Audit Trail Routine for Recording Additions, Edits and Deletes

Create a Modified Audit Trail Table

This version of the Audit Trail tool, in addition to recording edits to existing data, records details of additions to and deletions from the database. In order to do this an additional field is required in the Audit Trail table to identify what type of action the record represents. I have named this field *Action* with the data type *Text*. The modified table has the following fields:

Field Name	Data Type
AuditTrailID	AutoNumber (Primary Key)
DateTime	Date/Time
UserName	Text
FormName	Text
RecordID	Text
Action	Text
FieldName	Text
OldValue	Text
NewValue	Text

This allows the data to clearly indicate what sort of action prompted the Audit Trail entry (*Fig. 5*):


tblAuditTrail

AuditTrailID	DateTime	UserName	FormName	Action	RecordID	FieldName	OldValue	NewValue
5	11/04/2013 14:52:22	Martin	frmEmployees	EDIT	1	Salary	93000	105000
6	11/04/2013 14:53:21	Martin	frmEmployees	DELETE	723			
7	11/04/2013 14:54:59	Martin	frmCustomers	NEW	31			
8	11/04/2013 14:55:22	Martin	frmEmployees	EDIT	5	Firstname	Clare	Claire

Fig. 5 The modified Audit Trail table.

Write the Code that Records Additions, Edits and Deletes

As with the previous version this code uses ADO so the database needs a reference to the appropriate object library (see the instructions above for how to do this). In addition to the *IDField* parameter this version of the *AuditChanges* macro has a second parameter (I have named it *UserAction*) that is used to define the kind of action that prompted the Audit Trail entry. The entire code procedure is shown below (*Listing 4*):

Listing 4: A procedure to record additions, edits and deletes in the Audit Table

```
Sub AuditChanges(IDField As String, UserAction As String)
    On Error GoTo AuditChanges_Err
    Dim cnn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim ctl As Control
    Dim datTimeCheck As Date
    Dim strUserID As String
    Set cnn = CurrentProject.Connection
    Set rst = New ADODB.Recordset
    rst.Open "SELECT * FROM tblAuditTrail", cnn, adOpenDynamic, adLockOptimistic
    datTimeCheck = Now()
    strUserID = Environ("USERNAME")
    Select Case UserAction
        Case "EDIT"
            For Each ctl In Screen.ActiveForm.Controls
                If ctl.Tag = "Audit" Then
                    If Nz(ctl.Value) <> Nz(ctl.OldValue) Then
                        With rst
                            .AddNew
                            ![DateTime] = datTimeCheck
                            ![UserName] = strUserID
                            ![FormName] = Screen.ActiveForm.Name
                            ![Action] = UserAction
                            ![RecordID] = Screen.ActiveForm.Controls(IDField).Value
                            ![FieldName] = ctl.ControlSource
                            ![OldValue] = ctl.OldValue
                            ![NewValue] = ctl.Value
                            .Update
                        End With
                    End If
                End If
            Next ctl
        Case Else
            With rst
                .AddNew
                ![DateTime] = datTimeCheck
                ![UserName] = strUserID
                ![FormName] = Screen.ActiveForm.Name
                ![Action] = UserAction
                ![RecordID] = Screen.ActiveForm.Controls(IDField).Value
                .Update
            End With
    End Select
AuditChanges_Exit:
    On Error Resume Next
    rst.Close
    cnn.Close
    Set rst = Nothing
    Set cnn = Nothing
    Exit Sub
AuditChanges_Err:
    MsgBox Err.Description, vbCritical, "ERROR!"
    Resume AuditChanges_Exit
End Sub
```

Write or copy and paste the code into a standard module so that it can be called from any form.

HOW THE CODE WORKS
The code differs from that described earlier after the *tblAuditTrail* recordset has been opened but before a new record is created. A Case Statement examines the second parameter (*UserAction*) and enters data into a new record accordingly. If the parameter indicates that the action is an EDIT all the appropriate information is written into the new record with the addition of the word "EDIT" (the value of the *UserAction* parameter) in the *Action* field. The only additional cases to consider are NEW and DELETE and since these require the same data entry requirements they are dealt with together in the *Case Else* part of the Case Statement. For new records and deletions there is no *Value* or *OldValue* data to be entered so there is no need to loop through the controls. the procedure simply records the timestamp, user ID and form name together with the ID of the record concerned and type of action read from the *UserAction* parameter. The error handler and exit routine are the same as before.

Prepare the Form and Add its Code

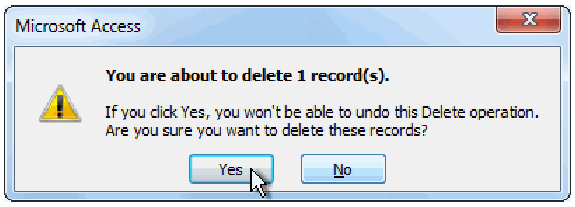
The *Tag* property of each control to be audited must be set to *Audit* as described in the previous example. As this example requires more actions to be audited the code that calls the *AuditChanges* routine is a little more complex. Again, the *BeforeUpdate* event is used to call the *AuditChanges* macro when a record is saved, which happens when a new record is added or an existing record is edited (*Listing 5*):

Listing 5: A procedure to call the AuditChanges routine (new and existing records)

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    If Me.NewRecord Then
        Call AuditChanges("EmployeeID", "NEW")
    Else
        Call AuditChanges("EmployeeID", "EDIT")
    End If
End Sub
```

HOW THE CODE WORKS
The *BeforeUpdate* event fires when data is saved to the underlying recordset. Access is able to determine whether this data applies to an existing record or if it constitutes a new record. The If Statement checks whether or not a new record is being created and calls the *AuditChanges* macro specifying the record's ID field and either "NEW" or "EDIT" as appropriate.

Deleting a record does not cause the *BeforeUpdate* event to fire. Instead, the *AfterDelConfirm* event is used. You might notice that there is an *OnDelete* event but this is not appropriate because it fires regardless of whether or not the user confirms the deletion when Access displays the usual warning (*Fig. 6*). So if *OnDelete* is used the code would run even if the user cancelled the deletion when asked. The *AfterDelConfirm* event fires only when the user confirms the deletion, so this is used to call the *AuditChanges* macro (*Listing 6*).


Microsoft Access

You are about to delete 1 record(s).

If you click Yes, you won't be able to undo this Delete operation.
Are you sure you want to delete these records?

Fig. 6 Access requires confirmation if the user deletes a record.

Listing 6: A procedure to call the AuditChanges routine (deleted records)

```
Private Sub Form_AfterDelConfirm(Status As Integer)
    If Status = acDeleteOK Then Call AuditChanges("EmployeeID", "DELETE")
End Sub
```

HOW THE CODE WORKS
The *AfterDelConfirm* event fires when the user dismisses the warning message by choosing one of the options (if they press the **[ESC]** key or use the message's Close button it assumes *No*). If the user chooses *Yes* the message returns the value *acDeleteOK* to the event procedure's *Status* parameter. A simple If Statement checks to see if this is the case and, if so, calls the *AuditChanges* macro specifying the record's ID field and "DELETE".

Download Example Databases

You can download samples of databases equipped with the databases described in this tutorial. The "Simple" database includes the first example in the tutorial in which the Audit Trail records edits to existing data with the option to exclude the recording of new records. The "Detailed" database includes the second example in the tutorial in which the Audit Trail records edits to existing data as well as the addition and deletion of records.

To get the files visit the online version of this tutorial at:

http://www.fontstuff.com/access/acctut21.htm