Access Tutorial 13: Event-Driven Programming Using Macros

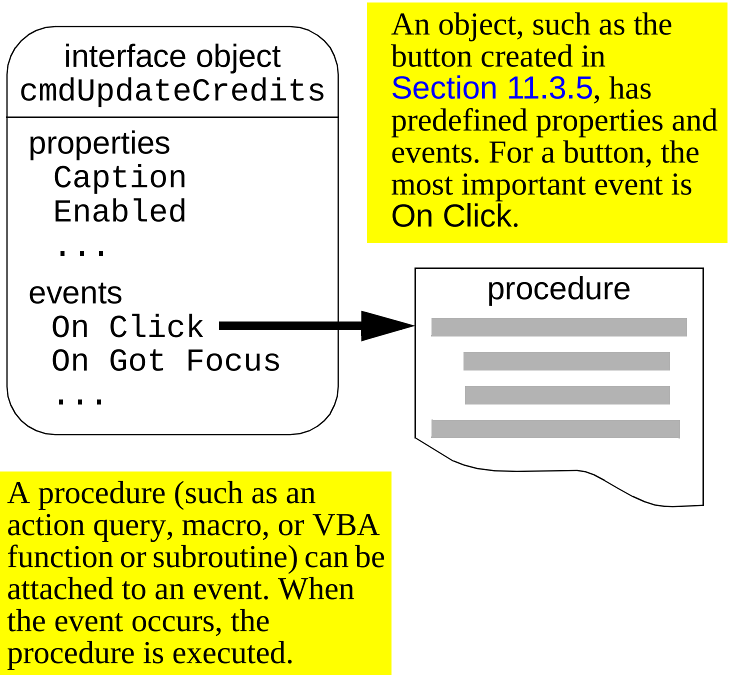
13.1 Introduction: What is event-driven programming?

In conventional programming, the sequence of operations for an application is determined by a central controlling program (e.g., a main procedure). In **event-driven** programming, the sequence of operations for an application is determined by the user's interaction with the application's interface (forms, menus, buttons, etc.).

For example, rather than having a main procedure that executes an order entry module followed by a data verification module followed by an inventory update module, an event-driven application remains in the background until certain events happen: when a value in a field is modified, a small data verification program is executed; when the user indicates that the order entry is complete, the inventory update module is executed, and so on.

Event-driven programming, graphical user interfaces (GUIs), and object-orientation are all related since forms (like those created in Tutorial 6) and the graphical interface objects on the forms serve as the skeleton for the entire application. To create an event-driven application, the programmer creates small programs and attaches them to events associated with objects, as shown in Figure 13.1. In this way, the behavior of the application is determined by the interaction of a number of small manageable programs rather than one large program.

FIGURE 13.1: In a trigger, a procedure is attached to an event.


interface object
cmdUpdateCredits
properties
Caption
Enabled
...
events
On Click
On Got Focus
...
procedure
An object, such as the button created in Section 11.3.5, has predefined properties and events. For a button, the most important event is On Click.
A procedure (such as an action query, macro, or VBA function or subroutine) can be attached to an event. When the event occurs, the procedure is executed.

13.1.1 Triggers

Since events on forms “trigger” actions, event/procedure combinations are sometimes called **triggers**.

For example, the action query you attached to a button in Section 11.3.5 is an example of a simple, one-action trigger. However, since an action query can only perform one type of action, and since you typically have a number of actions that need to be performed, macros or Visual Basic procedures are typically used to implement a triggers in Access.

13.1.2 The Access macro language

As you discovered in Tutorial 12, writing simple VBA programs is not difficult, but it is tedious and error-prone. Furthermore, as you will see in Tutorial 14, VBA programming becomes much more difficult when you have to refer to objects using the naming conventions of the database object hierarchy. As a consequence, even experienced Access programmers often turn to the Access macro language to implement basic triggers.

The macro language itself consists of 40 or so commands. Although it is essentially a procedural language (like VBA), the commands are relatively high level and easy to understand. In addition, the macro editor simplifies the specification of the **action arguments** (parameters).

13.1.3 The trigger design cycle

To create a trigger, you need to answer two questions:

1. What has to happen?
2. When should it happen?

Once you have answered the first question ("what"), you can create a macro (or VBA procedure) to execute the necessary steps. Once you know the answer to the second question ("when"), you can attach the procedure to the correct event of the correct object.

⚠ Selecting the correct object and the correct event for a trigger is often the most difficult part of creating an event-driven application. It is best to think about this carefully before you get too caught up in implementing the procedure.

13.2 Learning objectives

- ❒ What is event-driven programming? What is a trigger?
- ❒ How do I design a trigger?
- ❒ How does the macro editor in Access work?
- ❒ How do I attach a macro to an event?
- ❒ What is the SetValue action? How is it used?

- ❒ How do I make the execution of particular macro actions conditional?
- ❒ What is a switchboard and how do I create one for my application?
- ❒ How to I make things happen when the application is opened?
- ❒ What are the advantages and disadvantages of event-driven programming?

13.3 Tutorial exercises

In this tutorial, you will build a number of very simple triggers using Access macros. These triggers, by themselves, are not particularly useful and are intended for illustrative purposes only.

13.3.1 The basics of the macro editor

In this section, you are going to eliminate the warning messages that precede the trigger you created Section 11.3.5.

As such, the answer to the “what” question is the following:

1. Turn off the warnings so the dialog boxes do not pop up when the action query is executed;
2. Run the action query; and,
3. Turn the warnings back on (it is generally good programming practice to return the environment to its original state).

Since a number of things have to happen, you cannot rely on an action query by itself. You can, however, execute a macro that executes several actions including one or more action queries.

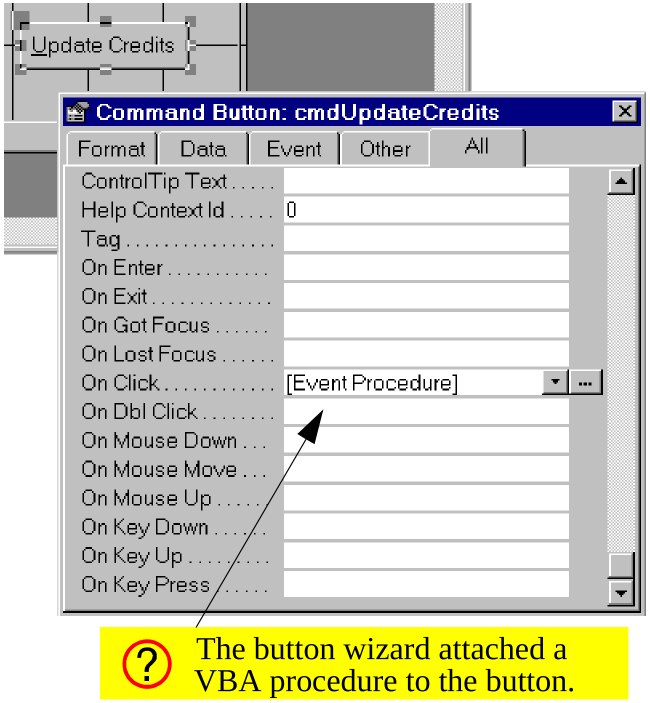
- Select the *Macros* tab from the database window and press *New*. This brings up the macro editor shown in Figure 13.2.
- Add the three commands as shown in Figure 13.3. Note that the `OpenQuery` command is used to run the action query.
- Save the macro as `mcrUpdateCredits` and close it.

13.3.2 Attaching the macro to the event

The answer to the “when” question is: When the `cmdUpdateCredits` button is pressed. Since you already created the button in Section 11.3.5, all you need to do is modify its *On Click* property to point the `mcrUpdateCredits` macro.

- Open `frmDepartments` in design mode.
- Bring up the property sheet for the button and scroll down until you find the *On Click* property, as shown in Figure 13.4.

FIGURE 13.4: Bring up the *On Click* property for the button.



ControlTip Text	
Help Context Id	0
Tag	
On Enter	
On Exit	
On Got Focus	
On Lost Focus	
On Click	[Event Procedure]
On Dbl Click	
On Mouse Down	
On Mouse Move	
On Mouse Up	
On Key Down	
On Key Up	
On Key Press	

The button wizard attached a VBA procedure to the button.

FIGURE 13.2: The macro editor.

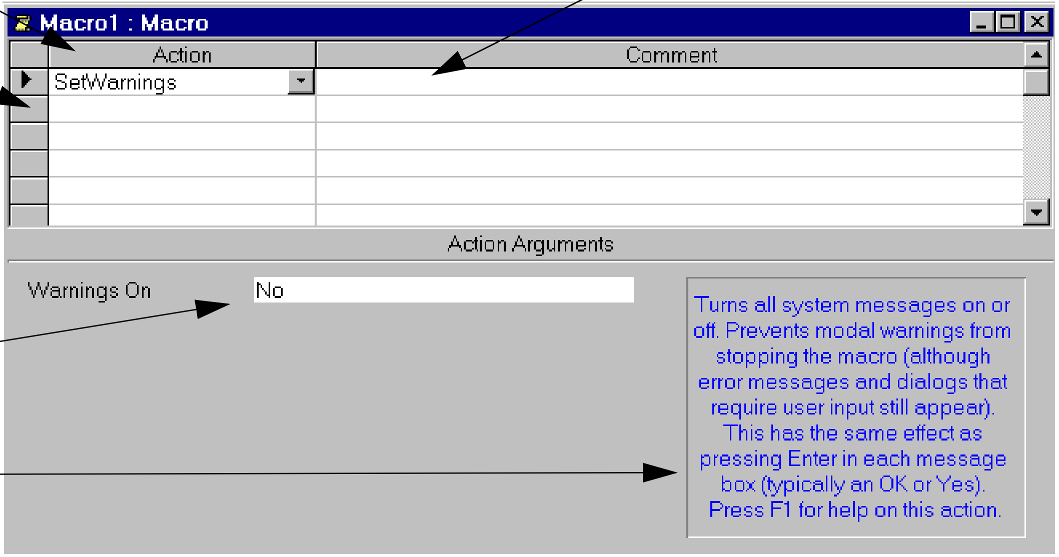
Macro actions can be selected from a list. The `SetWarnings` command is used to turn the warning messages (e.g., before you run an action query) on and off.

In the comment column, you can document your macros as required

Multiple commands are executed from top to bottom.

Most actions have one or more arguments that determine the specific behavior of the action. In this case, the `SetWarnings` action is set to turn warnings off.

The area on the right displays information about the action.



Action	Comment
SetWarnings	

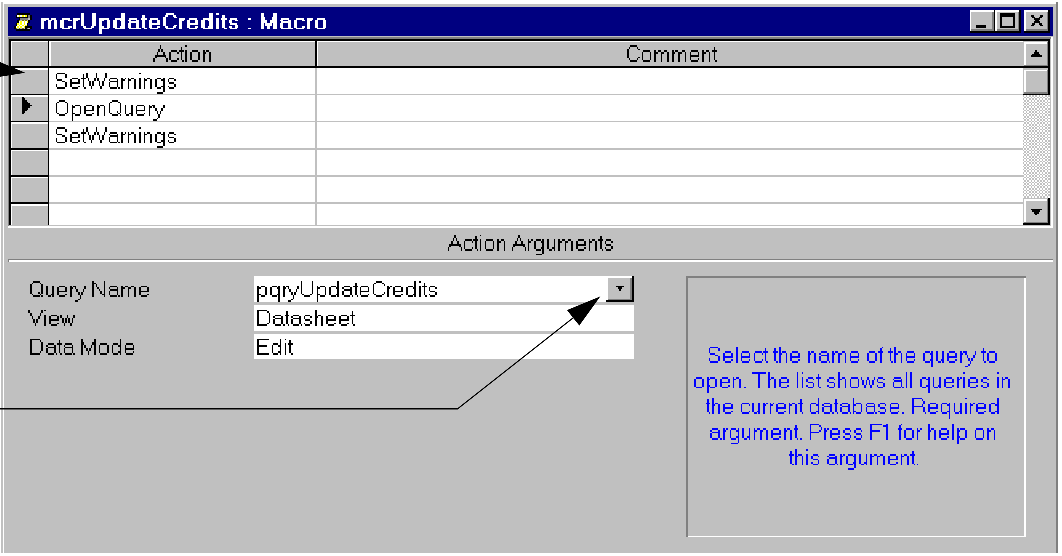
Warnings On: No

Turns all system messages on or off. Prevents modal warnings from stopping the macro (although error messages and dialogs that require user input still appear). This has the same effect as pressing Enter in each message box (typically an OK or Yes). Press F1 for help on this action.

FIGURE 13.3: Create a macro that answers the "what" question.

a Add the three commands to the macro.

b The arguments for the two `SetWarnings` actions are straightforward. For the `OpenQuery` command, you can select the query to open (or run) from a list. Since this is an action query, the second and third arguments are not applicable.



Action	Comment
SetWarnings	
OpenQuery	
SetWarnings	

Action Arguments

Query Name	pqryUpdateCredits
View	Datasheet
Data Mode	Edit

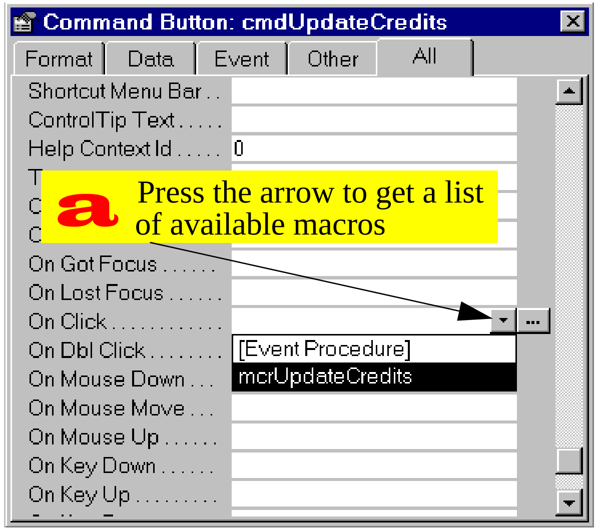
Select the name of the query to open. The list shows all queries in the current database. Required argument. Press F1 for help on this argument.

- Press the builder button (...) beside the existing procedure and look at the VBA subroutine created by the button wizard. Most of this code is for error handling.

? Unlike the stand-along VBA modules you created in Tutorial 12, this module (collection of functions and subroutines) is embedded in the `frmDepartments` form.

- Since you are going to replace this code with a macro, you do not want it taking up space in your database file. Highlight the text in the subroutine and delete it. When you close the module window, you will see the reference to the "event procedure" is gone.
- Bring up the list of choice for the *On Click* property as shown in Figure 13.5. Select `mcrUpdateCredits`.

FIGURE 13.5: Select the macro to attach to the *On Click* property.


Command Button: cmdUpdateCredits

Property	Value
Shortcut Menu Bar	
ControlTip Text	
Help Context Id	0
On Got Focus	
On Lost Focus	
On Click	
On Dbl Click	[Event Procedure]
On Mouse Down	mcrUpdateCredits
On Mouse Move	
On Mouse Up	
On Key Down	
On Key Up	

- Switch to form view and press the button. Since no warnings appear, you may want to press the button a few times (you can always use your roll-back query to reset the credits to their original values).

13.3.3 Creating a check box to display update status information

Since the warning boxes have been disabled for the update credits trigger, it may be useful to keep track of whether courses in a particular department have already been updated.

To do this, you can add a field to the `Departments` table to store this “update status” information.

- Edit the `Departments` table and add a *Yes/No* field called `CrUpdated`.

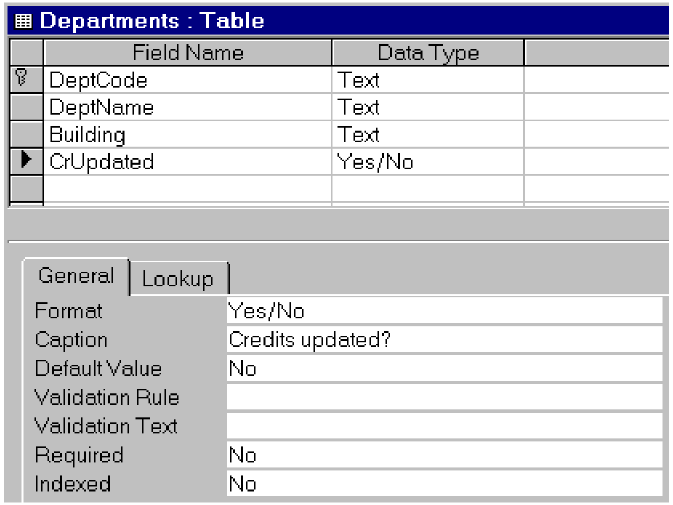
If you have an open query or form based on the `Departments` table, you will not be able to modify the structure of the table until the query or form is closed.

- Set the *Caption* property to `Credits updated?` and the *Default* property to `No` as shown in Figure 13.6.

Changes made to a table do not automatically carry over to forms already based on that table. As such, you must manually add the new field to the departments form.

- Open `frmDepartments` in design mode.
- Make sure the toolbox and field list are visible. Notice that the new field (`CrUpdated`) shows up in the field list.
- Use the same technique for creating combo boxes to create a bound check box control for the yes/no field. This is shown in Figure 13.7.

FIGURE 13.6: Add a field to the `Departments` table to record the status of updates.



Field Name	Data Type
DeptCode	Text
DeptName	Text
Building	Text
CrUpdated	Yes/No

Format	Yes/No
Caption	Credits updated?
Default Value	No
Validation Rule	
Validation Text	
Required	No
Indexed	No

13.3.4 The `SetValue` command

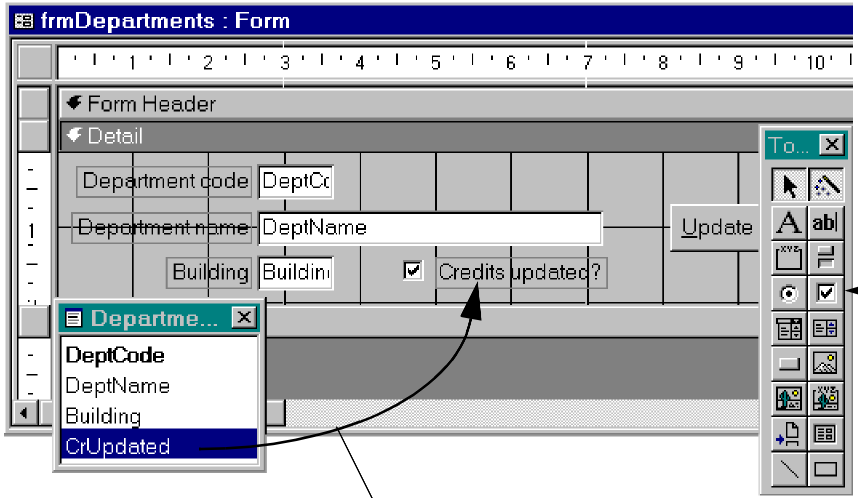
So far, you have used two commands in the Access macro language: `SetWarnings` and `OpenQuery`. In this section, you are going to use one of the most useful commands—`SetValue`—to automatically change the value of the `CrUpdated` check box.

- Open your `mcrUpdateCredits` macro in design mode and add a `SetValue` command to change the `CrUpdated` check box to `Yes` (or `True`, if you prefer). This is shown in Figure 13.8.
- Save the macro and press the button on the form. Notice that the value of the check box changes, reminding you not to update the courses for a particular department more than once.

13.3.5 Creating conditional macros

Rather than relying on the user not to run the update when the check box is checked, you may use a **conditional macro** to *prevent* an update when the check box is checked.

FIGURE 13.7: Add a check box control to keep track of the update status.

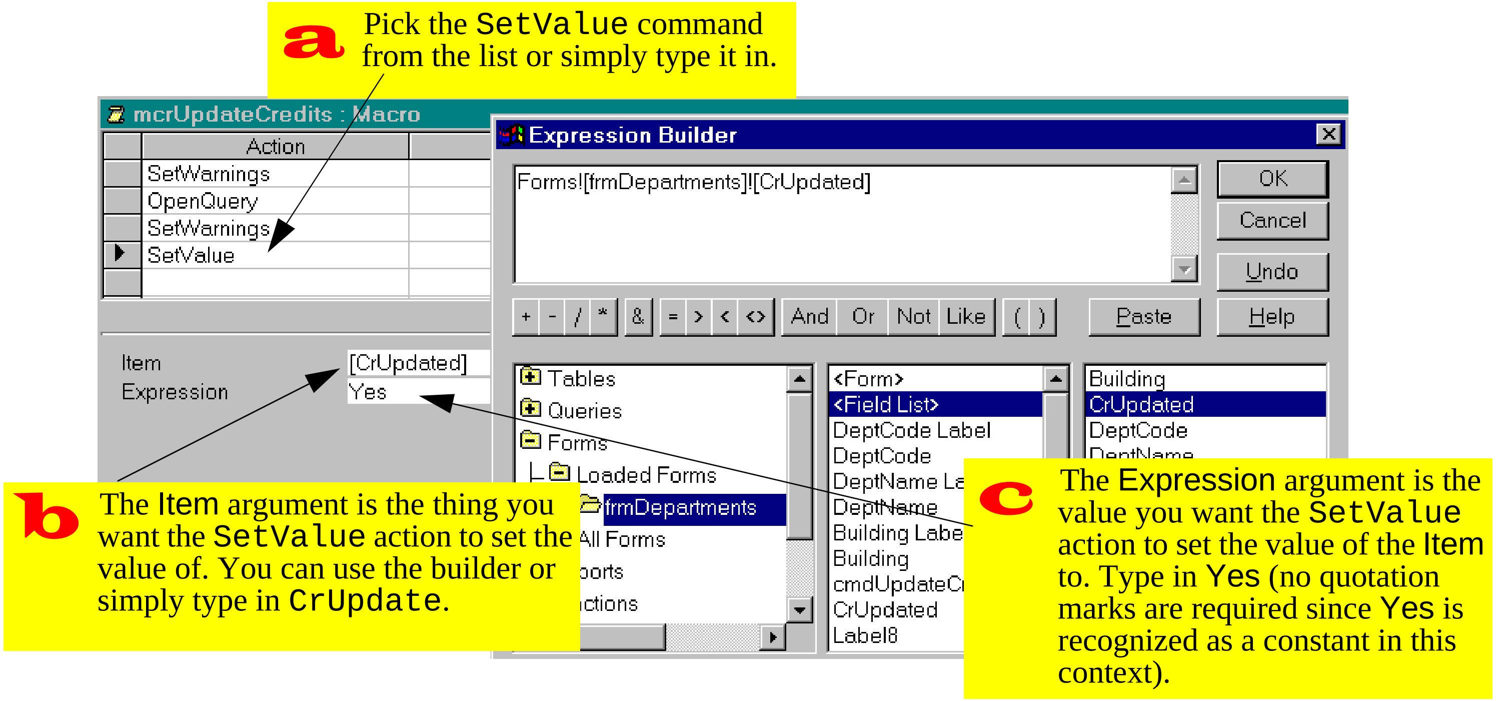


a Select the check box tool from the toolbox.

b Drag the `CrUpdated` field from the field list to the detail section.

? A check box is a control that can be bound to fields of the yes/no data type. When the box is checked, `True` is stored in the table; when the box is unchecked, `False` is stored.

FIGURE 13.8: **Add a `SetValue` command to set the value of the update status field when the update is compete.**


a Pick the `SetValue` command from the list or simply type it in.

b The Item argument is the thing you want the `SetValue` action to set the value of. You can use the builder or simply type in `CrUpdate`.

c The Expression argument is the value you want the `SetValue` action to set the value of the Item to. Type in Yes (no quotation marks are required since Yes is recognized as a constant in this context).

mcrUpdateCredits : Macro

Action
SetWarnings
OpenQuery
SetWarnings
SetValue

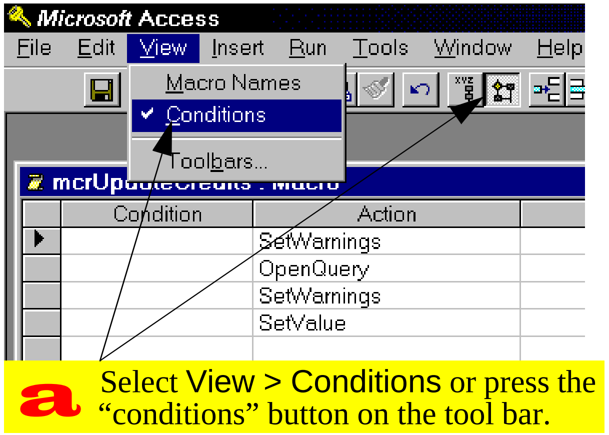
Item [CrUpdated]
Expression Yes

Expression Builder

```
Forms![frmDepartments]![CrUpdated]
```

- Select *View > Conditions* to display the conditions column in the macro editor as shown in Figure 13.9.

FIGURE 13.9: Display the macro editors condition column



13.3.5.1 The simplest conditional macro

If there is an expression in the condition column of a macro, the action in that row will execute if the condition is true. If the condition is not true, the action will be skipped.

- Fill in the condition column as shown in Figure 13.10. Precede the actions you want to execute if the check box is checked with `[CrUpdated]`. Precede the actions you do not want to execute with `Not [CrUpdated]`.

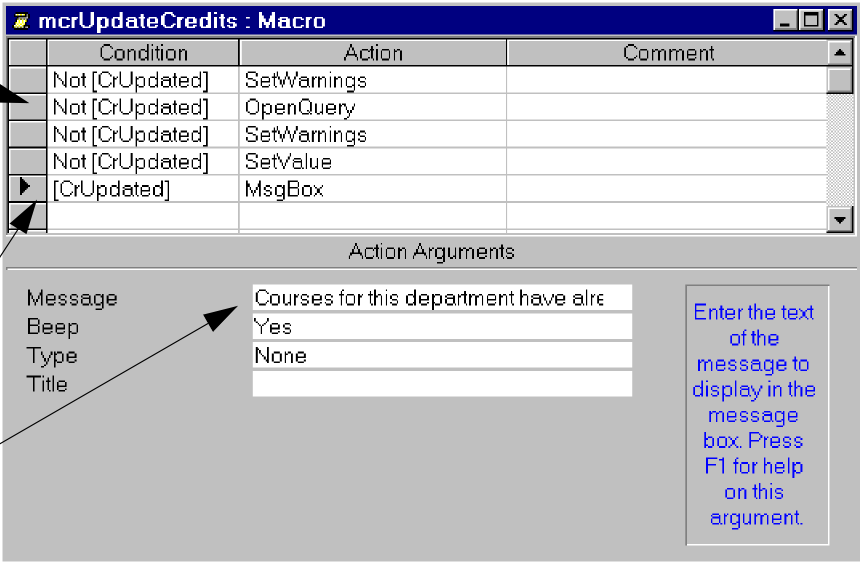
? Since `CrUpdated` is a **Boolean** (yes/no) variable, you do not need to write `[CrUpdated] = True` or `[CrUpdated] = False`. The true and false parts are implied. However, if a non-Boolean data type is used in the expression, a comparison operator must be included (e.g., `[DeptCode] = "COMM"`, `[Credits] < 3`, etc.)

FIGURE 13.10: Create a conditional macro to control which actions execute.

a The expression `Not [CrUpdated]` is true if the `CrUpdated` check box is not checked. Use this expression in front of the actions you want to execute in this situation.

b The expression `[CrUpdated]` is true if the `CrUpdated` check box is checked. In this situation, you should indicate to the user that the update is not being performed.

c The `MsgBox` action displays a standard Windows message box. You can set the message and other message box features in the arguments section.



Condition	Action	Comment
Not [CrUpdated]	SetWarnings	
Not [CrUpdated]	OpenQuery	
Not [CrUpdated]	SetWarnings	
Not [CrUpdated]	SetValue	
[CrUpdated]	MsgBox	

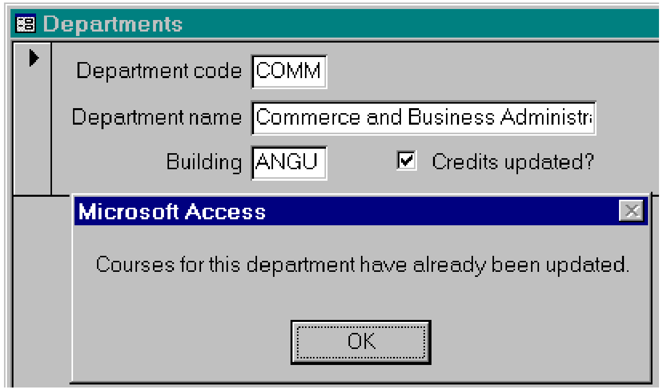
Action Arguments

Message	Courses for this department have alre
Beep	Yes
Type	None
Title	

Enter the text of the message to display in the message box. Press F1 for help on this argument.

- Switch to the form and test the macro by pressing the button. If the `CrUpdated` check box is checked, you should get a message similar to that shown in Figure 13.11.

FIGURE 13.11: The action query is not executed and the message box appears instead.



Department code COMM
Department name Commerce and Business Administr
Building ANGU
☑ Credits updated?

Courses for this department have already been updated.

13.3.5.2 Refining the conditions

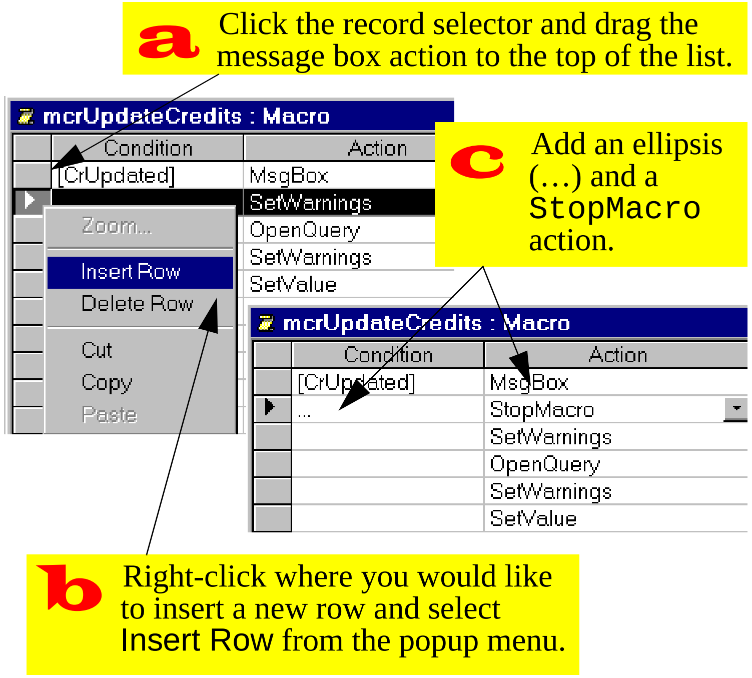
The macro shown in Figure 13.10 can be improved by using an ellipsis (…) instead of repeating the same condition in line after line. In this section, you will simplify your conditional macro slightly.

Move the message box action and condition to the top of the list of actions by dragging its record selector (grey box on the left).

- Insert a new row immediately following the message and add a `StopMacro` action, as shown in Figure 13.12.

The macro in Figure 13.12 executes as follows: If `CrUpdate` is true (i.e., the box is checked), the `MsgBox` action executes. Since the next line has an ellipsis in the condition column, the condition continues to apply. However, that action on the ellipsis line is `StopMacro`, and thus the macro ends without executing the next four lines.

FIGURE 13.12: Rearrange the macro actions and insert a new row.



If the `CrUpdate` box is not checked, the first two lines are ignored (i.e., the lines with the false condition and the ellipsis) and the update proceeds.

13.3.5.3 Creating a group of named macros

It is possible to store a number of related macros together in one macro “module”. These **group macros** have two advantages:

1. **Modular macros can be created** — instead of having a large macro with many conditions and branches, you can create a small macro that call other small macros.
2. **Similar macros can be grouped together** — for example, you could keep all you `Departments`-related macros or search-related macros in a macro group.

In this section, we will focus on the first advantage.

- Select *View > Macro Names* to display the macro name column.

- Perform the steps in Figure 13.13 to modularize your macro.
- Change the macro referred to in the *On Click* property of the `cmdUpdateCredits` button from `mcrUpdateCredits` to `mcrUpdateCredits.CheckStatus`.
- Test the operation of the button.

13.3.6 Creating switchboards

One of the simplest (but most useful) triggers is an `OpenForm` command attached to a button on a form consisting exclusively of buttons.

This type of “switchboard” (as shown in Figure 13.14) can provide the user with a means of navigating the application.

- Create an unbound form as shown in Figure 13.15.
- Remove the scroll bars, navigation buttons, and record selectors from the form using the form’s property sheet.
- Save the form as `swbMain`.

There are two ways to add button-based triggers to a form:

1. Turn the button wizard off, create the button, and attach an macro containing the appropriate action (or actions).
2. Turn the button wizard on and use the wizard to select from a list of common actions (the wizard writes a VBA procedure for you).

? Since the wizard can only attach one action to a button (such as opening a form or running an action query) it is less flexible than a macro. However, once you are more comfortable with VBA, there is nothing to stop you

FIGURE 13.13: **Use named macros to modularize the macro.**

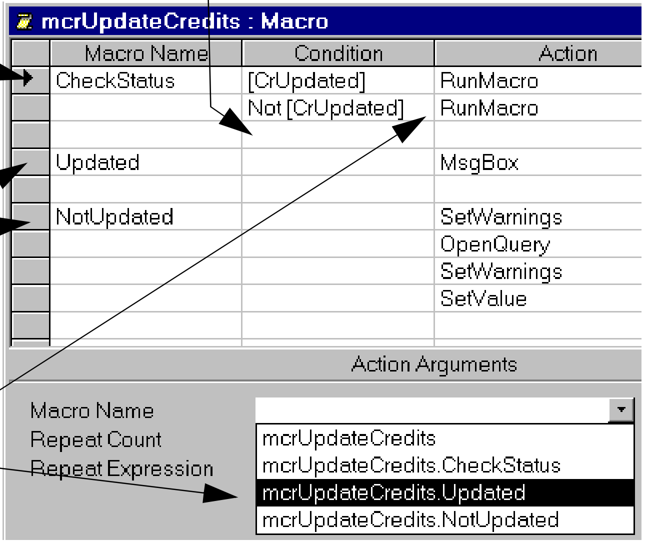
a Select View > Macro Names to display the macro names column.

? A macro executes until it encounters a blank line. Use blank lines to separate the named macros within a group.

b Create a named macro called `CheckStatus` that contains the conditional logic for the procedure.

c Create two other macros, `Updated` and `NotUpdated` that correspond to the logic in the `CheckStatus` macro.

d The `RunMacro` action executes a particular macro. Select the macro to execute from a list in the arguments pane. Note the naming convention for macros within a macro group.



mcrUpdateCredits : Macro

Macro Name	Condition	Action
CheckStatus	[CrUpdated]	RunMacro
	Not [CrUpdated]	RunMacro
Updated		MsgBox
NotUpdated		SetWarnings
		OpenQuery
		SetWarnings
		SetValue

Action Arguments

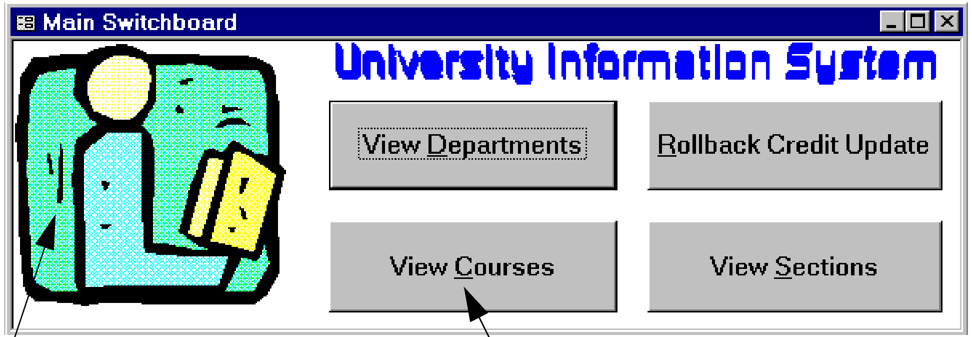
Macro Name	
Repeat Count	
Repeat Expression	

mcrUpdateCredits
mcrUpdateCredits.CheckStatus
mcrUpdateCredits.Updated
mcrUpdateCredits.NotUpdated

FIGURE 13.14: A switchboard interface to the application.

The command buttons are placed on an unbound form. Note the absence of scroll bars, record selectors, or navigation buttons.

Although it is not shown here, switchboards can call other switchboards, allowing you to add a hierarchical structure to your application.



Gratuitous clip art can be used to clutter your forms and reduce the application's overall performance.

Shortcut keys are include on each button to allow the user to navigate the application with keystrokes.

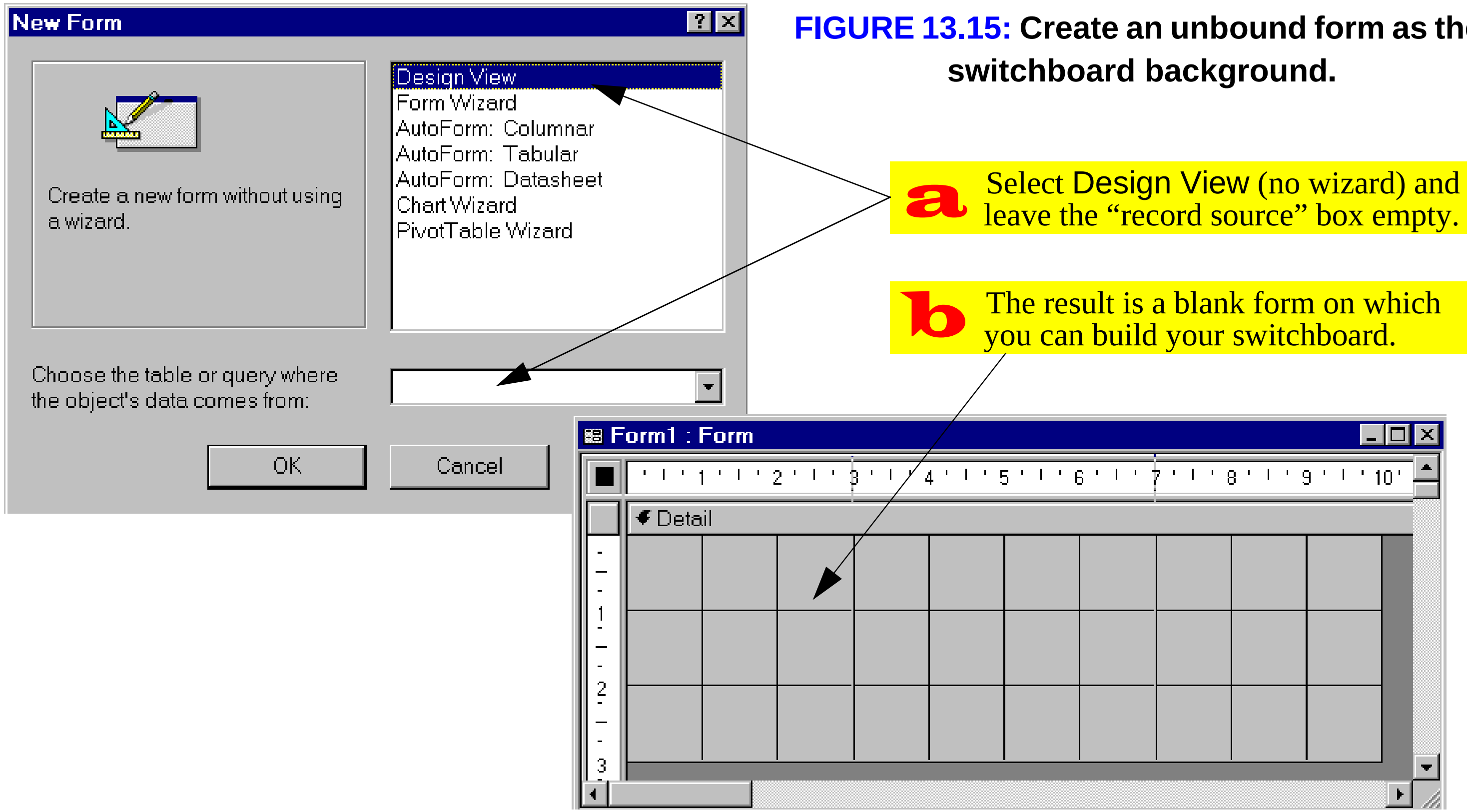


FIGURE 13.15: Create an unbound form as the switchboard background.

a Select Design View (no wizard) and leave the "record source" box empty.

b The result is a blank form on which you can build your switchboard.

from editing the VBA modules created by the wizard to add additional functionality.

13.3.6.1 Using a macro and manually-created buttons

- Ensure the wizard is turned off and use the button tool to create a button.
- Modify the properties of the button as shown in Figure 13.16.
- Create a macro called `mcrSwitchboard.OpenDept` and use the `OpenForm` command to open the form `frmDepartments`.
- Attach the macro to the *On Click* event of the `cmdDepartments` button.
- Test the button.

13.3.6.2 Using the button wizard

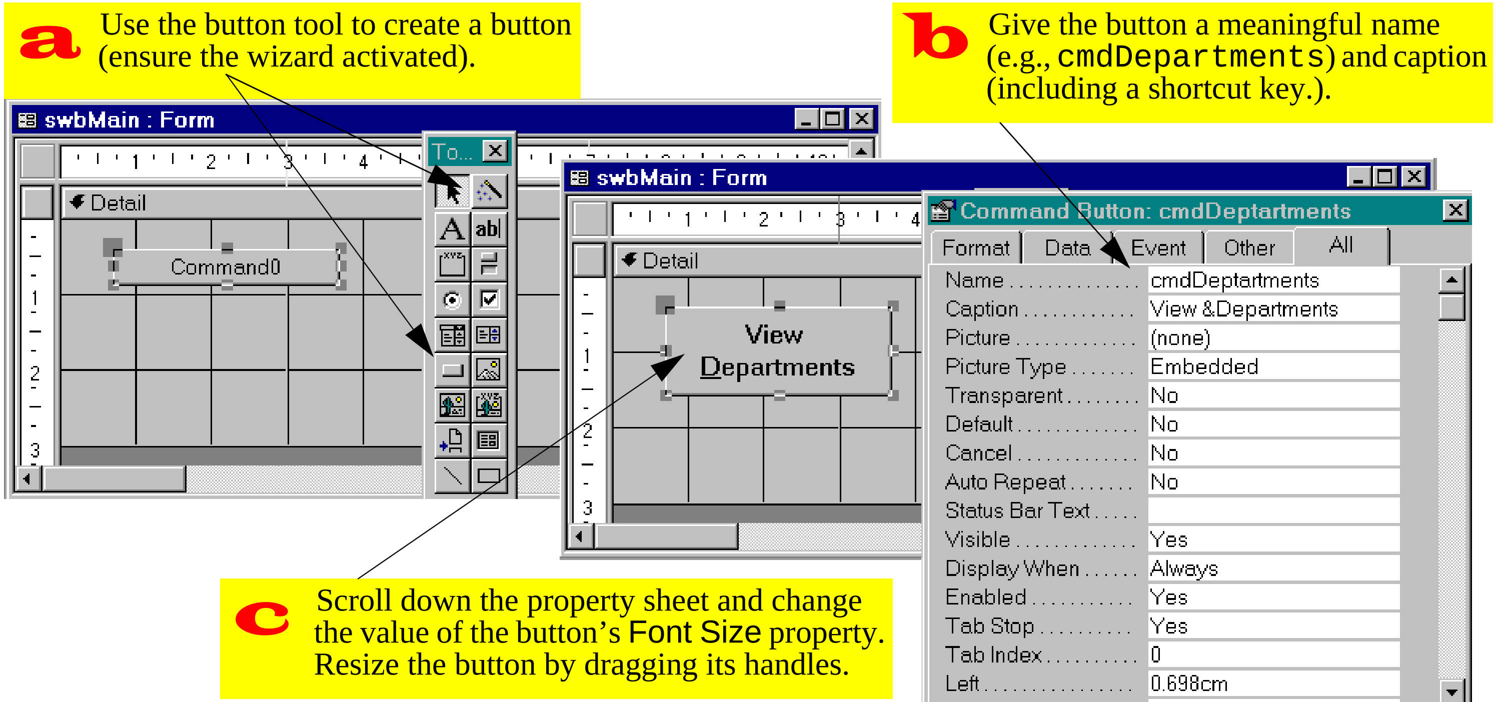
- Turn the button wizard back on and create a new button.
- Follow the directions provided by the wizard to set the action for the button (i.e., open the `frmCourses` form) as shown in Figure 13.17.
- Change the button's font and resize it as required.

You can standardize the size of your form objects by selecting more than one and using *Format > Size > to Tallest* and *to Widest* commands. Similarly, you can select more than one object and use the "multiple selection" property sheet to set the properties all at once.

13.3.7 Using an `autoexec` macro

If you use the name `autoexec` to save a macro (in lieu of the normal `mcr<name>` convention), Access will execute the macro actions when the database is opened. Consequently, auto-execute macros are

FIGURE 13.16: Create a button and modify its appearance.



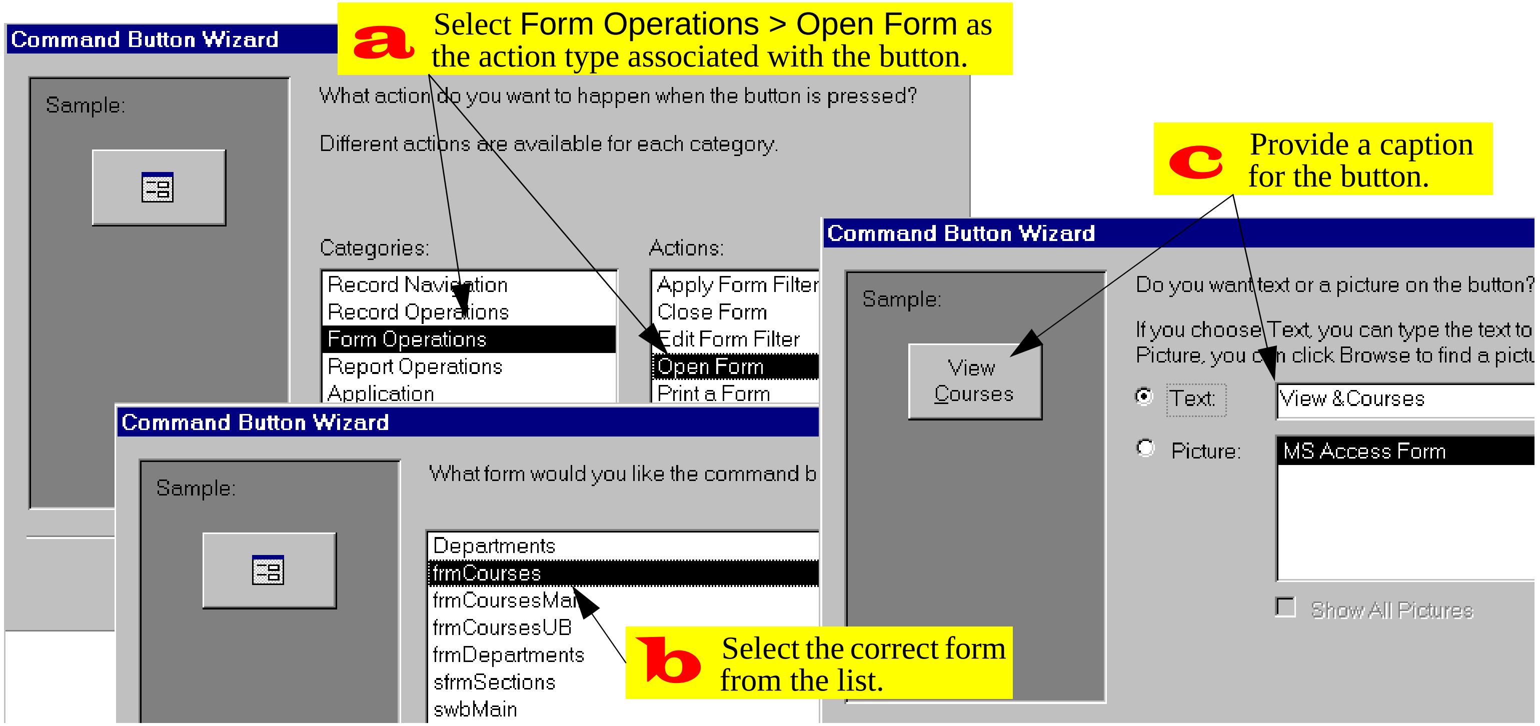
a Use the button tool to create a button (ensure the wizard activated).

b Give the button a meaningful name (e.g., `cmdDepartments`) and caption (including a shortcut key.).

c Scroll down the property sheet and change the value of the button's Font Size property. Resize the button by dragging its handles.

Name	cmdDeptartments
Caption	View &Departments
Picture	(none)
Picture Type	Embedded
Transparent	No
Default	No
Cancel	No
Auto Repeat	No
Status Bar Text	
Visible	Yes
Display When	Always
Enabled	Yes
Tab Stop	Yes
Tab Index	0
Left	0.698cm

FIGURE 13.17: **Use the command button wizard to create a button for the switchboard.**



a Select Form Operations > Open Form as the action type associated with the button.

Command Button Wizard

Sample:

What action do you want to happen when the button is pressed?

Different actions are available for each category.

Categories:
- Record Navigation
- Record Operations
- Form Operations
- Report Operations
- Application

Actions:
- Apply Form Filter
- Close Form
- Edit Form Filter
- Open Form
- Print a Form

b Select the correct form from the list.

Command Button Wizard

Sample:

What form would you like the command b

- Departments
- frmCourses
- frmCoursesMai
- frmCoursesUB
- frmDepartments
- sfrmSections
- swbMain

c Provide a caption for the button.

Command Button Wizard

Sample:

View Courses

Do you want text or a picture on the button?

If you choose Text, you can type the text to Picture, you can click Browse to find a pictu

(●) Text: View &Courses

( ) Picture: MS Access Form

[ ] Show All Pictures

often used to display a switchboard when the user starts the application.

Another typical auto-execute operation is to hide the database window. By doing this, you unclutter the screen and reduce the risk of a user accidentally making a change to the application (by deleting a database object, etc.).

To unhide the database window, select *Window > Unhide* from the main menu or press the database window icon (▤) on the toolbar.

The problem with hiding the database window using a macro is that there is no `HideDatabaseWindow` command in the Access macro language. As such, you have to rely on the rather convoluted `DoMenuItem` action.

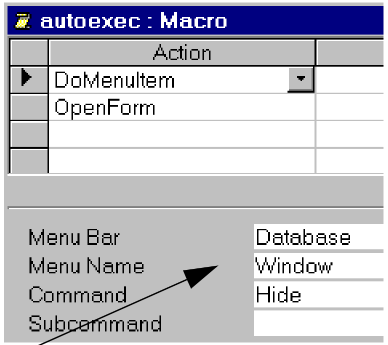
As its name suggests, the `DoMenuItem` action performs an operation just as if it had been selected from the menu system. Consequently, you need to know something about the menu structure of Access before you create your macro.

In version 8.0, the `DoMenuItem` action has been replaced by the slightly more intuitive `RunCommand` action. See on-line help for more information on `RunCommand`.

- Create an auto-execute macro
- Add the `DoMenuItem` and `OpenForm` actions to hide the database window and open the main switchboard, as shown in Figure 13.18.
- Close the database and reopen it after a short delay to test the macro.

In version 7.0 and above, you do not need to use an autoexec macro to hide the database window and open a form. Instead, you can right-click on the database window, select

FIGURE 13.18: **Create an auto-execute macro.**


autoexec : Macro

Action
DoMenuItem
OpenForm

Menu Bar	Database
Menu Name	Window
Command	Hide
Subcommand	

a For the `DoMenuItem` action, select the Window > Hide commands from the Database menu (i.e., the menu that is active when the database window is being used).

Startup, and fill in the properties for the application.

13.4 Discussion

13.4.1 Event-driven programming versus conventional programming

The primary advantages of event-driven programming are the following:

1. **Flexibility** — since the flow of the application is controlled by events rather than a sequential program, the user does not have to conform to the programmer's understanding of how tasks should be executed.
2. **Robustness** — Event-driven applications tend to be more robust since they are less sensitive to the order in which users perform activities. In conventional programming, the programmer has to anticipate virtually every sequence of activities the user might perform and define responses to these sequences.

The primary disadvantage of event-driven programs is that it is often difficult to find the source of errors when they do occur. This problem arises from the object-oriented nature of event-driven applications—since events are associated with a particular object you may have to examine a large number of objects before you discover the misbehaving procedure. This is especially true when events cascade (i.e., an event for one object triggers an event for a different object, and so on).

13.5 Application to the assignment

- Add “update status” check boxes to you transaction processing forms (i.e., `Orders` and `Shipments`)
- Create a conditional macro for your `Shipments` form to prevent a particular shipment from being added to inventory more than once.
- Create a main switchboard for you application. It should provide links to all the database objects your user is expected to have access to (i.e., your forms).