Tutorial

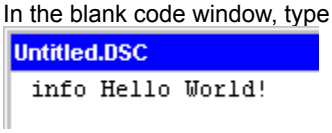
If you are new to Visual DialogScript, this short tutorial will introduce you to the main features of the system.

Lesson 1

Most books and tutorials on programming begin with a program to display the message "Hello World!" This one will be no different ...

Visual DialogScript programs are written using the DialogScript language. To start a program, choose File / New / Script Program from the menu, or press the New Script  toolbar button.

In the blank code window, type the following line:



info is a DialogScript command. You can get online help on it (and all commands) by putting the cursor in the word, right-clicking, and choosing Help at Cursor from the context menu, or pressing **Ctrl+F1**.

The **info** command displays a small dialog box with an information symbol and the text you supply. A few things are worth pointing out here. First, DialogScript isn't generally very fussy about the case of commands, so if you wrote INFO, info or Info, the result would be the same. The text, of course, is displayed exactly how you wrote it.

The general format of commands is:

COMMAND parameters

where "parameters" is information you supply. Some commands have no parameters, some have only one, and others may take more than one parameter. In all cases, a space separates the command from the first parameter, and commas separate parameters from each other.

Before running your new script, it's a good idea to save it. Press the Save button  and choose a name, such as "Hello."

To run the script, you could use either the Trace button  or press **Shift+F9**. or the Run button  (**F9**).
The Run button executes the script normally. The Trace button shows you how the script is running by highlighting each line of the program as it is executed. This makes execution run more slowly than when Run is used.

You can stop the script at any point by pressing the Stop button  (**Ctrl+F2**). From there you can step through it a line at a time using the Single Step button  (keyboard equivalent **F8**) or continue execution using the Continue button  (**Ctrl+F9**)..

Run the script, and you will see the message displayed:

You can specify your own title for dialogs displayed by the script, by using the **title** command. Insert a line, so that the script now looks like this:

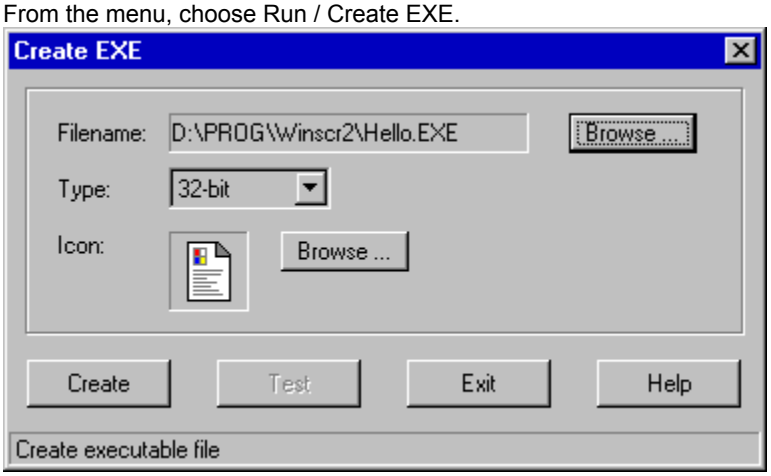
C:\PROG\Scripts\Hello.dsc

```
title HelloApp
info Hello World!
```

and run it again. You will now see your own title displayed in the dialog title bar.

Having written and tested your first DialogScript program, you can now create an executable (.EXE) version which can be run standalone, given its own shortcuts and so on.

From the menu, choose Run / Create EXE.



From this dialog, you could change the name and/or location of the compiled EXE file by pressing the first Browse... button. You could specify that you want to create a 16-bit program for Windows 3.1 by changing the option in the drop-down list. You could also choose an icon for the program by pressing the second Browse... button alongside the icon picture.

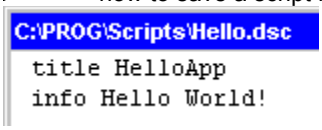
For now, just press OK to save the script to an EXE file.

You can now test the compiled program by pressing.the Test button. (For this to work, the runtime engine DSRUN.EXE or DSRUN16.EXE must be on the search path when you execute the program.)

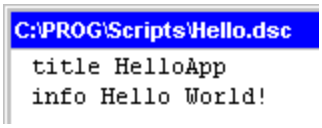
That concludes Lesson 1.

In this lesson, you have learnt:

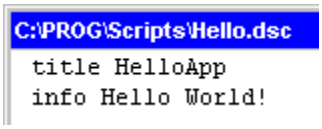
- how to start a new script;
- how to call up command help;
- how to save a script file;



how to test a script using the Run and Debug buttons;



how to set the title for a script application;



how to make an executable version of a script file.

Return to Contents page

Lesson 2

In this lesson you will discover the use of variables, functions and how to use the debugger.

Open a new script file, and enter the following lines of code:

C:\PROG\Scripts\lesson 2.dsc

```
title Counter
%C = 0
repeat
  %I = @input("Enter a name (blank to finish):","")
  %C = @succ(%C)
until @null(%I)
info You entered %C items
```

To get a description of what the commands do, put the cursor in the command (the first word on the line) and press **Ctrl+F1**. You can also do this for functions (which begin with an @ sign.)
As well as commands, this script contains assignments, for example %C = 0. The % indicates that C is a variable. When a variable appears on the left of an equals sign it is having a value assigned to it. When one appears to the right of an equals or a command the variable is replaced by its contents before the assignment takes place or the command is executed.

Functions are similar to commands except that they return a value. Like variables, functions are evaluated and replaced by the result before the command or assignment takes place.

Let's look at the code in this example briefly.

```
title Counter
```

This command gives your program a title, as we saw in Lesson 1. When run as an EXE program, the title is the name that appears on the Windows 95 Task Bar.

```
%C = 0
```

In this line, the string '0' is assigned to variable C. Note: *string* **not** *integer value*.

```
repeat
```

This is the start of a loop which must be paired with an **until** command further down.

```
  %I = @input("Enter a name (blank to finish):","")
```

In this line, the result of the **@input** function is assigned to variable I. As the online help will confirm, the effect of @input is to display a small dialog box to invite input from the user, with a prompt string as the first argument and an optional default value as the second.

Worthy of note is the use of quotes round the strings in the function. It is not usually necessary to put quotes round strings in DialogScript. The exceptions are if the string contains characters like %, @, brackets or commas, which you don't want treated as variables, functions, parameter separators and so on. In this case, there are brackets in the first string, so the quotes are needed.

Also worth noting is the fact that this line, and all those within the **repeat** .. **until** pair, are indented by a couple of spaces. This is a notational convenience which makes the script easier to read, but makes no difference to its actual execution.

```
  %C = @succ(%C)
```

Here, the value of C is replaced by the result of @succ(%C). **@succ** expects one argument, in the form of a number, and returns the next number: its successor. In mathematical terms, this could be written as %C = %C + 1. Note, however, that if you *had* written the expression in that way, %C would then contain the string "0 + 1". DialogScript does not understand mathematics. The only way to perform calculations in scripts is to use the mathematical functions, which treat strings as numbers and return numbers as strings.

```
until @null(%I)
```

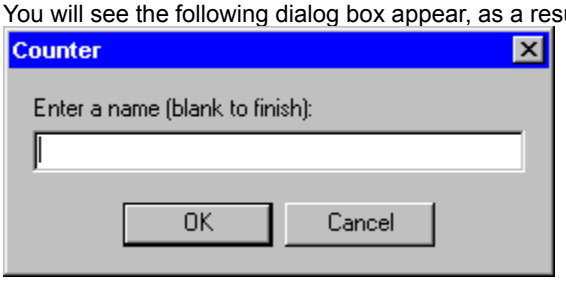
This is the terminating command of the **repeat** loop above. It says, stop looping when the value assigned to variable I above is an empty string.

```
info You entered %C items
```

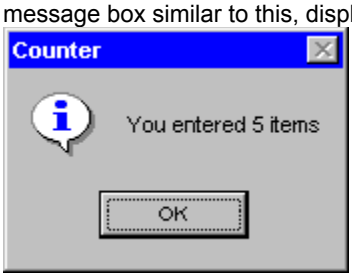
Finally, we display a message box showing the value of variable C. Because C is already a string, there is no need to use a special conversion function to print it, as would be needed by most programming languages.

Save this script, then run it.

You will see the following dialog box appear, as a result of the **@input** function:

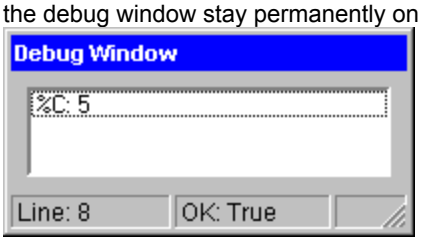


Enter a few names, then just press OK with a blank entry, to terminate the repeat .. until loop. You will then see a message box similar to this, displayed by the **info** command on the last line:



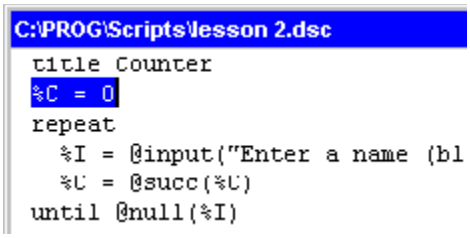
You may have noticed that the count displayed was one greater than the number of names you entered. The reason is probably obvious to you already, but for the purpose of illustrating how to use the debugger we will pretend it isn't.

First, we will open the debug window, by pressing the  button or pressing **F6.** Size and position the debug window so you can see it and the code window at the same time. From the Options menu you can choose to have the debug window stay permanently on top if you wish.



The debug window lists all the variables that have a current value assigned to them. It also shows the line number of the command currently being executed, and the status of the system variable OK, which is set to False by certain commands in certain conditions, and can be tested within the script using the @OK() function.

Now press the Single Step button ▶ or press **F8**.
Watch the code window. You will see a bar move down the window, highlighting the line that is about to be executed.



When the bar moves past a line in which a value is assigned to a variable, you will see the variable, and the value it now contains, listed in the debug window.
The ability to view the contents of variables at any stage during the execution of a script is a valuable aid to debugging scripts which don't work as you want them to.

In this example, it would show you that %C is being incremented even when you don't enter a name (to exit) and so the count is one greater than it should be.

Single stepping through a long script can be tedious, particularly if there are loops in it. Therefore, DialogScript also offers *breakpoints*. You can set a breakpoint at any point in a script, run it in the debugger, and the script execution will stop when it gets to that point. If the debug window is open it will show you the current contents of all the variables at that point. You can continue from there to the next breakpoint, or single step over a critical section of code.

To set a breakpoint, put the cursor on the line you want execution to stop at, right click the mouse, and then select Add Breakpoint at Cursor, or press **Alt+B**. You will see a line saying BREAK appear in the script.

```
repeat
  %I = @input("Enter a name
  %C = @succ(%C)
until @null(%I)
BREAK
info You entered %C items
```

Run the script and you will see that the execution halts at the line following the BREAK. An error message dialog will appear, giving you the line number and the message "Breakpoint reached."

To continue after the breakpoint, press the Continue button  or **Ctrl+F9**. Alternatively you could use single step, ▶ (**F8**.)
To clear the breakpoint, put the cursor on the word BREAK and choose Remove Breakpoint at Cursor from the context menu. **Alt+B** removes a breakpoint if the cursor is currently on a breakpoint. You can use **Alt+C**, or Clear All Breakpoints, to remove all the breakpoints from a script in one go.
Note: you should not insert or remove breakpoints by typing or deleting the word BREAK, if you intend resuming execution of the script. Adding or removing those lines affects the line number of lines below the breakpoint in the script. If you add or remove breakpoints through the menu or keyboard shortcuts then the current line number is automatically adjusted so that the script will resume at the correct point.

Note also that a script may not work correctly if you make any changes to it and then continue execution from the point at which it is currently stopped. You should always run the script from the start, using Run ▶ or Trace ▶ after making any changes. If you want to single step the script from the start, choose Run / Reset from the menu, to reset the variables to null and the line counter to the beginning, before stepping through the script.

This concludes Lesson 2.

In this lesson, you learnt:

- how variables are assigned;
- what functions are;
- how DialogScript treats strings and numbers;
- how to set and remove breakpoints;
- how to trace the execution of a script by single stepping;
- how to view the contents of variables using the Debug Window;
- that you should always restart a script from the beginning after making changes to it.

Return to Contents page

Lesson 3

In this lesson we will create a simple dialog based program. We will use the Dialog Wizard to generate the skeleton code for the program, and then add code to make it do something useful.

To begin creating the program, choose File / New / Dialog Wizard from the main menu.

When the Wizard appears, click Next to move to the second page. In the box that says "Enter a title for your DialogScript application" enter Alarm Clock.

The best place to put the TIMER event code is before the first **wait event**. That way, the time display is updated immediately after the dialog is created, instead of waiting 10 seconds with the display showing 00:00 before the first update.

You need to add seven lines, including the label, before the label **evloop.**

```
DIALOG CREATE,Alarm Clock,-1,0,160,60,STYLE(CL(
  TEXT(TIME;10;10;90;32;00:00;CLOCK),BUTTON(Set
:TIMER
  %T = @datetime(hh:nn)
  dialog set,time,%T
  if @equal(%T,%A)
    warn Alarm at %T
    %A =
  end
:evloop
  wait event,10
```

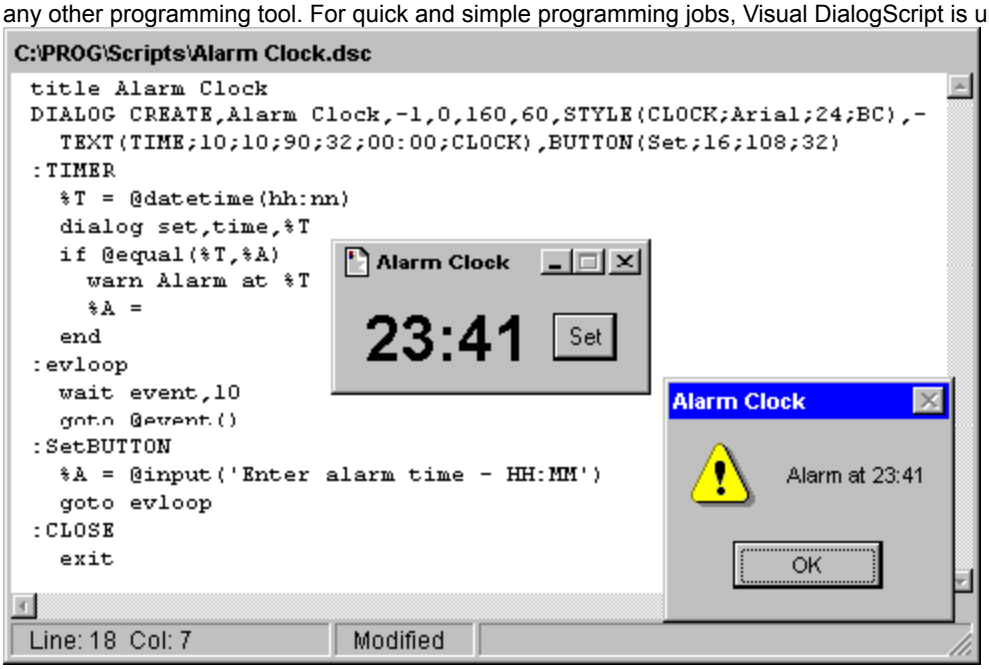
Add

The first line after the label TIMER gets the time in the right format into variable %T. Note that nn, not mm, is used to get the minutes: mm would return the month number.

The second line puts the value of %T into the text element on the dialog.

The third line tests whether the time %T is equal to the alarm time (if any) %A. If they are equal, a warning message is displayed. %A is set to null after the warning dialog has been closed, to prevent the warning from appearing again.

That completes the program. You wrote just eight lines of code to create a working alarm clock utility. You will find that Visual DialogScript allows you to create other simple utilities in far less time and using far fewer lines of code than any other programming tool. For quick and simple programming jobs, Visual DialogScript is unbeatable.



That concludes Lesson 3.

In this lesson, you learned:

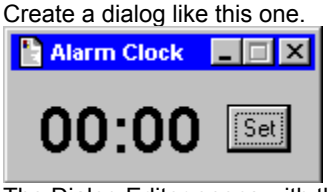
- how to use the Dialog Wizard to create a skeleton dialog based program;
- how to use styles to change the text attributes on a dialog;
- how DialogScript programs respond to user interactions;
- how to add code to respond to button press, dialog close and timer events.

Return to Contents page.

Press Next again.

Click the button to run the Dialog Editor.

Create a dialog like this one.

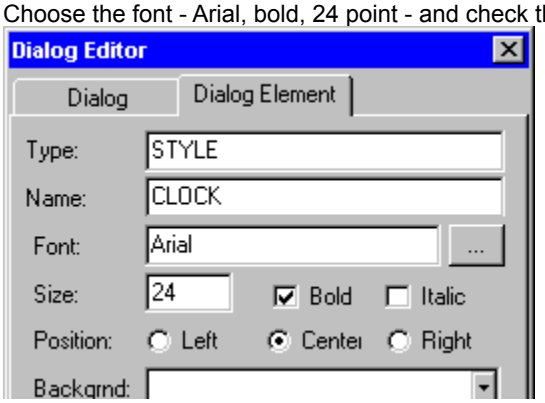


The Dialog Editor opens with the Dialog page showing. This page gives the properties for the dialog itself, and lists the dialog elements that specify other information about how the dialog is to look.
The first line is the caption of the dialog. It should read Alarm Clock.
The second line gives the position of the top left corner of the dialog, and its width and height. A value of -1 for the top means that the dialog will appear in the center of the screen. The width should be 160 and the height 60.

Now to add the text (for the time display) and the button to the dialog.

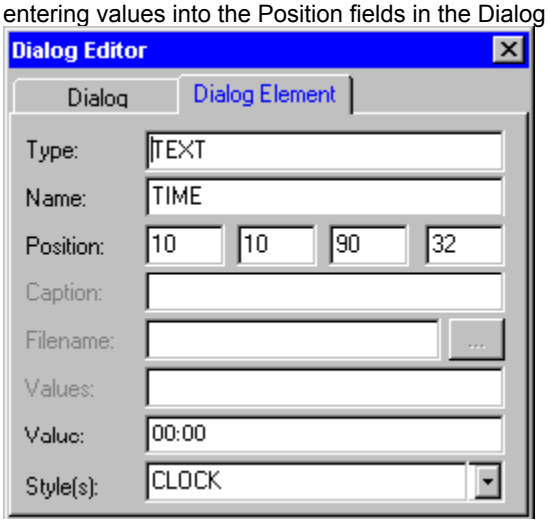
First, we need to define a style for the text element. Styles are used to specify things like font and color, if these depart from the Windows default. We want to display the time in Arial, bold, 24 point, and have it centered in the text element.

To insert an element you press **Ins,** or right click on the Dialog Elements list and choose Add from the popup menu. When the Add New Element dialog appears choose STYLE from the drop down list, and give the style a name such as CLOCK. Press OK. You will now be on the Dialog Element page with a set of fields that relate to defining a style. Choose the font - Arial, bold, 24 point - and check the radio button to have the text justification centered.



Next we shall add the text element.

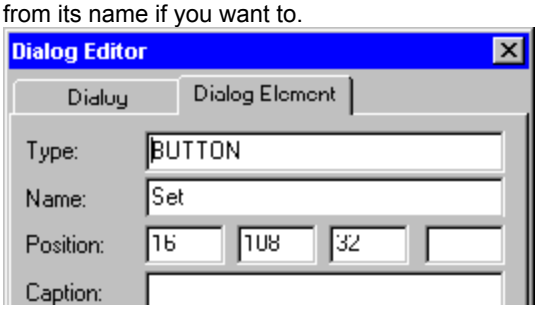
Go back to the Dialog page (click the tab or press OK once) and press **Ins** again. This time select TEXT from the drop down list. Name the element something like TIME. Size the element so that it is about 90 wide and 32 high. You can do this either by dragging the handles that appear on the element on the dialog itself, or more precisely by entering values into the Position fields in the Dialog Editor.



Change the Value of the text element to say 00:00. From the drop down list, select the CLOCK style you just defined.

The style does not immediately take effect on the displayed dialog. To see the effect of the large bold typeface you defined, change back to the Dialog tab and key **Ctrl+R**, or select Refresh from the popup menu.

Finally we add the Set button. Press Ins again, choose BUTTON from the drop down list, and give the button the name Set. By default, the name of a button appears as its caption, though you can give a button a different caption from its name if you want to.

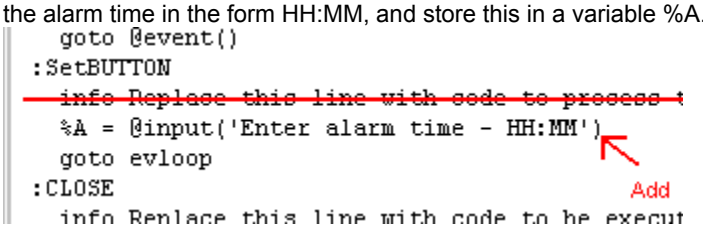


You should change the default width of the button to 32, then drag it into position to the right of the time display.

Press OK, then OK again to return to the Dialog Wizard. Press Next, and the instructions from the Wizard tell you to press Finish to generate the code. The code will appear in the Code Window. You should save the code. Give it the name Alarm Clock.

You can now run the code, by pressing Trace ▶ or Run ▶ . Note that when you press the Set button, or the Close button at the top right of the title bar, a message box appears reminding you that you have got to write code to respond to the button press.

First, let's write the code to respond to the Set button. When this button is pressed, we want to ask the user to enter the alarm time in the form HH:MM, and store this in a variable %A.



The line

```
goto @event()
```

causes the program to go to a label with the name of an event that has occurred. Events occur when the user interacts with the dialog, by (for example) pressing a button.

When the user presses the Set button a SETBUTTON event occurs, and the program goes to the label SetBUTTON. (Note: labels are not case sensitive.)

Delete the **info** message line following the label that was generated by the Wizard, and insert a line that says:

```
%A = @input('Enter alarm time - HH:MM')
```

Save the program, and then run it to test what happens when you press the Set button.

For the alarm clock, we don't require any special processing to take place when the user closes the dialog, so you can simply delete the info message generated by the Wizard on the line that follows the label CLOSE.

```
  goto evloop
:CLOSE
  info Replace this line with code to be executed bef
  exit
```

Now we have to update the clock, check whether the alarm time has been reached, and display a message if it has.

To update information periodically we need to generate a TIMER event. To do this, we specify an interval, in seconds, at the WAIT command. By changing the line:

```
wait event
```

to

```
wait event,10
```

we will generate a TIMER event at 10 second intervals, in addition to the events that will occur when the user presses a button.

This means that every 10 seconds, the

```
goto @event()
```

command will cause the program to go to a label called TIMER. There is no such label at the moment, and if you try to run the progarm it will wait 10 seconds and then fail with an error: Label not found.