

ESPL Manual

Ensign Software Programming Language

Ensign 10

Copyright © 2019 Ensign Software, Inc.

Last Update: 3 April 2019

Table of Contents

Introduction	4
ESPL Language Features	4
Documentation Format	5
ESPL Programming Window	6
Projects	7
Suggestions for Designing an ESPL Program	7
Creating a new Project	7
Opening an Existing Project	8
Editing your ESPL Program	8
Designing Forms	8
Tool Palette and Components	9
Object Inspector	10
Adding some Programming Code	11
Running a Program	12
Saving the project	12
Adding more Features	13
Changing Component Properties	13
Writing Code - Events and Event Handlers	15
Code completion	16
Debugging scripts	17
ESPL Programming	18
Variable Types	18
Colors	19
Constants	20
Playback	20
Program Structure	20
Variable, Function, and Procedure Names	21
Assign Statements	21
Strings	21
Comments	21
Variables	22
Indexes	22
Arrays	22
Case statements	23
Function and Procedure declaration	23
Calling a subroutine	24
Passing parameters	24
Accessing objects	25
Calling DLL functions	27
Supported Types	28
Include Libraries	28
Secure Library Files	29
Declaring Forms in ESPL	29

[Event Redirection](#).....31

ESPL Manual

Introduction

The Ensign Software Programming Language (ESPL) allows traders to create custom chart studies, lines, reports, and tools. It can also be used to develop trading systems, alerts, custom forms, and scans. The language contains hundreds of programming statements, functions, events, methods, and properties. This manual documents each programming statement and provides many examples.

The language is nearly identical to Delphi Pascal programming. Users who are familiar with Delphi will easily adapt to the ESPL language. The language includes hundreds of customized commands that provide unparalleled power and control over nearly every aspect of Ensign.

Note: This new ESPL language for Ensign 10 is not backwards compatible with old ESPL programs written for Ensign Windows. Modifications to old ESPL programs written for Ensign Windows will be necessary in order to run with this new version of ESPL for Ensign 10.

ESPL Language Features

- begin .. end blocks
- procedure and function declarations
- if .. then .. else
- for .. to .. do .. step
- while .. do
- repeat .. until
- try .. except and try .. finally blocks
- case statements
- array constructors (x:=[1, 2, 3];)
- ^ , * , / , and , + , - , or , <> , >= , <= , = , > , < , div , mod , xor , shl , shr operators
- access to object properties and methods (ObjectName.SubObject.Property)
- Integrated Development Environment (IDE)
- Debugging tools: breakpoints, single step, variable watch window

Documentation Format

The following conventions are used throughout the documentation to define syntax.

<u>Convention</u>	<u>Description</u>
Boldface	Programming functions and statements.
()	Parentheses enclose parameters that are necessary for each function or statement. Commonly required parameters include numbers, price values, colors, types, bar index locations, etc.
<i>Italics</i>	Parameters and Variables. The functionality of each parameter is documented so that you will know what the parameter specifies and is used for. Parameter values must match the indicated variable type (ex. integer, real, string). For example, an integer value should not be entered as a parameter if a string value is expected.
[]	Optional parameters are enclosed in square brackets. Optional parameters provide additional functionality to programming statements, but can be omitted if not necessary.
{ }	Curly brackets enclose comments. Comments help document programming code and are ignored when a program runs.
<code>Courier</code>	Example ESPL programs are shown using the <code>Courier New Font</code> type. If desired, sample programs can be entered and run in the Ensign ESPL Editor window.

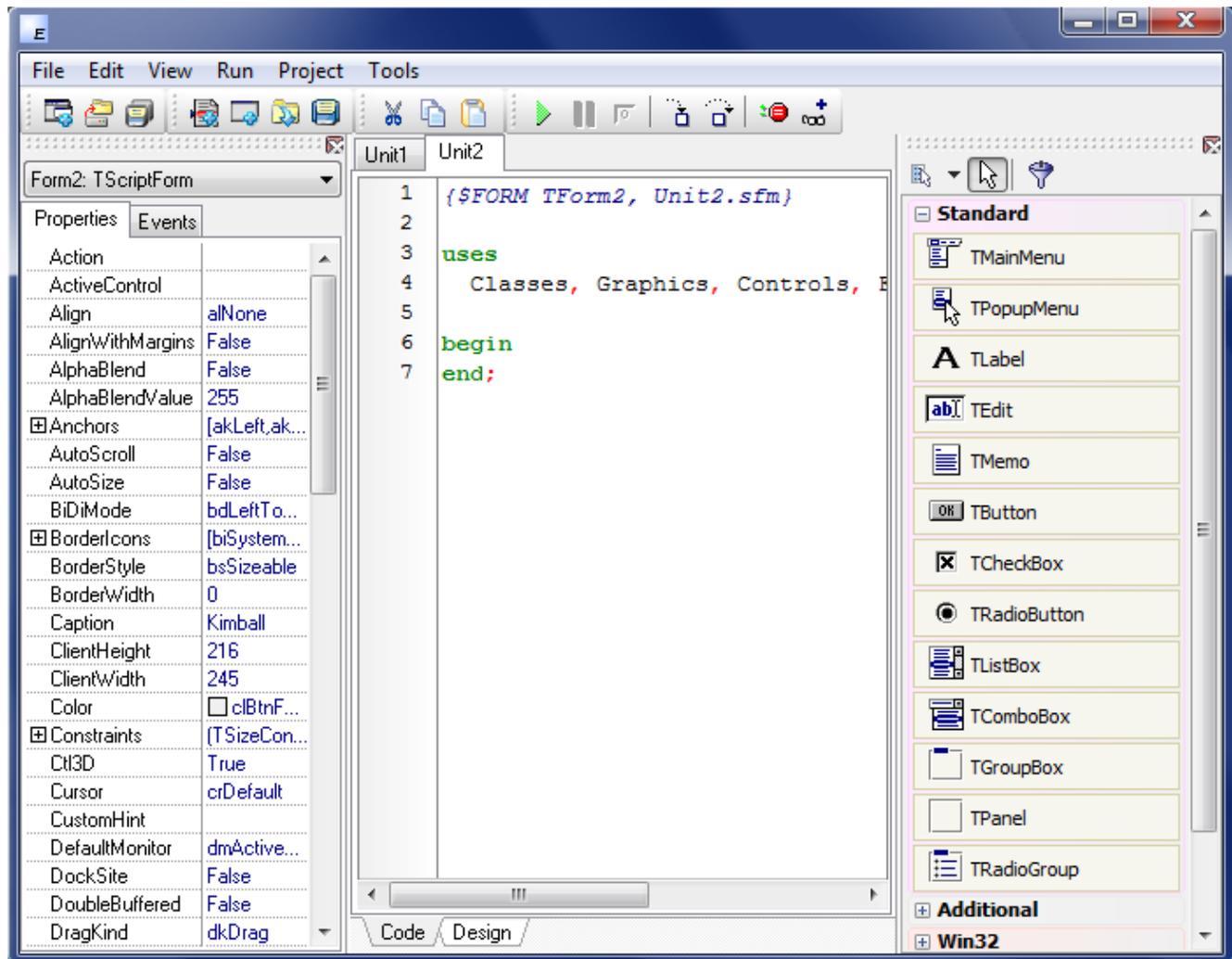
Each documented function and statement uses the following format.

<u>Heading</u>	<u>Description</u>
SYNTAX	Shows the required syntax for the function or statement
DESCRIPTION	Describes what the function or statement does.
PARAMETERS	Describes parameters, variables, and type names that are necessary for a function or statement.
EXAMPLE	Shows an example program using the function or statement.
NOTE	Calls attention to important information regarding the use of the function or statement.



ESPL Programming Window

ESPL applications are programmed and developed using the ESPL programming window which is an Integrated Development Environment. The ESPL programming window is used to create and design forms and to program applications that accomplish your customized needs. To view the ESPL programming window click the ESPL button on the Main toolbar.



The ESPL programming window has an Object Inspector on the left, a Code Editor window in the middle, and a Tool Palette on the right. A Toolbar and several Menu items are displayed at the top of the window. Keyboard shortcuts are available for most of the menu items.

The Object Inspector is used to view and edit the Properties and Events for objects and forms. The Code Editor window is used to type and view your programming code. The Tool Palette allows you to select and utilize many different useful components for your forms.

Projects

A Project is a collection of ESPL programming files that can be compiled and run to perform your customized programming tasks. The ESPL programming language can be used to create projects that will plot custom chart studies, plot unique chart lines, create complex reports, scan symbols, and test your private trading systems.

Suggestions for Designing an ESPL Program

A programmer will often design a new program mentally, or on paper, before actually writing any programming code. The first step in developing an ESPL program is to decide what the program should do and what the user should see when the program runs.

Will the program utilize a form?

Will the form need menus, buttons, edit boxes, or other components?

Will the program plot some lines on a chart?

Will the program access some chart data?

Will the program need to display some calculation results?

Answering these types of questions can help you decide if an additional form is necessary in your project. If a form is required, then you will need to select and decide where to place the components that you would like to use (example: buttons, edit boxes, labels, combo boxes, option buttons, etc.). After you design the form and interface for the program it will be easier to start writing code, and you can decide what Events the components on the form should recognize. For example, what should happen when the user clicks a particular button.

Creating a new Project

To create a new project, select **File | New Project** from the menu, or click the **New Project** button on the toolbar. When a new project is started, a ProjectFile is created that keeps track of all the other files and settings for the project. This file must have a file extension of **.ssproj**. You cannot view or edit the ProjectFile in the ESPL programming window. However, when you initially save your project you will be asked to name the ProjectFile. You should name the ProjectFile to be the logical name of your application (like **MyStudies.ssproj**, or **MyScan.ssproj**, or **TradingSystem5.ssproj**, etc.). We suggest that you create a new folder and save the ProjectFile and its associated files in that folder.

Example: `\Ensign10\ESPL\MyScan\`

Additional files associated with the project are script files (called Units), and Form files. Unit files have a file extension of **.psc**. Forms are comprised of 1 Unit file and 1 Form file. Form files are automatically created and saved when its associated Unit file is saved. Form files have a file extension of **.sfm**.

When you begin a new project, ESPL will create by-default a main unit (Unit1) and a form unit (Unit2). These two files comprise the initial project. If you run the new ESPL application right away (by clicking the green run button or pressing the **F9** function key), a blank form window

will appear on your computer screen. If you don't need a form in your program, then you can remove Unit2 and its associated Form from the default project.

Main Unit

Each project has a main unit (initially named Unit1). The designated main unit is the script file that will be executed when you press F9 or click the Run button. If necessary, you can change which unit is the designated main unit by selecting Project | Select Main Unit from the menu, and then selecting a unit from the list of included script files.

Creating/adding units/forms to the project

You can create or add existing units/forms to the project by choosing the "File | New unit", "File | New Form" and "File | Open (add to project)" menu options. If you are creating a new one, you will be prompted with the same dialog as above, to choose the language of the new unit. If you're adding an existing unit, then the IDE will detect the script language based on the file extension.

Opening an Existing Project

To Open and display an existing project in the ESPL programming window, select File | Open Project from the menu (or click the Open Project button on the toolbar), then browse to the project file on your hard disk and select it.

Editing your ESPL Program

Use the Code Editor window to edit and program your ESPL programs. Features of the Code Editor window include:

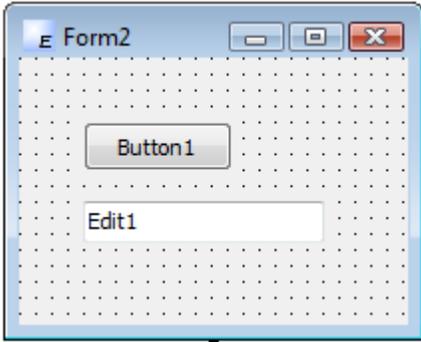
- code completion (pressing Ctrl+Space)
- syntax highlight
- line numbering
- clipboard operations
- automatic indentation

Designing Forms

A Form can be designed and used for many purposes. Forms are often used as the user interface to run and control a program. A Form can be used to collect input, open and read files, print reports, and to display lists and images. Forms can be designed with custom labels, edit boxes, menus, buttons, and check boxes. When designing a Form, remember that there are two files that control a Form (the Unit file and its associated Form file). Switch the Code Editor window view between the Unit programming code and the Form designer view by pressing the F12 key or by clicking the Code and Design tabs at the bottom of the Code Editor window.



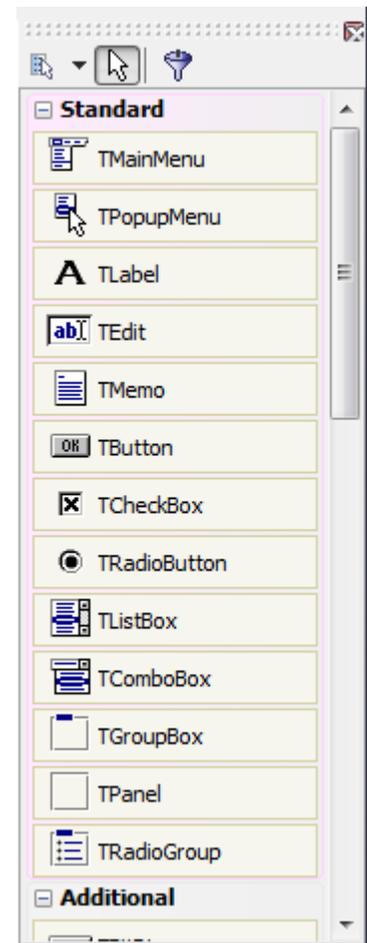
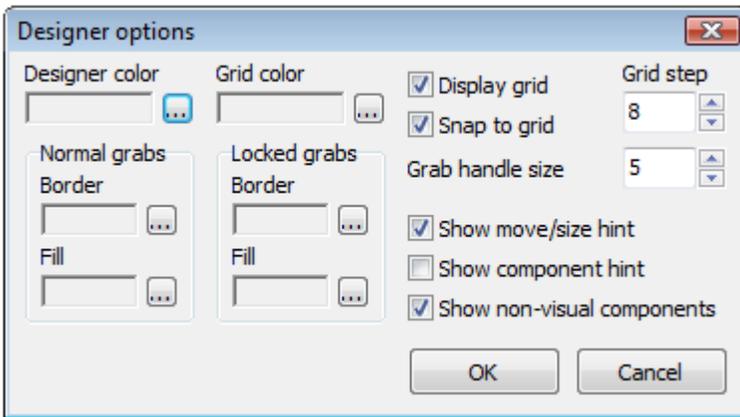
In the following simple example, a button and an edit box have been placed on a form.



Form Designer Features:

- Multi-selection of components
- Full use of all Clipboard operations
- Alignment palette (menu "Edit | Align")
- Bring to front / Send to back
- Tab order dialog
- Size dialog
- Locking/unlocking controls
- Grid and Snap to Grid

You can customize the Form Designer view settings by selecting Tools | Designer Options from the menu. The look and feel of the ESPL Programming window can be customized with your favorite colors, grid options, and hint settings.



Tool Palette and Components

The Tool Palette is used to create the user interface and functionality of a Form. The Tool Palette contains a collection of useful Components (also called tools or objects) that can be placed on a Form. Components can be used to display information or allow the user to perform an action. For example a Label is used to display text, an Edit box allows the user to input some data, a Button can be used to initiate actions. Any combination of components can be placed on a form. When your program is running a user can interact with the components on the form.

To place a component on a Form, first select (click) the component on the Tool Palette that you want to use, then click the mouse on the Form. The component will be located at the position where the mouse was clicked, with a default width and height. Components can be moved and resized on a Form by dragging the component

with the mouse, or by changing the appropriate properties in the Object Inspector (like Height, Width, Top, and Left). Components can also be moved and resized by using the following keyboard keys:

CTRL+Up Arrow: Move the component upwards on the form.
CTRL+Down Arrow: Move the component downwards on the form.
CTRL+Left Arrow: Move the component to the left on the form.
CTRL+Right Arrow: Move the component to the right on the form.

SHIFT+Up Arrow: Decrease the height of the component.
SHIFT+Down Arrow: Increase the height of the component.
SHIFT+Left Arrow: Decrease the width of the component.
SHIFT+Right Arrow: Increase the width of the component.

To remove a component from a form, first click on it and then press the Delete key on the keyboard.

Each component has specific Properties and Events that allow you to control your program at design time and at run time. Several components are available for use on the tool palette. They are grouped according to the function they perform (Standard, Additional, Win32, and Dialogs). Each group displays icons representing the components you can use to design your application interface. For example, the Standard group includes controls such as the edit box, label, button, and listbox.

Each time you start a new project you begin with an empty form window. The default form name is Form2 (in Unit2). This form can be renamed, resized and moved. It has a caption and the three standard minimize, maximize and close buttons (at the top right corner of the form).



When a form is the active window and you press the F12 key, the Code Editor window will be displayed that contains the code for the form. Press F12 again to revert back to the form view (or click the Design tab at the bottom). As you add components to a form and design the user interface of your application, ESPL automatically generates some underlying code for the form and its components. The Properties (settings) and Events of each component can be changed by using the Object Inspector window. It is your task, as the programmer, to decide what happens when a user clicks a button or changes the text in an Edit box, etc.

Object Inspector

Each form and its components have Properties and Events. Properties such as color, size, position, caption can be modified and customized to your exact needs. Events can also be enabled or monitored, such as a mouse click, key press, or component activation. The Object Inspector (left side of the ESPL Window) displays the properties and events (note the two tabs at the top of the Object Inspector) for the selected component and allows you to change the property value or select the response to an event.

For example, each form has a Caption property (the text that appears on the form's title bar). To change the Caption of Form2 first activate the form by clicking on it. Find the Caption property in the Object Inspector and see that it has a current value of 'Form2'. Change the Caption text by simply typing some new text (like MyForm). When you press ENTER the Caption of the form will change to MyForm.

Use the Object Inspector to change any of the Properties for the form or a component. For example, the screen position of a form can be specified by editing the Left and Top property values.

Properties	Events
Action	
ActiveControl	
Align	alNone
AlignWithMargins	False
AlphaBlend	False
AlphaBlendValue	255
⊞ Anchors	[akLeft,akTop]
AutoScroll	False
AutoSize	False
BiDiMode	bdLeftToRight
⊞ BorderIcons	[biSystemMenu,biMini]
BorderStyle	bsSizeable
BorderWidth	0
Caption	Form2
ClientHeight	206
ClientWidth	312

Adding some Programming Code

The Code Editor window is used to enter your programming code. For example, instead of using the Object Inspector to change the Caption of a form, you could add some programming code to change the caption.

To add programming code that changes the Caption of the form when the program runs, do the following. First, double-click the mouse on Form2 to display the Code Editor window that contains the programming code for the form. Note: You can also double-click the OnCreate Event for Form2 in the Object Inspector.

The form has a collection of events such as a mouse click, key press, or component activation for which you can specify some additional behavior. In this example, the Event is called OnCreate. This event occurs when the form is created (when the program runs).

Add the code on line 8. The code will be executed when the program runs and the form is first created. The code changes the form caption to display 'Hello Friend!' plus the date and time.

```

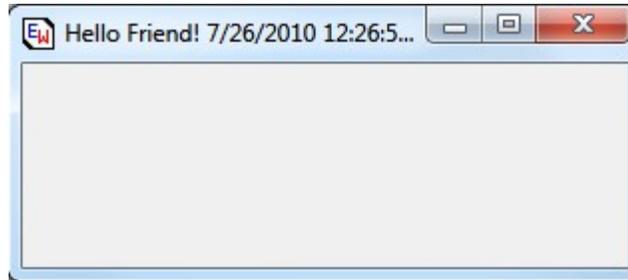
1  {$FORM TForm2, Unit2.sfm}
2
3  uses
4    Classes, Graphics, Controls, Forms, Dialogs;
5
6  procedure Form2Create(Sender: TObject);
7  begin
8    Caption := 'Hello Friend! ' + DateTimeToStr(Now);
9  end;
10
11 begin
12 end;

```

Running a Program

To see the results of your programming, compile and run the program (which is comprised of all the files in the project). To run the program click the green Run button on the toolbar, or choose Run (from the Run menu), or press the F9 key on the keyboard. The ESPL compiler will build the project and run the program (application). If the compiler ever detects an error in the programming code it will display an Error window. In the case of an error, you would click OK and the Code Editor would place the cursor on the line of code containing the error.

If the program compiles with no errors, then the program will run and you will see a blank form on the screen. Every time you run this program the form caption will display Hello Friend! along with the date and time that the program was run.



There is not much you can do with the form window in this simple example. You can move it, resize it, or click the X button to close it.

Saving the project

After you have started a new programming project, you will want to save the project to the hard disk. This will allow you to reopen and run the programming code later (after shutting down and rerunning the Ensign program). Periodically save your project, even during development, so that you always have a backup copy.

To save a project and all of its associated files select File | Save All from the menu, or click the Save All button on the toolbar. By default, ESPL projects are saved to the \Ensign10\ESPL folder. We suggest that you create a new folder (inside the ESPL folder) for each separate project. This will help you to stay organized and avoid mixing files from different projects. When you save a project for the first time you will be asked to name and save each of the Unit files, and also to name and save the Project file (this should be the name of your program). For example,

save Unit2 as	Unit2.psc (the form will autosave as Unit2.sfm)
save Unit1 as	Main.psc
save Project1 as	Hello.dpr

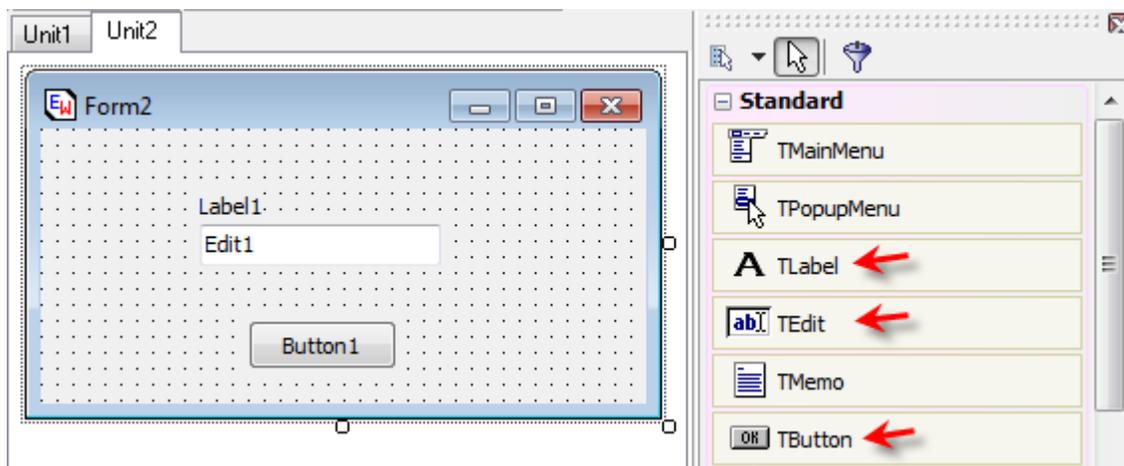
Note: In the Code Editor window, the Unit1 tab will be changed to Main.

Adding more Features

Lets build on the previous example and add some components to the form. First, click on the form and then move the mouse to the Tool palette and select the 'Standard' group. We will add three standard Windows components and write some example code to see how the components work together.

Add these three components:

- TLabel : use this component when you want to add some text to a form that the user can't edit.
- TEdit : standard Windows edit control. Edit controls are used to retrieve text that users type.
- TButton : use this component to put a standard push button on a form.



For example, select the TLabel entry in the Tool palette, then click the mouse on the form. A label should appear on the form. Next, select the TEdit tool, then click on the form again. Then, select the TButton tool, then click on the form. If necessary, you can drag the components around on the form to locate them as shown below.

Changing Component Properties

After you place components on a form, you can set their properties with the Object Inspector. The properties are different for each type of component, some properties apply to most components. Altering a component property, changes the way a component behaves and appears in an application.

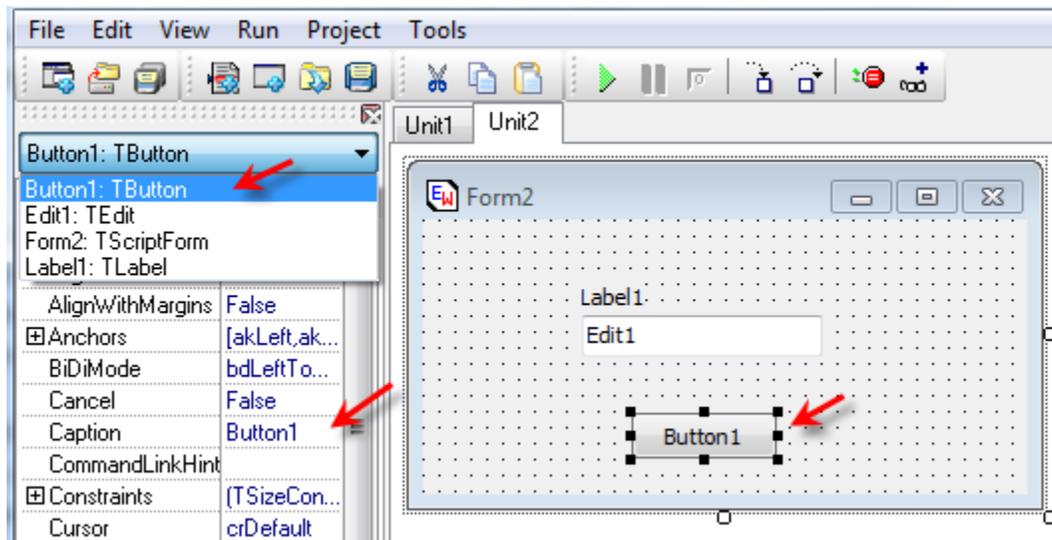
All the components have a property called Name. The Name property specifies the name of the component as referenced in the programming code. When you first place a component on a form, ESPL will provide a default name for the component, such as Label1, Edit1, Button1. A good programmer will usually change the names to be more descriptive and meaningful. For example, a form might have 3 buttons. Instead of using the default names of Button1,

Button2, and Button3, you might change the names to be btnStart, btnPause, and btnExit (assuming that these were actions that the buttons would perform). Give the components a meaningful name before writing further code that refers to them. This is done by changing the value of the Name property in the Object Inspector.

To change a component property you must first activate the component. When you click on a component (to activate it) small square handles appear at each corner and in the middle of each side. Another way to select a component is to click its name in the drop down list that appears at the top of the Object Inspector. The list shows all the components on the active form along with their type and name.

When a component is selected, its properties (and events) are displayed in the Object Inspector. To change the component property click on a property name in the Object Inspector; then either type a new value or select from the drop-down list.

For example, change the Caption property for Button1 (I'll refer to components by their names) to 'Hello...' (of course without the single quotation marks)



Components have different kinds of properties; some can store a Boolean value (True or False), like Enabled. To change a Boolean property double click the property value to toggle between the states. Some properties can hold a number (ie. Width or Left), a string (ie. Caption or Text) or even a set of "simple valued" properties. When a property has an associated editor, to set complex values, an ellipsis button appears near the property name. For example if you click the ellipsis of the Font property a Font property dialog box will appear.

Now, change the Caption (the static text the label displays on the form) of Label1 to 'Your name please:'. Change the Text property (text displayed in the edit box - this text will be changeable at run time) of Edit1 to your name.

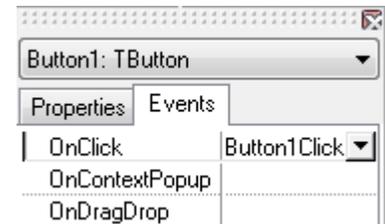
Writing Code - Events and Event Handlers

To really enable components to do something meaningful you have to write some action-specific code for each component you want to react on user input. Remember: components are building block of any ESPL form, the code behind each component ensures a component will react on an action.

Each component, beside its properties, has a set of events. Windows as event-led environment requires the programmer to decide how a program will (if it will) react on user actions. You need to understand that Windows is a message-based operating system. System messages are handled by a message handler that translates the message to ESPL event handlers. For instance, when a user clicks a button on a form, Windows sends a message to the application and the application reacts to this new event. If the OnClick event for a button is specified it gets executed.

The code to respond to events is contained in event procedures (event handlers). All components have a set of events that they can react on. For example, all clickable components have an OnClick event that gets fired if a user clicks a component with a mouse. All such components have an event for getting and loosing the focus, too. However if you do not specify the code for OnEnter and OnExit (OnEnter - got focus; OnExit - lost focus) the event will be ignored by your application.

To see a list of events a component can react on, select a component and in the Object Inspector activate the Events tab. To really create an event handling procedure, decide on what event you want your component to react, and double click the event name.



For example, select the Button1 component, and double click the OnClick event name. ESPL will bring the Code Editor to the top of the screen and the skeleton code for the OnClick event will be created.

```
Unit1 Unit2
1      {$FORM TForm2, Unit2.sfm}
2
3      uses
4          Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
5
6      procedure Button1Click(Sender: TObject);
7      begin
8
9      end;
10
```

Note: For the moment there is no need to understand what all the words in the above code stand for. Just follow along, we'll explain all that later.

As you will understand more clearly through this course, a procedure must have a unique name within the form. The above procedure, ESPL component event-driven procedure, is named for you. The name consists of: the name of the component name "Button1", and the event name "Click". For any component there is a set of events that you could create event handlers for. Just creating an event handler does not guarantee your application will do something on the event - you must write some event handling code in the body of the procedure.

We'll now write some code for the OnClick event handler of Button1. Alter the above procedure body to:

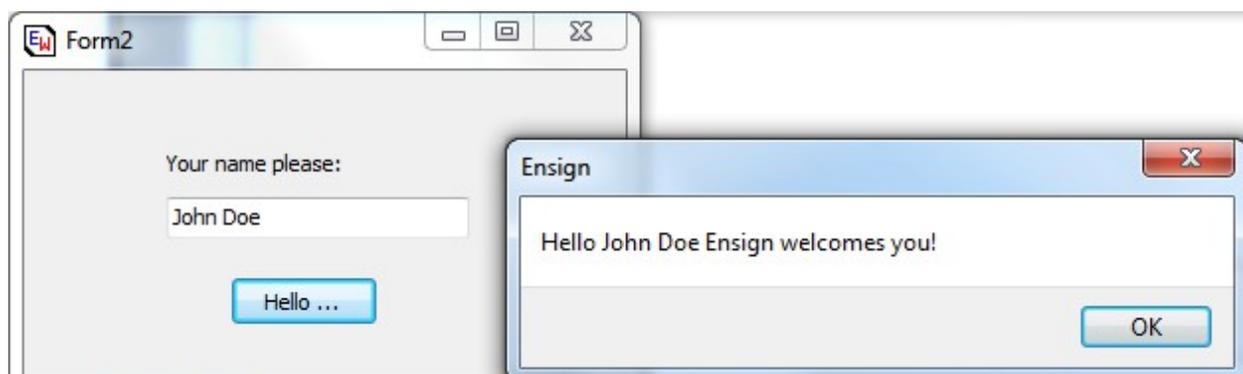
```
1  {$FORM TForm2, Unit2.sfm}
2
3  uses
4    Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
5
6  procedure Button1Click(Sender: TObject);
7  begin
8    s := 'Hello ' + Edit1.Text + ' Ensign welcomes you!';
9    ShowMessage(s);
10 end;
```

Code completion

When you reach to the second line and write "Edit1." wait a little. ESPL will display a list box with all the properties of the edit box you can pick. In general, it lists valid elements that you can select from and add to your code.



Now, hit F9 to compile and run your project. When the program starts, click the Button1 ('Hello...'). A message box will pop up saying 'Hello your name, Ensign welcomes you!'. Change the text in the Edit box and hit the Button again...



What follows is a simple explanation of the code that runs this application. Let's see.

- The first line under the **begin** keyword, `s := 'Hello ' + Edit1.Text + ' Ensign welcomes you!'`; sets a value for the variable `s`. This assignment involves reading a value of the Text property for the Edit component. The Text property of an Edit component holds the text string that is displayed in the edit box. That text is of the TCaption type, actually the string type.
- The last statement, before the **end** keyword, `ShowMessage(s);`, is the one that calls the message dialog and sends it the value of variable `s` - this results in a pop up box you see above.

That's it. Again, not too smart, not too hard but serves the purpose. By now you should know how to place components on a form, set their properties and even do a small ESPL application.

Here are some exercises for you:

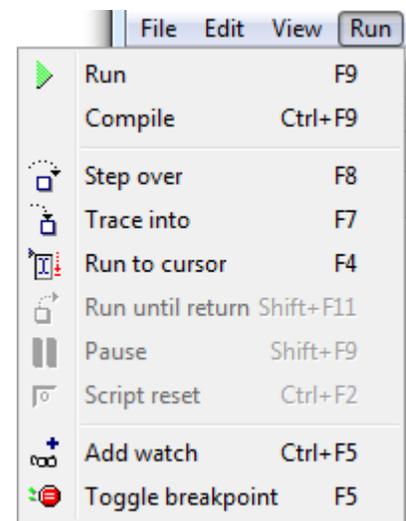
1. Play with the Color property of the Form object
2. Use the Font property Editor to alter the font of the TLabel component
3. Find out about the PasswordChar property of the TEdit component to create a simple password dialog form
4. Try adding code for the OnCreate event of a form to make a form appear centered on a screen. Also, become familiar with the Position property of the TForm object.

Debugging scripts

Use the IDE to run and debug scripts. The main features of the debugger are:

- Breakpoints
- Watches
- Step over/Trace into
- Run to cursor/Run until return
- Pause/Reset

The image shows the options under the menu item "Run". The shortcuts keys or the menu/toolbar buttons can be used to perform running/debugging actions, like run, pause, step over, trace into, etc.. You can also toggle a breakpoint on/off by clicking on the left gutter in the code editor. A watch can be added to inspect variable values.



ESPL Programming

Variable Types

Program statements and functions often require parameters. The parameter values can be Variables, Colors, or Predefined ESPL Constant values depending on the statement.

Integer	A signed 32-bit integer in the range of -2147483648 to 2147483647, used for loops and counters.
Real	A double precision floating point variable, used for holding market Price values and decimal values.
Boolean	A boolean variable can have the values of either 0 (False) or 1 (True), used for True or False tests.
String	A dynamically allocated string up to two gigabytes in length, used for holding Text or Ascii characters.
Currency	Used for currency values. Will not have rounding problems.
Variant	A powerful variable type that can hold any value of any type.
TDateTime	Used for Date and Time functions.
TStringList	Used to access properties and methods of String Lists.
TArray	Used to access properties and methods of Arrays.
TFont	Used to access Font properties.
TForm	Used to access Form properties and methods.
THandle	Used to access the Handle of objects.
TScreen	Used to access Screen properties and methods.

Colors

Colors are represented by numeric values. ESPL has defined the following color Constants which can be used any place that a color is required as a parameter. Color constants always start with the letters 'cl'. The colors listed in the 2nd column below will match the color theme on your computer. Include *Graphics* in the *uses* clause of the unit. Example:

```
uses Graphics;
begin
  SetPen(clWhite, 1, eDot);
end;
```

clAqua		cl3DDkShadow	
clBlack		cl3DLight	
clBlue		clActiveBorder	
clDefault		clActiveCaption	
clDkBlue		clAppWorkSpace	
clDkGray		clBackground	
clDkGreen		clBtnFace	
clDkRed		clBtnHighlight	
clFuchsia		clBtnShadow	
clGray		clBtnText	
clGreen		clCaptionText	
clLime		clGrayText	
clLtBlue		clHighlight	
clLtGray		clHighLightText	
clLtGreen		clInactiveBorder	
clLtRed		clInactiveCaption	
clMaroon		clInactiveCaptionText	
clNavy		clInfoBk	
clNone		clInfoText	
clOlive		clMenu	
clOrange		clMenuText	
clPurple		clWindow	
clRed		clWindowFrame	
clTeal		clWindowText	
clSilver			
clWhite			
clYellow			

Constants

Several ESPL commands use predefined ESPL Constants as parameters. The Constants represent numeric values. Using Constant names is easier than programming with numbers. A few examples are shown below. By design, the Constant names start with a lower case 'e'.

eClear	eDate	eTime	eHigh	eLast
eLow	eArrow	eSymbol	eRSI	eSto
eNet	eVolume	eDot	eOpen	eAve

Example: `writeln(GetVariable(eSymbol));`

There are many ESPL constants. They are documented with their specific ESPL commands.

Playback

ESPL programs can be tested with a live simulated data source during or after market hours. For example, if the markets are closed and your charts are not moving, and your ESPL program requires real-time chart data to trigger an event or a signal, then use the Playback feature Selecting **Set-Up | Playback** from the menu. Start a Playback session and do your testing and development using the playback feed.

Program Structure

The structure of an ESPL program is made of two major blocks, 1) Procedure and Function declarations, and 2) Main block of programming code. Both are optional, but at least one should be present in the program. There is no need for the main block to be inside begin..end. It could be a single statement. Statements should be terminated with the ';' character. Begin..end blocks are allowed to group statements.

<p><u>Example 1</u></p> <pre>procedure DoSomething; begin CallSomethingElse; end; begin DoSomething; end;</pre>	<p><u>Example 3</u></p> <pre>function MyFunction; begin result:='Sell the Market'; end;</pre>
<p><u>Example 2</u></p> <pre>begin CallSomething; end;</pre>	<p><u>Example 4</u></p> <pre>CallSomething;</pre>

Variable, Function, and Procedure Names

Variable names, Function names, and Procedure names should begin with a character (a..z or A..Z), or '_', and can be followed by alphanumeric characters or the '_' character. They cannot contain any other characters or spaces.

<u>Valid Names</u>	<u>Invalid Names</u>
VarName2	2Var
_MyProcedureName	My Name
MYFUNCTION99B3	Var-Name
_____MYname_____	This, is, not, valid

Assign Statements

Assign statements are accomplished by using :=

```
MyInteger := 2; {Assigns 2 to MyInteger}
MyString := 'Buy ' + '500 shares.'; {Assigns text to MyString}
```

Strings

Strings (a sequence of characters) are declared using a single quote ' character. Double quotes " are not used. You can also use #nn to declare a character inside a string. There is no need to use the '+' operator to add a character to a string.

```
StringA := 'This is some text';
StringB := 'This text is ' + 'concatenated';
StringC := 'A string ending with CR and LF characters'#13#10;
StringD := 'Some text with ' #40#41 ' characters in the middle';
```

Comments

Comments can be used anywhere in an ESPL program. You can use // characters, or (* *) blocks, or { } blocks. Using // characters will finish at the end of line.

```
writeln('Hello World'); // This is a line ending comment

//This is a comment before ShowMessage
ShowMessage('SELL now');

(* This is another comment *)
ShowMessage('Your trade has been filled');

{ This is another valid comment } ShowMessage('BUY the Market');
```

Variables

There is no need to declare variable types in an ESPL program. Variables are implicitly declared. However, you can optionally declare variables and their variable type using the **var** statement. When **var** is absent the variable is auto-defined upon first detection in the script. The following three examples all work fine.

Example 1

```
procedure MyMessage;
begin
  S:='Place Order Now'; ShowMessage(S);
end;
```

Example 2

```
var A;
begin A:=0; A:=A+1; end;
```

Example 3

```
var S: string;
begin S:='Price Target has been Reached!'; ShowMessage(S); end;
```

Variables have an unknown initialize value. Therefore, assign a value of zero to a variable named `Sum` before using it in a FOR loop like this:

```
Sum := 0;
for I := 1 to 10 do Sum := Sum + I;
```

Indexes

Strings, arrays and array properties can be indexed using "[" and "]" characters. For example, if `Str` is a string variable, then the expression `Str[3]` will return the third character in the string denoted by `Str`, and `Str[N + 1]` would return the character immediately after the one indexed by `N`.

```
MyChar := MyStr[2];
MyStr[1] := 'A';
MyArray[1,2] := 1530;
Lines.Strings[2] := 'Some text';
```

Arrays

ESPL supports array constructors and variant arrays. To construct an array, use "[" and "]" characters. You can construct a multi-index array by nesting array constructors. You can then access the arrays using indexes. If the array is a multi-index array, separate the indexes using the "," character. If a variable is a variant array, ESPL will automatically support indexing with that variable. A variable is a variant array if it was assigned using an array constructor. Arrays in ESPL are all 0-base indexed.

```

NewArray := [ 2,4,6,8 ];
Num := NewArray[1];           {Num receives "4"}
MultiArray:=[['green','red','blue'],['apple','orange','lemon']];
Str := MultiArray[0,2];       {Str receives 'blue'}
MultiArray[1,1] := 'new orange';

```

Case statements

The **Case** statement provides a readable alternative to complex nested **if** conditionals. If the `selectorExpression` matches one of the `caseList` items, then the respective statement (or block of statements) is executed. The `selectorExpression` is any expression of any type. Each value represented by a `caseList` item must be unique.

Statements can be a semicolon delimited sequence of statements. A **Case** statement can have an optional final **else** clause. If none of the `caseList` items have the same value as the `selectorExpression` then the statements in the **else** clause (if there is one) are executed.

```

SYNTAX:
case selectorExpression of
  caseList1: statement1;
  caseList2: statement2;
  ...
  caseListn: statementn;
else
  elsestatements;
end;

```

```

Example:
case MyValue of
  1,2,3,4,5: Caption := 'Low';
  6,7,8,9:   Caption := 'High';
  else      Caption := 'Out of range';
end;

```

Function and Procedure declaration

Procedures and Functions, referred to collectively as routines, are self-contained statement blocks that can be called from different locations in a program. A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value. Function calls, because they return a value, can be used as expressions in assignments and operations: Example: `N := MyFunction(X);`

Declaration of functions and procedures are similar to Delphi, with the difference that you don't specify variable types. To return function values, use a **result** variable. Parameters by reference can also be used.

```

procedure UpcaseMessage (Msg) ;
begin
  ShowMessage (Uppercase (Msg) ) ;
end;

function TodayAsString;
begin
  Result := DateToStr(Date);
end;

function Max(A,B);
begin
  if A>B then Result := A else Result := B;
end;

procedure SwapValues(var A, B); Var Temp;
begin
  Temp := A; A := B; B := Temp;
end;

```

Calling a subroutine

If the script has one or more functions or procedures declared, they can be called by their name.

```

procedure DisplayHelloWorld;
begin
  ShowMessage('Hello world!');
end;

procedure DisplayByeWorld;
begin
  ShowMessage('Bye world!');
end;

begin
  DisplayHelloWorld;
  DisplayByeWorld;
end;

```

Passing parameters

Values of variables can be used in functions and procedures by passing them as parameters. The parameters are Variant types. ESPL doesn't need parameter types.

```

function Double (Num);
begin

```

```
    Result := Num*2;
end;
```

Accessing objects

One powerful feature of ESPL is access to registered objects such as buttons and menus. You can make reference to objects in script, change its properties, call its methods, and so on.

```
btnQuote.Caption := 'New caption';
btnQuote.Click;
```

Component objects can be placed on forms at design time. They can also be created programmatically at run time. Example:

```
uses
    Classes, Graphics, Controls, Forms, Dialogs, Unit2;

var
    MainForm: TForm2;
    btn: TButton;
begin
    MainForm := TForm2.Create(Application);
    MainForm.Show;

    btn := TButton.Create(Application);
    btn.parent := MainForm;
    btn.top := 10;
    btn.width := 100;
    btn.Caption := 'Help';
end;
```

The next example will create a check box on the main ribbon, and assign its OnClick event. Changing the check box state will hide or show draw tools on all charts.

```
uses
  Classes, Graphics, Controls;
var
  chkDraw: TCheckBox;

procedure MyClick; //click will hide/show draw lines on all charts
var I: integer;
begin
  for I := MyChild.Count downto 1 do begin //MyChild has all forms
    Window := I; //Window is global variable to point to a form
    ShowItem(eDrawTool, chkDraw.checked); //set draw flag on charts
  end;
end;

begin
  if ESPL = 1 then begin //execute via the 1 button on the ESPL form
    chkDraw := TCheckBox.Create(nil);
    chkDraw.parent := pagMain; //the Main ribbon will own the box
    chkDraw.caption := 'Draw Tools';
    chkDraw.left := 840; //widen Main ribbon to see the check box
    chkDraw.top := 11;
    chkDraw.OnClick := 'MyClick'; //name of procedure to handle click
  end;

  if ESPL = 2 then begin
    chkDraw.visible := false;
    chkDraw.free; //remove the object
  end;
end;
```

The other available ribbon panels are named pagSetup, pagWindow, and pagHelp.

Calling DLL functions

ESPL allows importing and calling of external DLL functions, by inserting special directives on declaration of script routines, indicating library name and, optionally, the calling convention, beyond the function signature. External libraries are loaded on demand, before function calls, if not loaded yet. See [Creating ESPL DLLs](#) example.

SYNTAX:

```
Function  functionName(arguments): resultType; [callingConvention];  
external 'libName.dll' [name ExternalFunctionName];
```

EXAMPLE:

```
function MyFunction(arg: integer): integer; external 'CustomLib.dll';
```

The example above imports a function called MyFunction from CustomLib.dll. Default calling convention, if not specified, is register. ESPL also allows you to declare a different calling convention (stdcall, register, pascal, cdecl or safecall) and to use a different name for DLL function, like the following declaration:

```
function MessageBox(hwnd: pointer; text, caption: ansistring;  
msgtype: integer): integer; stdcall; external 'User32.dll' name  
'MessageBoxA';
```

that imports the 'MessageBoxA' function from User32.dll (Windows API library), named 'MessageBox' to be used in script. The Declaration above can be used with functions and procedures (routines without a **result** value).

msgtype	integer
MB_OK	0
MB_OKCANCEL	1
MB_ABORTRETRYIGNORE	2
MB_YESNOCANCEL	3
MB_YESNO	4
MB_RETRYCANCEL	5
MB_ICONHAND	16
MB_ICONQUESTION	32
MB_ICONEXCLAMATION	48
MB_ICONASTERISK	64
MB_ICONWARNING	48
MB_ICONERROR	16
MB_ICONINFORMATION	64

Supported Types

ESPL supports following data Types on arguments and result of external functions:

Integer	PWideChar	Single
Boolean	AnsiString	Byte
Char	Currency	Shortint
Extended	Variant	Word
String	Interface	Smallint
Pointer	WideString	Double
PChar	Int64	Real
Object	Longint	DateTime
Class	Cardinal	Comp
WideChar	Longword	

Others types (records, arrays, etc.) are not supported. Arguments of the above types can be passed by reference, by adding **var** in the param declaration of the function.

Include Libraries

ESPL allows you to include multiple files (or libraries of code). Use the **uses** statement to specify the files to include in the current program file. For example,

```
{This is the first program file named Script1}
uses Script2;
begin
  Script2GlobalVar := 'Hello world!';
  ShowScript2Var;
end;

{This is the second program file named Script2}
var Script2GlobalVar: string;
procedure ShowScript2Var;
begin
  ShowMessage (Script2GlobalVar);
end;
```

When you execute the first script, it "uses" Script2, and is able to read and write global variables and call procedures from Script2. Script1 must know where to find Script2 via its identifier in the uses clause, for example:

```
uses Classes, Forms, Script2;
```

Commonly used libraries: Buttons, Classes, ComCtrls, Controls, Dialogs, ExtCtrls, Forms, Graphics, ImgList, IniFiles, Menus, StdCtrls.

Libraries are typically added automatically to a unit's **uses** statement as components are added to forms. For example, adding a TButton object to a form will automatically add **Buttons** to the **uses** statement.

Secure Library Files

When units are saved, two files are written. The files with the .psc are the ASCII source files for the script that is displayed in the editor. A non ASCII library file with a .lib extension is also saved. It is sufficient to distribute the .lib library file instead of the .psc source file to other users. Follow this process for distributing library files for security and secrecy when you do not want the source code to be displayed or changed.

1. Click the Save All button on the editor form. The library files are created.
2. Use menu *File | Remove from Project* to remove the 2nd, 3rd, or other units. Keep the main unit as that unit needs to remain. In the previous example, the Script2 unit could be removed from the project, but kept in the **uses** statement.
3. The main unit has a **uses** statement with references to the units removed from the project. In the previous example, the Script2.lib file would be distributed.
4. Distribute in a package the project file with its .sproj extension. Distribute the .lib file for each unit removed from the project. Distribute any .sfm form files.

Declaring Forms in ESPL

A powerful feature in ESPL is the ability to declare forms and use .sfm files to load form resources. With this feature you can declare a form to use it in a similar way as Delphi. For example,

```
//Main script
uses
  Classes, Forms, MyFormUnit;
var
  MyForm: TMyForm;
begin
  {Create instances of the forms}
  MyForm := TMyForm.Create(Application);
  {Initialize all forms calling its Init method}
  MyForm.Init;
  {Set a form variable. Each instance has its own variables}
  MyForm.MyFormGlobalVar := 'my instance';
  {Call a form "method". You declare the methods}
  MyForm.ChangeButtonCaption('Another click');
  {Accessing form properties and components}
  MyForm.Edit1.Text := 'Default text';
  MyForm.Show;
end;
```

```

//My form script
{$FORM TMyForm, myform.dfm}
var MyFormGlobalVar: string;
procedure Button1Click(Sender: TObject);
begin
    ShowMessage('The text typed in Edit1 is ' + Edit1.Text + #13#10 +
'And the value of global var is ' + MyFormGlobalVar);
end;

procedure Init;
begin
    MyFormGlobalVar := 'null';
    Button1.OnClick := 'Button1Click';
end;

procedure ChangeButtonCaption(ANewCaption: string);
begin
    Button1.Caption := ANewCaption;
end;

```

The sample scripts above show how to declare forms, create instances, and use their "methods" and variables. The second script is treated as a regular Library, so it follows the same concept of registering and using. The \$FORM directive is the main piece of code in the form script. This directive tells the compiler that the current script should be treated as a form class that can be instantiated, and all its variables and procedures should be treated as form methods and properties. The directive should be in the format {\$FORM FormClass, FormFileName}, where FormClass is the name of the form class (used to create instances, take the main script example above) and FormFileName is the name of a .sfm form which should be loaded when the form is instantiated.

The .sfm file is a regular Delphi file format, in text format. You cannot have event handlers defines in the sfm file, otherwise an error will raise when loading the sfm.

Event Redirection

This example shows how ESPL can be notified when the Ensign program is closing so that information can be saved before the program closes.

Redirect the OnCloseQuery event for the main Ensign form to an ESPL procedure which performs the clean-up tasks such as saving information. The main Ensign form is referenced with the component named `frmMain`. This example will print a message in the Output window when Ensign closes.

```
uses
  Forms;

procedure ShutDown;
begin
  writeln('Exiting');
end;

begin
  if ESPL = 3 then
    frmMain.OnCloseQuery := 'ShutDown';
end;
```

Click ESPL button 3 to establish the redirection of the OnCloseQuery event. Then when Ensign exits, the OnCloseQuery event fires and executes the ESPL ShutDown procedure which displays 'Exiting' in the Output window. Ensign continues its exiting processes and closes down.

The following Forms events can be redirected: (requires Forms in the Uses clause)

- OnClose
- OnCloseQuery
- OnHelp
- OnException
- OnGetHandle
- OnIdle
- OnSettingChange

The following Classes events can be redirected: (requires Classes in the Uses clause)

- OnNotify All events in any object that are of TNotifyEvent type are supported.
(Button: OnClick, OnMouseEnter, OnMouseDown,
Combo: OnChange, OnEnter, OnExit, etc.)