

ESPL Manual

Ensign Software Programming Language

Ensign 10

Copyright © 2014 Ensign Software, Inc.

Last Update: 31 January 2014

Table of Contents

Introduction	11
ESPL Language Features	11
Documentation Format	12
ESPL Programming Window	13
Projects	14
Suggestions for Designing an ESPL Program	14
Creating a new Project	14
Opening an Existing Project	15
Editing your ESPL Program	15
Designing Forms	15
Tool Palette and Components	16
Object Inspector	17
Adding some Programming Code	18
Running a Program	19
Saving the project	19
Adding more Features	20
Changing Component Properties	20
Writing Code - Events and Event Handlers	22
Code completion	23
Debugging scripts	24
ESPL Programming	25
Variable Types	25
Colors	26
Constants	27
Playback	27
Program Structure	27
Variable, Function, and Procedure Names	28
Assign Statements	28
Strings	28
Comments	28
Variables	29
Indexes	29
Arrays	29
Case statements	30
Function and Procedure declaration	30
Calling a subroutine	31
Passing parameters	31
Accessing objects	32
Calling DLL functions	33
Supported Types	34
Include Libraries	34
Secure Library Files	35
Declaring Forms in ESPL	35

Event Redirection	37
ESPL Statements	38
Abs	38
Account	38
ActiveChild	
ActiveChart	39
AddLine	39
AddNote	41
AddOverlay	42
AddStudy	
AddStudyOnStudy	43
Alert	
GetAlert	46
AlertEvent	47
Align	49
And	50
Application	50
Arc	
Chord	
Ellipse	
Pie	51
ArcCos	
ArcSin	
ArcTan	
Cos	
CoTan	
Sin	
Tan	53
Arrays	54
AutoESPL	56
AutoRefresh	57
Ave	
ExpAve	
Sum	57
Average	
ExpAverage	
Summation	58
Bar	
ChartBar	59
BarBegin	
BarEnd	
BarLeft	
BarRight	
BarBeginLeft	61
Beep	62
Begin...End	62
Bullet	63

Buttons	64
CallBack	66
Chart	67
ChartLoad	67
ChartRefresh	68
ChartReplace	69
ChartSave	70
Chat	
ChatRoom	70
ChDir	
MkDir	
Rmdir	71
ChildCount	71
Child	72
Choose	72
Chr	
Ord	74
Clipboard	74
AssignFile	
Append	
Reset	
Rewrite	
CloseFile	
ReadLn	
WriteFile	
WriteLnFile	
EOF	
DeleteFile	
FileExists	
DirectoryExists	
RenameFile	76
ColorBars	79
ConvertPrice	
FormatPrice	80
Copy	81
CopyFile	81
CreateProcess	82
Date	
DateStr	82
DateToLong	
LongToDate	
LongToTime	
TimeToLong	
TimeToString	
DwordToTDate	
TDateToDword	83
DateToStr	84

DateToString	84
DayOfWeek	84
Dec	
Inc	85
DecodeDate	85
DecodeTime	86
Delete	86
DeleteBar	87
DeleteData	87
DimArray	88
Div	88
dlgColor	
dlgColor2	89
dlgFont	89
dlgOpen	
dlgSave	90
dlgPrint	
dlgPrinterSetup	91
Download	91
DownloadData	92
DrawPhase	93
Drawing	94
Email	94
EmailForm	94
EmailFormTab	96
EncodeDate	96
EncodeTime	97
Encrypt	
Decrypt	
Hash	97
Exp	
Ln	
Log2	
Log10	98
ExtractFileDrive	
ExtractFileExt	
ExtractFileName	
ExtractFilePath	98
Filter	99
Find	
FindMarket	101
FindClose	
FindFirst	
FindNext	102
FindStudy	
FindStudyName	103
FindWindow	105

Finished	106
Flash	107
FloatToStr	
StrToFloat	107
FloodFill	108
For	108
ForceDirectories	109
Format	109
FormatDateTime	111
Formation	112
Frac	
Round	
Trunc	114
FTPdownload	
FTPupload	114
Function	115
Get	116
GetBar	116
GetCell	
SetCell	
SelectedCell	
RowColor	117
GetData	118
GetLevels	119
GetStudy	
SetStudy	120
GetToken	
SetToken	129
GetUser	
SetUser or Plot	130
GetVariable	
SetVariable	134
GV Global Variables	138
Highest	
Lowest	139
Holiday	140
HTTP	140
IF..Then..Else	141
Import	141
ImageToFile	142
Index1	
Index2	
Index3	
Index4	
Index5	
Index6	143

IndexToX	
XToIndex	143
Initialize	144
InputBox	
InputQuery	144
Insert	145
InsertBar	145
IntToHex	
IntToStr	
StrToInt	
StrToPrice	146
IsNumeric	147
IsSelected	147
IT	147
KeyDown	148
LeftStr	
RightStr	
ReverseString	149
Length	150
LineTo	
MoveTo	
MoveToLineTo	150
Layout	
LayoutName	
LayoutOpen	151
LowerCase	
UpperCase	
UpCase	152
Manager	152
Max	
Min	153
Menu Commands	154
Merge	155
MessageDlg	
MessageDlgPos	156
Mod	157
Mouse	157
NewsFind	
NewsStory	
NewsText	
NewsTitle	
NewsSymbol	158
Now	159
Output	160
Pause	161
Pi	161
Play	161

Pos	162
Power	162
Pred	
Succ	163
PriceToY	
YToPrice	163
Pt1X, Pt2X, Pt3X, Pt1Y, Pt2Y, Pt3Y	164
PtX1, PtX2, PtX3, PtX4, PtX5, PtX6	
PtY1, PtY2, PtY3, PtY4, PtY5, PtY6	165
Quote	165
Random	
Randomize	166
Rectangle	
RoundRect	166
Register	167
Regression	168
Remove	169
Repeat...Until	170
ResetTrades	170
SaveToAscii	171
Scheduler	172
Screen	173
sCustom	174
Select	174
Section	175
SendKeys	176
SetArray	177
SetBar	177
SetBrush	
SetPen	178
SetData	180
SetDateTime	180
SetMyFocus	181
SetLength	181
SetLine	181
SetStudyLine	182
ShellExecute	183
Show	184
ShowMessage	
ShowMessagePos	185
sList	186
sLog	
sStudyLog	
sLineLog	
sSoundLog	186
sPath	187
Speak	187

Spreadsheet	187
Sqr	
Sqrt	188
Std	
StdDev	189
Str	190
String Lists	191
StringToDate	
StrToDate	194
System	195
Template	196
TCP Connections	196
TextAdd	
TextBox	
TextCaption	
TextClear	197
TextOut	198
TextWidth	199
TFont	199
TForm	200
Time	201
Timer	201
TimeStr	202
Top100	202
Trade	203
TradeReport	204
Trim	
TrimLeft	
TrimRight	205
UDP Connections	205
Put	
Update	206
Val	207
Var	207
VarToStr	208
vArray	208
VarType	209
Volatility	209
WWW	210
While...Do	210
Window	211
WinExec	211
WinExist	212
Write	
Writeln	212
ESPL Sample Programs	214
Plotting Study Lines on a Chart	214

ToolBar and ToolButton	215
TStringGrid Example	216
Study Rising Falling Flag	217
Creating ESPL DLLs	219
Appendix	222
USES Clause Libraries	222
StrUtils Library Statements	222
Additional ESPL Statements	222
SysUtils Library Statements	223
Study Constants	224
Data Point Constants	224
Bar Constants	225
Feed Constants	225
Markers	226
Image List	227

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.
Courier	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.

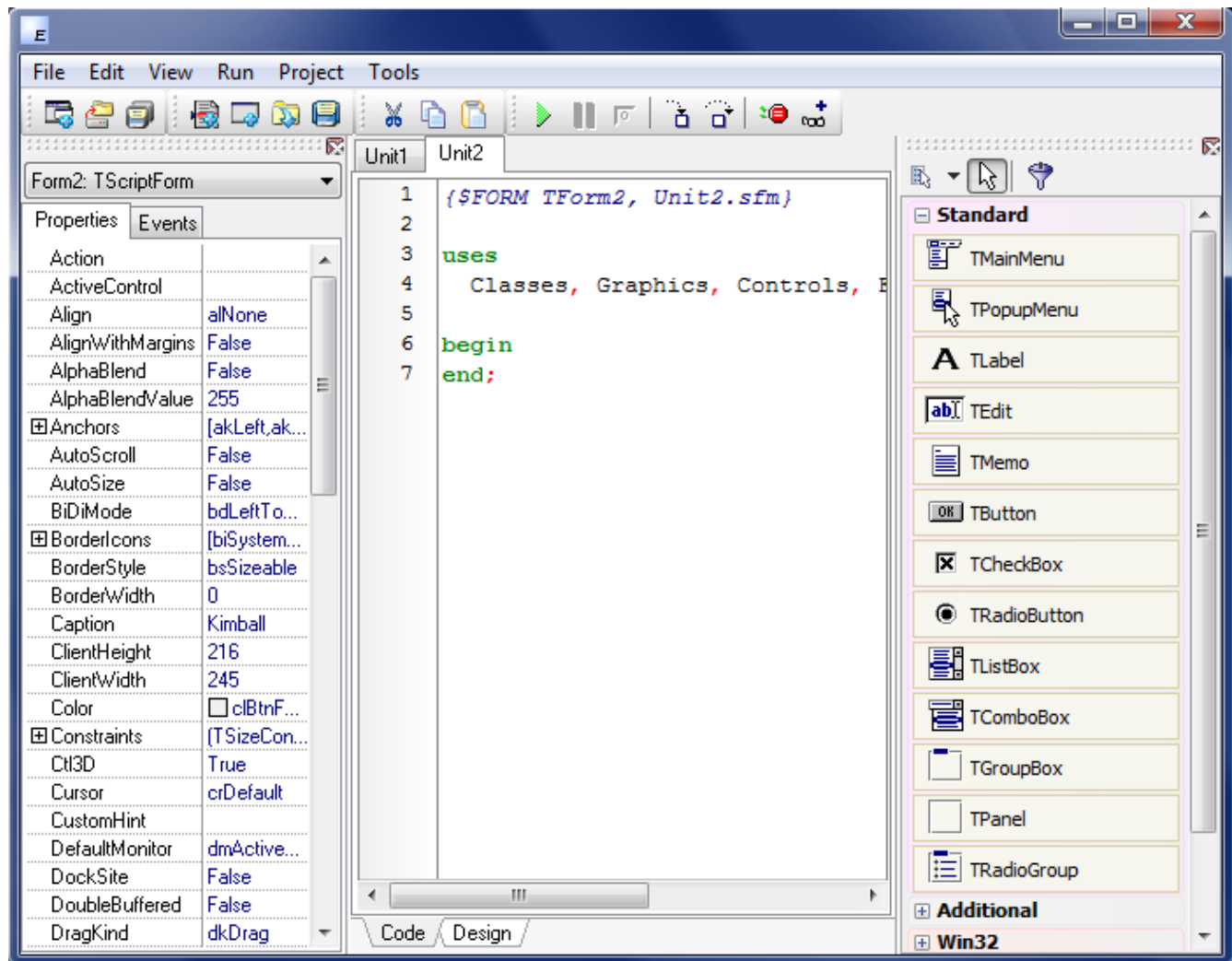
Each documented function and statement uses the following format.

<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:

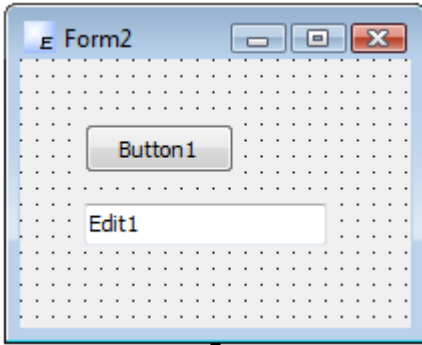
- code completion (pressing Ctrl+Space)
- syntax highlight
- line numbering
- clipboard operations
- automatic indentation

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.



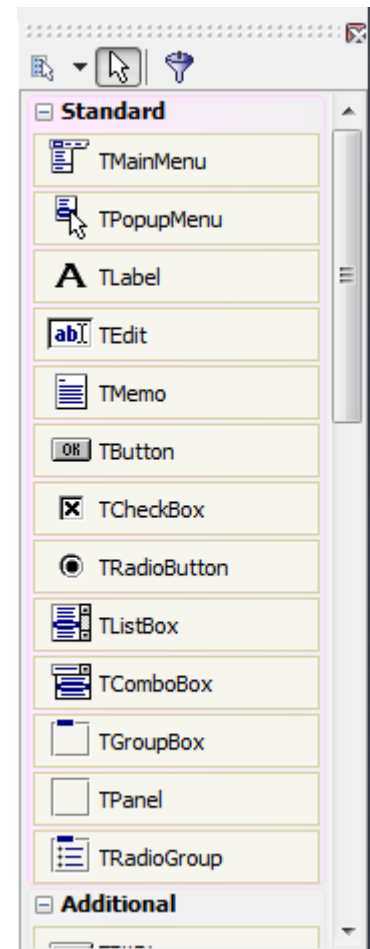
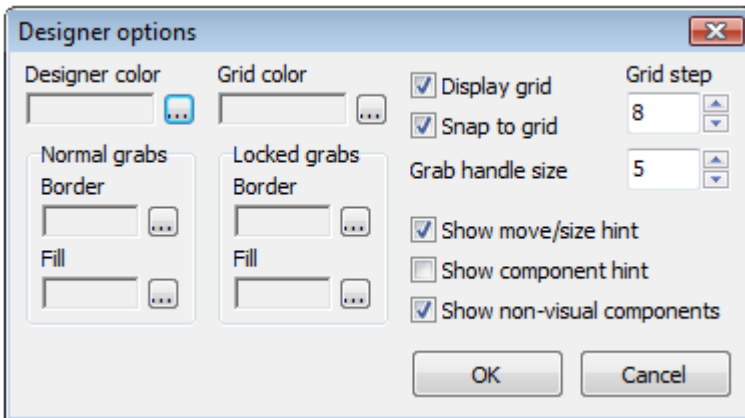
In the following simple example, a button and an edit box have been placed on a form.



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.

```

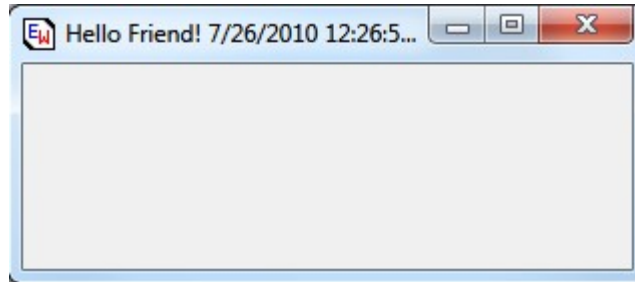
1  {$FORM TForm2, Unit2.sfm}
2
3  uses
4    Classes, Graphics, Controls, Forms, Dialogs;
5
6  procedure Form2Create(Sender: TObject);
7  begin
8    Caption := 'Hello Friend! ' + DateTimeToStr(Now);
9  end;
10
11 begin
12 end;

```

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

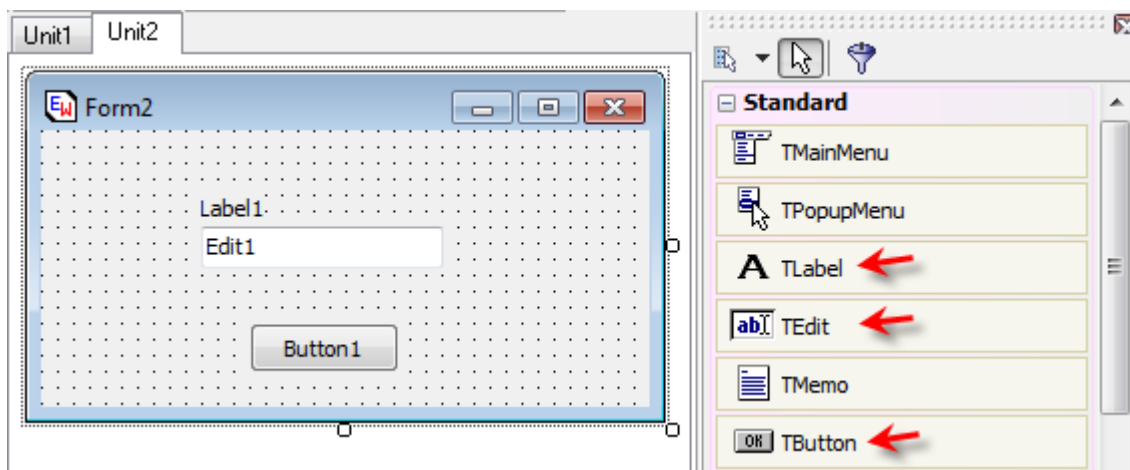
Note: In the Code Editor window, the Unit1 tab will be changed to Main.

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.

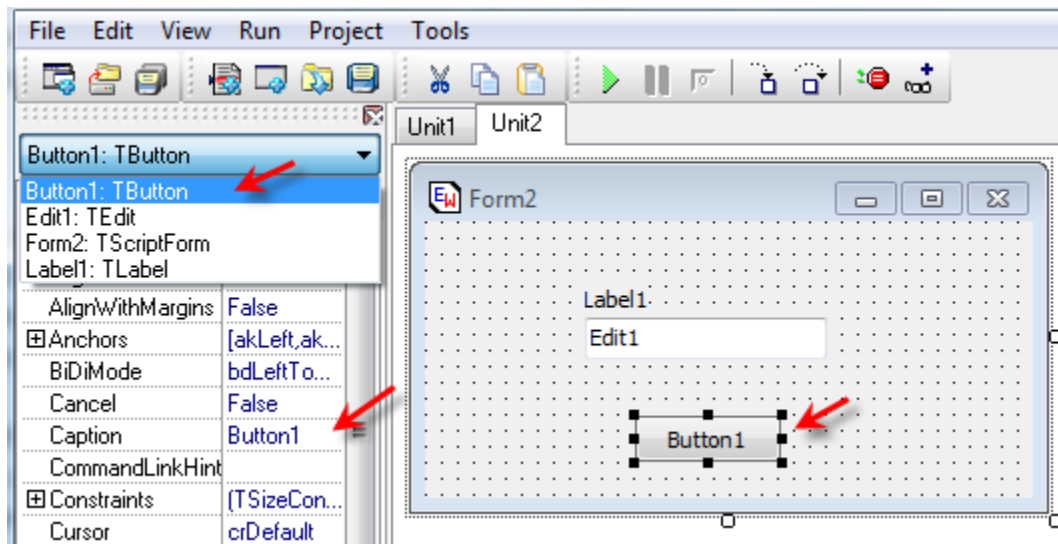
All the components have a property called Name. The Name property specifies the name of the component as referenced in the programming code. When you first place a component on a form, ESPL will provide a default name for the component, such as Label1, Edit1, Button1. A good programmer will usually change 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.

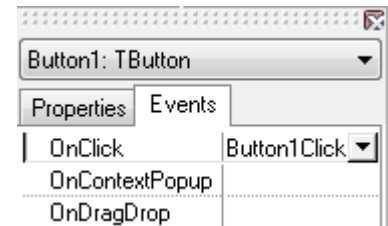
Writing Code - Events and Event Handlers

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.

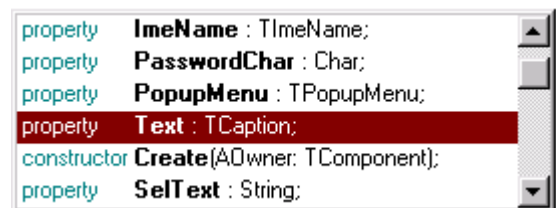
As you will understand more clearly through this course, a procedure must have a unique name within the form. The above procedure, ESPL component event-driven procedure, is named for you. The name consists of: the name of the component name "Button1", and the event name "Click". For any component there 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:

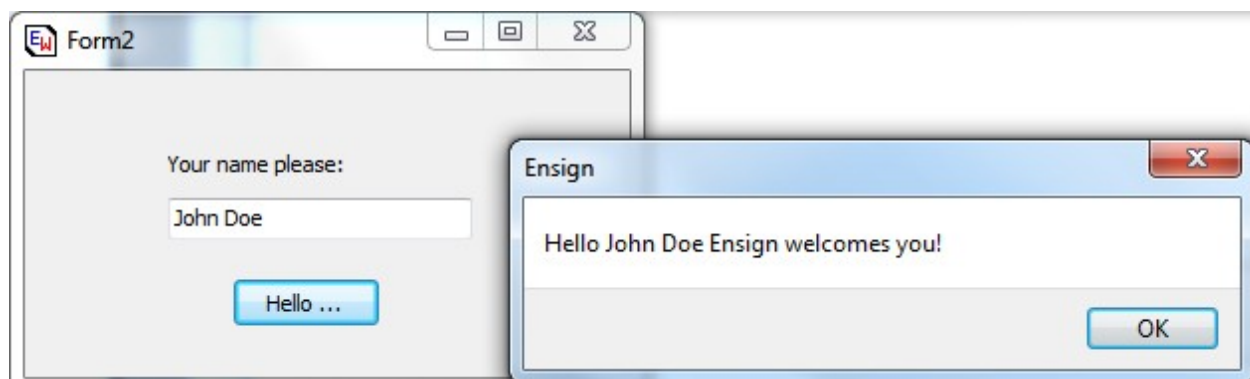
```
1  {$FORM TForm2, Unit2.sfm}
2
3  uses
4      Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
5
6  procedure Button1Click(Sender: TObject);
7  begin
8      s := 'Hello ' + Edit1.Text + ' Ensign welcomes you!';
9      ShowMessage(s);
10 end;
```

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:

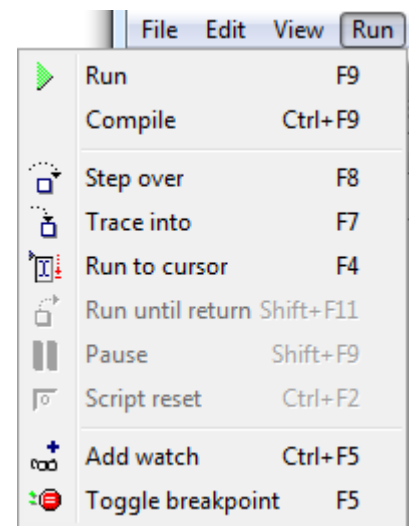
1. Play with the Color property of the Form object
2. Use the Font property Editor to alter the font of the TLabel component
3. Find out about the PasswordChar property of the TEdit component to create a simple password dialog form
4. Try adding code for the OnCreate event of a form to make a form appear 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;
  DisplayByWorld;
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;
```


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.)

ESPL Statements

The ESPL Statements are listed in alphabetical order. A small code example is included with each statement.

Abs

SYNTAX: **Abs**(*numeric expression*);

DESCRIPTION: The **Abs** function returns the absolute value of a numeric expression. The absolute value is the unsigned (non-negative) value of its parameter. For example, **Abs**(-50) and **Abs**(50) are both equal to 50. The numeric expression can be an integer-type or real-type expression.

EXAMPLE:

```
var                                {Start of variable declarations}
  x,y: real;                        {x and y are declared as Real variable type}
begin                               {Start of Main programming code}
  x := Abs(-10.3);                  {x is assigned value of the Abs function = 10.3}
  y := Abs(10-30);                  {y is assigned value of the Abs function = 20}
  writeln(x, ' ', y);               {x and y are printed in the output window}
end;                                {End of Program}
```

Account

SYNTAX: **Account**(*Number*: integer): integer;

DESCRIPTION: The **Account** function allows you to open and retrieve Trading Account information from the Account files in Ensign. This can be useful to retrieve account balances, market value, average trade value, total number of trades, winning and losing trades, account names, account numbers, account phone numbers, etc. The **Account** function opens the specified account *Number* window. Ensign will display the trading account window. The values in the account window can then be retrieved by using the **GetVariable** function. The **Account** function will return the Window handle number. See the **GetVariable** documentation for details on retrieving account information from the active account window.

PARAMETERS:

Number: Enter the Trading Account number to open. Example, the number 3 would open trading account 3.

EXAMPLE: The following example will open and retrieve account information from accounts 1 through 3. Notice how account information can be read directly from the cells in the trading account window, using the **GetCell** command.

```
var                                {Start of Variable declarations}
  i,j: integer;                    {Declare i and j as integers}
begin                               {Start of Main programming code}
  for i := 1 to 3 do                {Loop from 1 to 3}
  begin                               {start of loop block of code}
    Account(i);                     {Open the Account Window}
    writeln(GetVariable(eName));     {Print Account Name}
    writeln('Balance ',GetVariable(eProfit)); {Print Account Balance}
    writeln('Trade Count ',GetVariable(eTrades)); {Number of Trades}
    writeln('Profit Trades ',GetVariable(eWinTrades)); {Print Winning Trades}
```

```

writeln('Loss Trades ',GetVariable(eLossTrades)); {Print Losing Trades}
for j := 1 to GetVariable(eGrid) do writeln(GetCell(j,1));
writeln(''); {Print a blank line}
mnuCloseWindow.click; {Close Account Window}
Application.ProcessMessages;
end; {End of loop block of code}
end; {End of program}

```

ActiveChild ActiveChart

SYNTAX: **ActiveChild;**
 ActiveChart;

DESCRIPTION: Ensign allows multiple windows to be opened from the main application toolbar. All of the Chart windows, Quote windows, Alert windows, News windows, etc., are child windows to the Ensign application. The **ActiveChild** function is used to determine which child window currently is active and has the focus. A pointer value is returned from the function which can be used to identify the active child window. The pointer is a property of a *TForm* variable type. The following example program declares a *TForm* variable, and then finds the **ActiveChild** window. The window is then minimized.

ActiveChart is similar to **ActiveChild**. When a stack has focus, **ActiveChild** will return the stack whereas **ActiveChart** will look inside of the stack and return the chart on the surface of the stack. **ActiveChart** returns the chart that has the focus or which most recently had the focus.

EXAMPLE:

```

var {Start of variable declarations}
  Form1: TForm; {Form1 declared as TForm variable type}
begin {Start of Main programming code}
  FindWindow(eChart); {Find an Open Chart Window}
  SetMyFocus; {Set the Focus so it is the Active window}
  Form1 := ActiveChild; {Form1 is assigned ActiveChild window}
  Form1.WindowState := wsMinimized; {WindowState is Minimized}
end;

```

AddLine

SYNTAX: **AddLine**(*Name, Tab, Index1*: integer, *Price1*: real, [*Index2*: integer, *Price2*: real, *Index3*: integer, *Price3*: real, *Location*: constant]): integer;

DESCRIPTION: The **AddLine** command is used to draw a Line or Draw Tool on a chart. The line settings from the specified *Tab* are used when the tool is drawn. *Index* and *Price* parameters specify the location of the tool on the chart. Some tools require up to 3 chart points. If desired, the *Location* parameter can be used to draw the tool in a study sub-window. The default location is on the chart. The **AddLine** command returns the Object ID (handle) of the Draw Tool. The Object ID is used to identify the tool and can be used by the **SetLine**, **GetStudy**, **SetStudy**, and **Remove** commands to perform further operations on the tool. The **AddLine** command returns a zero value if an error occurs, or the line object was unable to draw on the chart. Use the **GetStudy**, **SetStudy**, and **SetLine** commands to get and set the colors, styles, tokens, and percentage levels (after the Line or Draw Tool is already on the chart).

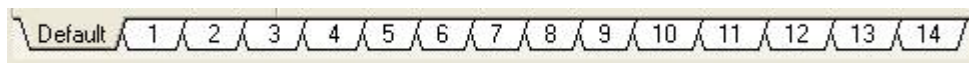
PARAMETERS:

Name: Specifies the line object to draw. The following are predefined type names that can be used to identify the line type.

eAlan	2 – draws an Alan Square on the chart
eAndrews	3 - draws an Andrew's Pitchfork tool on the chart
eArrow	1 - draws a Marker on the chart (Note: there are dozens of possible Markers)
eCircle	2 - draws a Circle on the Chart
eCycle	2 - draws a Cycle tool on the chart
eESPLTool	3 - draws an ESPL programmed Draw tool on the chart
eFibCycle	2 - draws a Fibonacci Cycles tool on the chart
eFibRuler	1 – draws a Fibonacci Ruler tool on the chart
eFibonacci	2 - draws Fibonacci lines on the chart
eGann	2 - draws Gann lines on the chart
eGannSquare	2 - draws a Gann Square on the chart
eGartley	3 – draws a Gartley Butterfly on the chart
eLevel	1 - draws the Daily Price Levels tool on the chart
eLine	2 - draws a Line on the chart
eLinear	2 - draws a Linear Regression line on the chart
eParallel	3 - draws the Parallel lines tool on the chart
ePyraPoint	1 - draws a PyraPoint tool on the chart
eRetrace	2 - draws Fibonacci Retracement lines on the chart
eSpeed	2 - draws Speed Lines on the chart
eSupport	3 - draws the Support and Resistance tool on the chart
eWave	3 - draws Elliott Wave lines on the chart

Note: 1= requires 1 chart location point, 2= requires two points, 3= requires three points. For example, the eArrow, ePyraPoint, and eLevel objects use only *Index1* and *Price1* since there is only one chart point to indicate. The eLinear object uses only *Index1* and *Index2* since Price is not considered. In this case enter *Price1* and *Price2* with zero values.

Tab: Specifies the *Tab* setting to use from the Draw Tools properties window. Enter a number from 0 to 14. For example, an entry of 0 will draw the tool using all the colors, styles, and marker settings from the 'Default' tab.



Index1: Specifies the chart bar index position for the starting point of the line object.
Index2: Specifies the chart bar index position for the ending point (2nd point) of the line object. :
Index3: Specifies the chart bar index position for the ending point (3rd point) of the line object. :
For example, if a chart contains 1000 bars, a line can be drawn from bar index 950 to 1000.

Price1: Specifies the chart price for the starting point of the line object.
Price2: Specifies the chart price for the ending point (2nd point) of the line object. :
Price3: Specifies the chart price for the ending point (3rd point) of the line object. :

Location: 0 or eChart - Draws the line object directly on the Chart (default).
1-9 - Draws the line object in the specified study sub-window 1 through 9.
eVolume - Draws the line object in study sub-window 9 (the Volume window).

EXAMPLE: The following program adds 4 draw tools to an IBM daily chart. The first tool is a simple Line which requires 2 chart points (using *Tab* setting 1). The second tool is a Fibonacci Levels draw tool which requires 2 chart points (using *Tab* setting 5). The third item is an Andrew's Pitchfork tool which requires 3 chart points (using *Tab* setting 1). The fourth tool is an Arrow Marker which requires 1 chart point (using *Tab* setting 1).

var


```

Bar1,Bar2,Bar3:integer;
Price1,Price2,Price3:real;
begin
Chart ('IBM.D');
Bar1:=BarEnd-75;   Bar2:=BarEnd-50;   Bar3:=BarEnd-25;
Price1:=Last (BarEnd-75);
Price2:=Last (BarEnd-50);
Price3:=Last (BarEnd-25);
AddLine (eLine,1,Bar1,Price1,BarEnd,Last (BarEnd));
AddLine (eLinear,1,Bar1,0,BarEnd,0);
AddLine (eFibonacci,5,Bar1,Price1,Bar2,Price2);
AddLine (eAndrews,1,Bar1,Price1,Bar2,Price2,Bar3,Price3);
AddLine (eParallel,1,Bar1,Price1,Bar2,Price2,Bar3,Price3);
AddLine (eArrow,1,Bar3,High (Bar3));
end;

```

AddNote

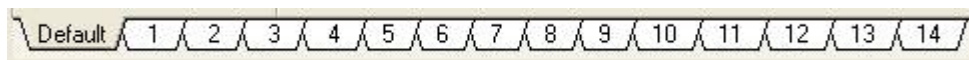
SYNTAX: **AddNote**(*NoteText*: string, *Tab*, *Index*: integer, *Price*: real [, *Location*: constant]): integer;

DESCRIPTION: The **AddNote** function is used to place a Note or Text on a chart. The note can have a variety of frames, colors, markers, shadows, and circles. The note can be Pinned to the chart, or float over the chart (not pinned). The note will use the settings from the specified *Tab* selection. The *Index* and *Price* parameters specify the note location on the chart. If desired, use the *Location* parameter to place the note in a study sub-window. A note can also be placed on the right edge of a chart in the Scale and Price Arrow area. **AddNote** returns the Object ID number for the note, or returns a zero value if adding the Note was unsuccessful. Use the **GetStudy** and **SetStudy** commands to get or change the Note settings after the note has been placed on the chart.

PARAMETERS:

NoteText: Specifies the note that is placed on the chart (example, 'Buy Here').

Tab: Specifies the *Tab* setting to use from the Note properties window. Enter a number from 0 to 14. For example, an entry of 0 will draw the Note using all the colors, styles, and marker settings from the 'Default' tab.



Index: Specifies the bar index where the note will be positioned horizontally on the chart. Example, an entry of 100 will position the note on the 100th bar of the chart.

If the note is placed in the price Scale (on the right edge of the chart), then the *Index* parameter should represent the horizontal pixel count from the left edge of the Scale panel. The vertical Scale grid line is at pixel 16.

A negative value will set the horizontal pixel position. -10 would start the note at the 10th pixel.

Price: Specifies the price level on the chart where the note will be positioned.

A negative value will set the vertical pixel position. -5 would locate the note 5 pixels down from the top of the chart.

Location: 0 or eChart - Draws the Note directly on the Chart (default).

- 1-9 - Draws the Note in the specified study sub-window 1 through 9.
- eVolume - Draws the Note in study sub-window 9 (the Volume window).
- 14 or eScale - Places the Note in the Price Scale (right edge of the chart).

EXAMPLE: The following program opens an IBM daily chart, and then adds a Note to the chart.

```
begin
  Chart('IBM.D');
  AddNote('Sell Here',1,BarEnd-20,High(BarEnd-20));
end;
```

EXAMPLE: The following will add the security name to the chart as a note. The example uses the tab 4 properties.

```
begin
  if ESPL = 9 then begin
    FindWindow( eChart); // Find the chart with focus
    sSymbol := GetVariable( eSymbol ); // Get the chart's symbol
    Find( eIQFeed, sSymbol ); // Get the symbol's quote record
    sName := GetData( eName ); // Read the security name
    {Go set tab 4 to have Pinned unchecked. Set the colors and font size}
    AddNote( sName, 4, -10, -10, 0 ); // Locate the note at x,y = (10,10)
  end;
end;
```

AddOverlay

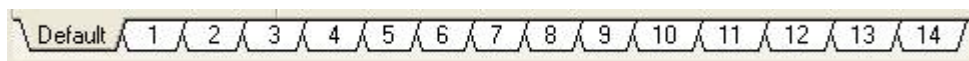
SYNTAX: **AddOverlay**(Chart: string, Tab, [Overlay, Host, Offset, Shift, Location, Type, Scale, Alignment, Grid: real]): integer;

DESCRIPTION: The **AddOverlay** function overlays a chart on another chart. This is a great way to view two or more charts within the same chart window. The settings from the specified *Tab* are used when the overlay is drawn. Optional multipliers, offsets, locations and other settings can also be specified. The **AddOverlay** function returns the Object ID number (handle) for the overlay. The function returns a zero value if adding the overlay was unsuccessful. Use the **GetStudy**, **SetStudy**, and **SetLine** commands to change the settings, colors and line styles of the overlay after it has been drawn. Since there may be multiple Chart windows open at the same time, it is important to first identify the Chart that the overlay should be placed on. A global ESPL variable named *Window* is used to specify which chart to place the overlay on. The *Window* variable can be manually assigned a window pointer value (if you have been keeping track of the window handles), or you can use the **FindWindow** or **Chart** functions to set the *Window* variable.

PARAMETERS:

Chart: Specifies the chart symbol to overlay on the host chart.

Tab: Specifies the *Tab* setting to use from the Overlays properties window. Enter a number from 0 to 14. For example, an entry of 0 will draw the Overlay using all the colors, styles, and marker settings from the 'Default' tab.



Overlay: Is an optional multiplier for the overlay data. This can be used to shift (multiply) all of the overlay chart prices to a different price scale. The default is 1.

- Host:* Is an optional multiplier for the host data. This can be used to shift (multiply) all of the host chart prices to a different price scale. The default is 1.
- Offset:* Specifies an optional vertical price adjustment in the overlay data. The default is 0.
- Shift:* Specifies an optional left or right shift of the overlay data. The default is 0.
- Location:* 0 or eChart - Draws the Overlay directly on the Chart (default).
1-9 - Draws the Overlay in the specified study sub-window 1 through 9.
eVolume - Draws the Overlay in study sub-window 9 (the Volume window).
- Type:* Specifies the Chart Type for the overlay chart. The following Chart Types can be used. The default is 0.
0 = Plot the overlay chart using the same Chart Type as the host chart.
1 = Plot the overlay chart as a Bar Chart.
2 = Plot the overlay chart as a Line Chart.
3 = Plot the overlay chart as a Japanese Candlestick chart.
4 = Plot the overlay chart using Close Ticks Only.
5 = Plot a Spread Chart. The spread chart equals the (host - overlay).
6 = Plot a Ratio Chart. The ratio chart equals the (100* host / overlay).
- Scale:* Specifies which price scale to use for the overlay chart. The following *Scale* options are available choices. The default is 1. Double-click the mouse on the chart scale numbers to switch the view of the scale prices.
0 = Use the host chart's price scale (the overlay chart data will be scaled using the host's range).
1 = Scale the overlay data using its own price range. The overlay price scale numbers will not be visible.
2 = Scale the overlay data using its own price range. The chart will display the overlay's scale numbers.
3 = Don't plot the overlay. The overlay data is not visible but the data is still available for calculations.
- Alignment:* Specifies an optional Alignment adjustment of the overlay bars. Shifting the Overlay prevents the overlay bars from drawing exactly on top of the chart bars. 0=Align to left side, 1=Align to center, 2=Align to right side.
- Grid:* Specifies and draws grid lines in the study sub-window. The *Grid* parameter is ignored if the study is not drawn in a study sub-window.

EXAMPLE: The following example opens an IBM daily chart. A Microsoft daily chart is then overlaid on the IBM chart (using *Tab* setting 1). The Microsoft chart is drawn using its own price scale, and is plotted as a Line chart with no price offsets nor multipliers. The price scale for the Microsoft chart is hidden. NOTE: Double-click the mouse on the IBM price scale numbers, on the chart, to switch the view to the Microsoft price scale numbers.

```
begin
  Chart('IBM.D');                               {Open IBM daily chart}
  AddOverlay('MSFT',1,1,1,0,0,eChart,2,1,0);    {Add MSFT overlay Line chart}
end;
```

AddStudy

AddStudyOnStudy

SYNTAX: **AddStudy**(*Study* [, *Tab*, *Parameter1*, *Parameter2*, *Parameter3*, *Offset*, *Shift*, *Location*, *DataPoint*, *Alignment*, *Grid*: real]): integer;
AddStudyOnStudy(*Study* [, *Tab*, *Parameter1*, *Parameter2*, *Parameter3*, *Offset*, *Shift*, *Location*, *OnStudy*, *Value*, *Alignment*, *Grid*: real]): integer;

The following syntax is used for adding the 'Color Band' Study.

AddStudy(eBnd [, *Tab*, 0, 0, 0, 0, *ESPL*, *Location*, *Position*, *Study*: real, *CloseOnly*: boolean]): integer;

DESCRIPTION:

AddStudy adds a Study to a chart and sets the study parameters. If the parameters are omitted then the settings from the specified *Tab* are used when the study is drawn. Use the **GetStudy** and **SetStudy** commands to get and set the study parameters, colors, and line styles after the study has been applied to a chart. Since there may be multiple Chart windows open at the same time, it is important to first identify the Chart that the study will be placed on. A global ESPL variable named *Window* is used to specify which chart to place the study on. The *Window* variable can be manually assigned a window pointer value (if you have been keeping track of the window handles), or you can use the **FindWindow** or **Chart** functions to set the *Window* variable.

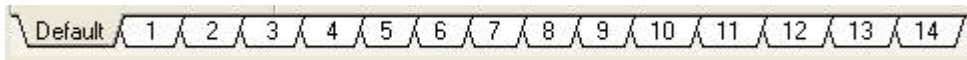
AddStudyOnStudy adds a technical study to a chart based on another study's values. For example, a Moving Average of a Relative Strength Index can be added to a chart. This assumes that the Relative Strength Index study is already present on the chart. The settings from the specified *Tab* are used when the study is drawn.

Both the **AddStudy** and **AddStudyOnStudy** commands return the Object ID number (handle) of the added study. The object number can be used by the **GetStudy**, **SetStudy**, **ChartRefresh** and **Remove** functions to perform further operations on the study. The functions both return a zero value if adding the study was unsuccessful.

PARAMETERS:

Study: Use one of the following predefined ESPL constants to specify which study to add to the chart. See [Appendix – Study Constants](#).

Tab: Specifies the *Tab* setting to use from the Study's properties window. Enter a number from 0 to 14. For example, an entry of 0 will draw the Study using all the colors, styles, and marker settings from the 'Default' tab.



Parameter1: Usually specifies the number of bars to use in the study calculation.

Parameter2: Usually specifies the optional moving average value of the study.

Parameter3: Specifies a 3rd study parameter for some studies. :

Offset: Specifies an optional vertical price offset for the study. It is also used to specify the *ESPL* variable value when running ESPL Studies. Example: `AddStudy(eESPL, 1, 5, 1, 1, 100, 0, eChart)` will run an ESPL Study with the *ESPL* variable value set to 100. NOTE: Typical *ESPL* values would be in the range of 100-109.

Shift: Specifies the number of bars to optionally shift a study left or right on the chart. Example, a value of 5 would shift the study line forward 5 bars. A value of -5 would shift the study line backwards 5 bars.

Location:
0 or eChart - Draws the Study directly on the Chart (default).
1-9 - Draws the Study in the specified study sub-window 1 through 9.
eVolume - Draws the Study in study sub-window 9 (the Volume window).

DataPoint: This parameter is used by the **AddStudy** command. The *DataPoint* parameter specifies the Data Point to use for the study. Example: 0=Close, 1=High, 2=Low, etc. View the 'Data Point' panel in a study's Properties window for a list of available Data Points. The position of the Data Point in the list determines what number to use (starting with 0). The Data Points are different for some studies. The following example shows the Data Point values for the RSI study (see the Properties window for the RSI study). See [Appendix – Data Points](#)

NOTE: A *DataPoint* value of 2 would specify that an RSI study be added to the chart, using the Low price of each bar as the input values for the RSI study. The values to use for the *DataPoint* parameter should be from 0 to 34 based on the position of the Data Point for each study.

- Alignment:* Specifies an optional Spread Alignment adjustment. Shifting the lines prevents the Spread histogram from drawing exactly on top of the chart bars.
0 = Align to left side, 1 = Align to center, 2 = Align to right side.
- Grid:* Specifies and draws grid lines in the study sub-window. The *Grid* parameter is ignored if the study is not drawn in a study sub-window.
- OnStudy:* This parameter is used by the **AddStudyOnStudy** command. The *OnStudy* parameter specifies the Object ID (handle) number of the original study to which the new study is calculated from. The default *OnStudy* value is the object number of the last study added to the chart. NOTE: This is a case in which you should store the object numbers of studies, so that subsequent studies can be performed on them. You can also use the **FindStudy** command to determine a previous studies object number.
- Value:* The *Value* parameter is used by the **AddStudyOnStudy** command and can be entered as a 0,1, 2, or 3. When adding a study on another study, it may be necessary to specify which value (of the original study) to use as input for the new study. The *Value* specifies which of four possible study values to use for the **AddStudyOnStudy** function. For example, 0=Study value, 1=Study Average value, 2= Spread between the two, etc.. The Bollinger Band study draws 3 lines and an optional Spread line. Any of the 4 lines can be used as input for a new study. The possible *Value* parameters from the Bollinger Band study would be 0=Upper Band, 1=Middle Average line, 2=Lower Band, 3=Spread. The default is 0.

EXAMPLE: The following example code adds a Bollinger Band study to a presumed chart, then adds a Stochastic study using the Lower Bollinger Band values as input for the Stochastics study. The Stochastic study parameters are set to 9, 3, and 5. This will generate a 9 period slow Stochastic study of the Lower Bollinger Band.

```
H := AddStudy(eBol); {Add Bollinger Band study}
AddStudyOnStudy(eSto,1,9,3,5,0,0,0,H,2); {Add Stochastic on Lower Band}
```

Used by the Color Band Study Only:

- Position:* The *Position* parameter specifies the Marker position for the Color Band study. Enter a number from 0-14.
- Study:* The *Study* parameter specifies the Color Bar study to apply to the Color Band study. Enter a number from 1-24 based on the position in the list. Or, select *Study* 0 to run an ESPL program on the Color Band study. This allows you to apply the Color Band Markers based on ESPL logic.
- CloseOnly:* This parameter can place a check mark in the 'Close Only' check box. Enter a value of 'False' to uncheck the box. Enter a value of 'True' to place a check mark in the box.
Example: AddStudy(eBnd, 1, 0, 0, 0, 0, 61, 0, 7, 0, True);

NOTE: See the Color Band ESPL program example in the **GetStudy** documentation.

EXAMPLE: The following code tries to locate a currently displayed IBM chart. If it can't find an open IBM chart, then it opens an IBM daily chart. After finding or opening the IBM chart, the program removes all studies from the chart, then adds a Momentum study, an RSI study, and a Stochastics study (based on the RSI study).

```
var {Start of variable declarations}
  Handle:integer; {Handle declared as an integer variable}
begin {Start of Main programming code}
  if FindWindow(eChart,'IBM.D')=0 then Chart('IBM.D');
```

```

Remove (eStudy);           {Remove all studies from the chart}
AddStudy (eMOM, 1);       {Add Momentum study to the chart}
Handle := AddStudy (eRSI, 1, 7, 1, 0, 0, 2, 2, 0, 1, 3);   {Add RSI}
AddStudyOnStudy (eSto, 1, 9, 3, 3, 0, 0, 1, Handle, 0, 1, 33288);
end;                       {End of program}

```

Alert GetAlert

SYNTAX: **Alert** (There are 8 variations of the Alert command, see below)
 GetAlert(*Symbol*: string, *Field*: integer): real;

- 1) **Alert**(*Visible*: boolean [, *Message*: string, *PanelColor*: integer, *FontColor*: integer, *Beep*: boolean, *FontSize*: integer]);
- 2) **Alert**(*Visible*: boolean [, *Message*: string, *PanelColor*: integer, *FontColor*: integer, *FileName*: string, *FontSize*: integer]);
- 3) **Alert**(eSet; *Price*: real [, *Field*: integer]);
- 4) **Alert**(eClear);
- 5) **Alert**(eHigh);
- 6) **Alert**(eLow);
- 7) **Alert**(*Symbol*: string, *Feed*: integer, *Price*: real, *Field*: integer);
- 8) **Alert**(eReset);

DESCRIPTION: The **GetAlert** function returns the price of a specific alert field. If no alert price is present for the specified field, then zero will be returned. The **Alert** statement is used to set and remove alerts on Charts and Quote Symbols. Alert variations 1 and 2 (above) display or remove an Alert panel on the top row of a chart. A message can be printed in the Alert panel. A Beep or sound file can also be played when the Alert panel is displayed. The Alert panel is often used when an alert condition has been met in the ESPL program. The Alert panel can be displayed on the chart, along with a sound. The color of the panel and the text in the panel can be customized. Any .WAV file can be played as the alert sound.

```
Alert(True, 'Sell Alert', clRed, clWhite, 'C:\WINDOWS\MEDIA\CHIMES.WAV');
```

Alert variations 3 through 6 are used to set and clear price alerts from a Chart. Some examples are shown below.

```

Alert(eSet, 9800);           Sets a price alert on a chart at 9800. The price can be above or below the current price.
Alert(eSet, 75000, 7);      Sets a Daily Volume alert for the chart symbol.
Alert(eClear);             Removes all Alerts for this chart symbol
Alert(eHigh);              Removes just the High price alert from a chart.
Alert(eLow);               Removes just the Low price alert from a chart.

```

Alert variation 7 is used to set or remove an alert on a specified Symbol. The Symbol and Market Group must be specified. The Alert will be added to or removed from the Ensign Alerts list. The *Field* parameter specifies the type of alert that is being set.

```

Alert('IBM', FindFeed('IBM'), 12200, eHigh);   Sets a High alert for IBM at $122 dollars.
Alert('F', FindFeed('F'), 2800, 2);           Sets a Bid high alert for Ford at $28 dollars.
Alert('XOM', FindFeed('XOM'), 5000, 6);       Sets a Tick Volume alert for XOM at 5000 shares.
Alert('GE', FindFeed('GE'), 0, eNone);        Removes all Alerts for GE

```

Alert variation 8 is used to completely clear and delete all price alerts. The Alerts list will be cleared of all alerts.

```
Alert(eReset);
```

PARAMETERS:

Visible: Enter as True to display the Alert panel. Enter as False to hide the Alert panel.

Message: *Message* is the text that will be displayed in the Alert panel (example: 'Sell Alert').

PanelColor: *PanelColor* specifies the color of the Alert panel (example: c1Red).

FontColor: *FontColor* specifies the color of the Alert panel text message (example: c1White).

Beep: Enter as True to sound a beep. Enter as False for no beep or sound.

FileName: *FileName* specifies the sound file to play. The *FileName* parameter should include the complete path and filename of the sound file that you want to play. The file should be a .WAV, .MID, or .RMI file type. If a *Filename* is not provided, then the **Alert** statement will attempt to play 'C:\WINDOWS\MEDIA\CHORD.WAV' as a default sound. If the *FileName* cannot be located on the hard disk, then a Beep will sound.

FontSize: *FontSize* specifies the size of the text in the Alert panel. The default *FontSize* is 9. Acceptable *FontSizes* range from 8 to 16.

Price: *Price* is the alert value. Note: When setting an alert on a chart, if the specified alert price is above the current Last price then a High alert will be set. If the specified alert price is below the current Last price then a Low alert will be set.

Symbol: Specifies the Symbol to set or get the alert.

Market: Market is number that represents a Symbol's Market Group. The Market Group for a Symbol can be retrieved using the **FindMarket** function.

Field:

- 0 or eHigh = Sets a High alert for the Last price
- 1 or eLow = Sets a Low alert for the Last price
- 2 = Sets a High alert for the Bid price
- 3 = Sets a Low alert for the Bid price
- 4 = Sets a High alert for the Ask price
- 5 = Sets a Low alert for the Ask price
- 6 or eTickVolume = Sets a Tick Volume alert
- 7 or eVolume = Sets a Daily Volume alert
- 8 or eNone = Removes an alert from the Alert List

EXAMPLE: The following example opens an IBM daily chart. All the price alerts are removed from the chart, then an Alert Panel is displayed on the chart, and the CHIMES.WAV file is played. The Alert panel is removed from the chart after 10 seconds. Next, a new price alert is added to the chart, equal to the High price of the last bar on the chart (BarEnd). Lastly, a High price alert is set at \$58.00 for the MSFT symbol. The **GetAlert** function is used to retrieve and print an alert field.

```
begin
  Chart('IBM.D');           {Open an IBM daily chart}
  Alert(eClear);           {Remove all price Alerts from the chart}
  Alert(True, 'Place Sell Order Now', c1Red, c1White,
    'C:\WINDOWS\MEDIA\CHIMES.WAV');
  Pause(10);              {Pause the ESPL program for 10 seconds}
  Alert(False);           {False removes the Alert panel from the chart}
  Alert(eSet, High(BarEnd)); {Set a new price Alert on the Chart}
  Alert('MSFT', FindMarket('MSFT'), 5800, eHigh);
  writeLn(GetAlert('MSFT', eHigh));
end;
```

AlertEvent

SYNTAX: **AlertEvent**(*Event*: constant [, *ESPL*: integer, *Symbol*: string]);

DESCRIPTION: The **AlertEvent** statement is used to enable or disable the monitoring of specific events. If the event occurs, a call is made to the ESPL program with the *ESPL* variable set to a specified value. Since the event is identified by the *ESPL* value, this allows an ESPL program to perform specific tasks whenever the event occurs. For example, an event can be enabled for when the market price crosses an alarmed trend line. Mouse movement over a chart is another event that can be enabled with the **AlertEvent** command. Information about the event is passed to the global *IT* string variable.

PARAMETERS:

Event : The *Event* parameter can be one of the following predefined event constants. This parameter is used to enable Ensign to monitor the specific event. If the event occurs, then a call is made to the ESPL program.

eAlerts Enables an event for Alert Objects placed on a chart to test study conditions.
eClear Disables all Alert events.
eLine Enables an event for alarmed trend lines. The event is triggered whenever the price crosses the line.
eNews Enables an event for News story Alerts. The event is triggered when a news alert occurs.
eQuote Enables an event for Alerts set for prices, bids, asks, tick volumes, and volume levels.

eSave Enables an event for when a Chart saves its data file prior to the chart closing or changing. This event can be triggered from many locations in the program. The content of the *IT* global variable can be checked to differentiate why the chart file is being saved. The *IT* text will have the word Save followed by one of these words: Roll, Build, Rebuild, Objects, Load, Open, Close, Template, Property, Timer, Time, Refresh, Update, ESPL 1, ESPL 2, ESPL 3. Example test script:

```
begin
  if ESPL=0 then AlertEvent (eSave,600);
  if ESPL=600 then writeln( IT );
end;
```

This test script writes the *IT* variable content in the Output window. For example, the script posted this when the EUR A0-FX.5 minute chart was closed: 10:55:59 – Save Close : EUR A0-FX.5

eLoad Enables an event for when a Chart opens or changes and its data file has been loaded.

eTrade Enables a Trade event for Symbols in the Ensign Alerts list. An AlertEvent will trigger with every Trade (change in the Last price). The event can be enabled for a single symbol, or for all symbols in the Alerts list. A symbol must be entered in the Alerts list before an eTrade event can be enabled for that symbol. Any value from 0 to 255 can be entered as the *ESPL* value. Examples:

```
AlertEvent(eTrade,70,'IBM'); // Enables a Trade event for all IBM trades, ESPL=70
AlertEvent(eTrade,0,'IBM'); // Removes the Trade event from IBM
AlertEvent(eTrade,85); // Enables a Trade event for all symbols, ESPL=85
AlertEvent(eTrade,0); // Clears Trade events for all symbols
```

eTick Enables a Trade event for a Symbol. This is just like eTrade except the symbol does not have to be in the Alerts list. Any value from 0 to 255 can be entered as the *ESPL* value. Examples:

```
AlertEvent(eTick, 70,'IBM'); // Enables an event for all IBM trades, ESPL=70
AlertEvent(eTick ,0,'IBM'); // Removes the event from IBM
AlertEvent(eTick ,0); // Removes all symbols from this event list
```

eMouse Enables an event for mouse movement over charts. Two global variables (*PtX* and *PtY*) are updated with the mouse X Y coordinates as the mouse moves over any chart. The global *Window* variable is also set to the chart which the mouse passes over. This allows you to read the position of the mouse on a chart and then determine the chart values. The chart filename is also passed to the *IT* variable.

ESPL: The *ESPL* value is used to identify an event. If the event occurs, the ESPL program is called with *ESPL* set equal to the value specified when the event was enabled. Example, **AlertEvent**(eNews, 77) will

cause *ESPL* to equal 77 whenever an Ensign News alert occurs. This allows the ESPL program to test for a *ESPL* value of 77 and respond to News alerts with the desired programming. NOTE: Assigning *ESPL* a value of zero will disable a specific event (example: **AlertEvent**(eNews, 0)). All alert events are disabled when editing is done on a script, or when a new ESPL script is loaded into the script editor. When using the **AlertEvent** statement, you can set the *ESPL* parameter value to any number of your choosing.

Symbol: The *Symbol* parameter is used by the eTrade event. Individual symbols in the Alerts list can be enabled to trigger an **AlertEvent** when the symbols trade. For example, **AlertEvent**(eTrade,90,'IBM'); will cause the program to call *ESPL*=90 for every IBM trade that occurs. The *IT* variable will contain the Symbol that traded.

EXAMPLE: The following program enables an Event for tracking mouse movement over charts. The program sets *ESPL* equal to 66 whenever the mouse event occurs. Click ESPL button 2 to enable the mouse event. Click ESPL button 3 to turn-off the event. When enabled, the 'ShowPrices' procedure (*ESPL*=66) is called when the mouse is moved over a chart. The chart name and the bar values of the High, Low, and Last are printed in a TextBox for the current mouse position.

```

procedure ShowPrices;                                var
{Start of variable declarations}
  i,SC,Mkt: integer;
  rLast, rHigh, rLow: real;
begin                                                {Start of ShowPrices procedure
programming}
  i := XtoIndex(PtX);                                {Convert Mouse X coordinate to Bar Index}
  rHigh := High(i);
  rLast := Last(i);
rLow := Low(i);
  SC := GetVariable(eScaleFactor);                    {Get the Scale Factor for the chart}
  Mkt:= GetVariable(eMarket);                          {Get the Market Group for the chart symbol}
  if FindWindow(eText)=0 then TextBox(' ',1,1,50,110); {Create Text Box}
  TextClear;                                          {Clear the Text Box}
  TextAdd(IT,False);                                  {Print Chart name in Text Box}
  TextAdd('Last '+FormatPrice(rLast,SC),False);
  TextAdd('High '+FormatPrice(rHigh,SC),False);
  TextAdd('Low '+FormatPrice(rLow,SC),False);
end;                                                  {End of Showprices procedure}

begin                                                {Start of Main Programming Code}
  if ESPL=2 then AlertEvent(eMouse,66);              {Enable Mouse Event with ESPL=66}
  if ESPL=3 then AlertEvent(eMouse,0);               {Turn-off the Mouse Event}
  if ESPL=66 then ShowPrices;                         {Call ShowPrices if 66 occurs}
end;                                                  {End of program}

```

Align

SYNTAX: **Align**(*Value*: variant [, *Width*: integer, *Alignment*: integer]): string;

DESCRIPTION: The **Align** function is used to format the alignment of a number or string. The function is passed a value which is then returned as a formatted string. The string can be aligned-left, aligned-right, or aligned-center. The width of the string can also be specified. The **Align** function is particularly useful for outputting numbers in a report or in columns.

PARAMETERS:

Value: The *Value* can be any number or string (example: 23.56). NOTE: Real variable types will be converted to show two decimal places. Boolean variable types will be converted to the words 'True' or 'False'.

Width: Enter a number to specify the width of the returned string. The default width is 10.

Alignment: Specifies the alignment. Enter one of the following predefined constants: `eCenter`, `eLeft`, or `eRight`. The default alignment is `eRight`.

EXAMPLE: The following example displays a chart and then adds a Stochastics study to the chart. The Stochastic values for the last 10 bars are then retrieved, Aligned, and printed in the output window. The alignment of the Stochastic numbers with a width of 6 makes the report more readable.

```
var
    S1, X : integer;           {S1 and X declared as integer variables}
begin
    {Start of Main programming code}
    Chart('IBM.D');           {Open an IBM daily chart}
    S1 := AddStudy(eSto);      {Add the Stochastic study to the chart}
    writeln('Stochastic Values'); {Print text to the output window}
    for X := BarEnd-10 to BarEnd do {Loop through the last 10 bars}
    begin
        {Start of the 'for' loop Statements}
        write('Date= ', Bar(eDate, X)); {Print the Date of each bar}
        write(' %K=', Align(GetStudy(S1, 2, X), 6));
        writeln(' %D=', Align(GetStudy(S1, 3, X), 6));
    end;
    {End of the 'for' loop Statements}
end;
    {End of program}
```

And

SYNTAX: **And**

DESCRIPTION: **And** is a logical operator that is often used with Boolean (True and False) conditions like the **if ...then** statement.

EXAMPLE: The following example determines if two conditions are True. If both conditions are True, then it will Beep.

```
var
    {Start of variable declarations}
    X, Y : integer;           {X and Y are declared as integers}
begin
    {Start of Main programming code}
    X := 5;                   {X is assigned the value of 5}
    Y := 7;                   {Y is assigned the value of 7}
    if (X<Y) and (X=5) then Beep; {If both conditions are True then Beep}
end;
    {End of program}
```

Application

SYNTAX: **Application**

DESCRIPTION: The **Application** program statement is used to read or set various Properties of the Ensign program. The statement can be used to retrieve the Ensign program file name, Handle, Help file name, and to control Hints. The *ProcessMessages* method is used to interrupt an ESPL program, so that Windows can respond to other system or program events. This may be necessary if an ESPL program is very lengthy and is keeping Windows from processing other events in its multi-tasking environment (hogging the CPU).

PROPERTIES: Properties can be read or set by appending the Property name after the **Application** statement.

ExeName: The Ensign file name and path information.
HelpFile: The name of the Help file that Ensign uses.
Handle: The Handle identifies Ensign. The program Handle for Ensign.
HintColor: The HintColor used by the Hint help boxes in Ensign.
HintHidePause: The time (in milliseconds) before hiding a Hint when the mouse still points to an item.
HintPause: The time before a Hint box appears when the mouse points to a control or menu item.
HintShortPause: The time to wait before bringing up a Hint if a hint has already been shown.
ShowHint: Specifies whether Hints are enabled or disabled. Set *ShowHint* to True or False.
Title: The text that appears with the Ensign icon in the Windows task bar.

METHOD: A method is a function that occurs when the method is invoked. *ProcessMessages* is the only available method for the **Application** statement.

ProcessMessages: Use *ProcessMessages* to permit Windows to process other events at the time *ProcessMessages* is called. The *ProcessMessages* method cycles the Windows message loop until it is empty and then returns control to the ESPL program. In lengthy ESPL programs, calling *ProcessMessages* periodically will allow Windows to respond to other system messages such as processing the data-feed or updating charts. Note: Do not use *ProcessMessages* if you are using the *eSignal* data-feed. The command does not work with the *eSignal* version of Ensign.

EXAMPLE: This program demonstrates the various Properties and the Method of the **Application** statement. After retrieving and setting various properties, a lengthy print loop is started. The **Application.ProcessMessages** statement allows Windows to process other system messages between each print, so that the computer will not be locked-up until the program completes.

```

var                                     {Start of variable declarations}
  j: integer;                           {j declared as an integer variable}
  ProgramName: string;                  {ProgramName declared as a string}
  HelpFile: string;                     {HelpFile declared as a string}
  ProgramHandle: THandle;                {ProgramHandle declared as a THandle}
begin                                    {Start of Main programming code}
  Output(eClear);
  ProgramName:= Application.ExeName;     {Retrieve the Ensign filename/path}
  HelpFile:= Application.Helpfile;      {Retrieve the HelpFile name}
  ProgramHandle:= Application.Handle;    {Retrieve the Ensign Handle}
  Application.HintColor:= clLtGreen;    {Set the Hint box colors to Green}
  Application.HintHidePause:= 5000;
  Application.HintPause:= 1000;
  Application.ShowHint:= True;          {Enables Hints in Ensign}
  Application.Title:= 'My Ensign';      {Change the Ensign Icon text}
  for j:=1 to 1500 do begin              {Start a lengthy 'for' loop}
    writeln('Loop Count = ',j);         {Print the value of the loop counter}
    Application.ProcessMessages;
  end;                                    {End of 'for' loop statements}
end;                                     {End of program}

```

Arc

Chord

Ellipse

Pie

SYNTAX: **Arc**(x1, y1, x2, y2, x3, y3, x4, y4: integer);
Chord(x1, y1, x2, y2, x3, y3, x4, y4: integer);

```

Ellipse(x1, y1, x2, y2: integer);
Pie(x1, y1, x2, y2, x3, y3, x4, y4: integer);

```

DESCRIPTION: The four drawing statements above are based on the ability to calculate and draw an Ellipse on a chart. The **Arc** statement draws a specified portion of an Ellipse. The **Chord** statement is the same as the **Arc** statement except the ends of the **Arc** are connected with a straight line. The **Pie** statement draws a specified pie piece of an Ellipse, with lines drawn from the center of the Ellipse to connect each end, thus forming a Pie shape. NOTE: A Circle can be drawn with the **Ellipse** statement. The drawings on the chart are not permanent and remembered. The ESPL program must redraw the shapes whenever necessary. The drawings will disappear when the chart is refreshed, moved, redrawn, etc.

The **Ellipse** statement draws an Ellipse on a Chart. The shape and placement of the Ellipse on the chart are determined by screen pixel coordinates. Screen pixel coordinates start in the top left corner of the chart window (0, 0). X-coordinates specify horizontal pixels moving to the right. Y-coordinates specify vertical pixels moving down. The Ellipse is calculated by specifying a rectangle boundary (x1, y1, x2, y2). The top-left point of the bounding rectangle is at pixel coordinates (x1, y1) and the bottom-right point is at (x2, y2). An Ellipse is calculated to fit within the rectangle boundary. NOTE: If the rectangle points form a square, then the **Ellipse** statement will draw a perfect Circle.

The **Arc** statement draws an arc line on a Chart. The Arc is a specified portion of an Ellipse. The shape and placement of the Arc on the chart are determined by screen pixel coordinates. The Arc is calculated by first specifying a rectangle boundary (x1, y1, x2, y2), similar to the **Ellipse** statement. The **Arc** statement only displays the portion of the Ellipse specified by the (x3, y3, x4, y4) coordinates. An imaginary line is calculated from the center point of the Ellipse to the (x3, y3) point. The Arc will start drawing from the intersection of this line, and then draw counterclockwise until it reaches the intersection of the (x4, y4) point from the center point.

The **Chord** statement is exactly the same as the **Arc** statement, except the end points of the Arc are connected by a straight line. For example, a Chord could be drawn that appears similar to a half-moon shape.

The **Pie** statement is also the same as the **Arc** statement, except the end points of the Arc are connected by two lines that extend from the exact center point of the Ellipse, thus forming a Pie shape.

All of the statements will draw on the chart that is currently referenced by the global ESPL *Window* variable as set by the **FindWindow** or **Chart** functions.

EXAMPLE: The following program draws an Arc, a Rectangle, two Ellipses, a Pie, and a Chord. Different line and color affects are created by changing the Brush and Pen properties. A Rectangle box is drawn using the same screen coordinates as the example Arc that is drawn. The Rectangle illustrates how each of the drawing functions are contained within a bounding rectangle. Click ESPL RUN button 11 and then drag the mouse on the chart to 3 random points on the chart. The shapes will draw at the X,Y locations specified in the programming code.

```

uses
  Graphics;
procedure DrawShapes
  SetPen(clBlue, 2, eSolid);      {Pen color=Blue, 2 pixel width, Style=Solid}
  Arc(50,10,250,110,50,60,250,10);  {Draw an Arc (blue, solid line)}

  SetPen(clWhite, 1, eDot);      {Pen color=White, 1 pixel width, Style=Dotted}
  SetBrush(clWhite, eClear);     {Brush fill color=White, fill Style=Clear}
  Rectangle(50,10,250,110);      {Draw Rectangle (white, dotted lines)}

  SetPen(clYellow, 2, eSolid);   {Pen color=Yellow, 2 pixel width, Style=Solid}
  Ellipse(50,110,250,210);      {Draw an Ellipse (yellow, solid line)}
  SetPen(clWhite, 1, eDot);     {Pen color=White, 1 pixel width, Style=Dotted}
  Ellipse(100,110,200,210);     {Draw an Ellipse, (a white, dotted Circle)}

  SetBrush(clRed, eHorizontal);  {Brush fill color=Red, fill style=Hor. lines}
  Pie(200,10,400,110,400,110,400,10);{Draw a Pie (Horizontal red line fill)}

```

```

SetPen(clRed, 3, eDot);      {Pen color=Red, 3 pixel width, Style=Dotted}
SetBrush(clWhite, eVertical);{Brush fill color=White,fill style=Vert lines}
Chord(300,100,600,200,500,100,400,200); {Draw a Chord Vertical filling}
end;

{*****Main Programming Code*****}
begin
  if ESPL=11 then DrawShapes;
end;

```

ArcCos

ArcSin

ArcTan

Cos

CoTan

Sin

Tan

SYNTAX: **ArcCos** (X: real): real;
 ArcSin (X: real): real;
 ArcTan(X: real): real;
 Cos(X: real): real;
 CoTan(X: real): real;
 Sin(X: real): real;
 Tan(X: real): real;

DESCRIPTION: These functions can be used to calculate various trigonometric values.

ArcCos The return value is in the range [0..Pi], in radians. X must be between -1 and 1.
 ArcCos returns the inverse cosine of X.

ArcSin X must be between -1 and 1. The return value will be in the range [-Pi/2..Pi/2], in radians.
 ArcSin returns the inverse sine of X.

ArcTan ArcTan returns the arctangent of X. X is a real expression that represents an angle in radians.

Cos Cos returns the cosine of the angle X. X is a real expression that represents an angle in radians.

CoTan Cotan returns the cotangent of X. The cotangent is calculated using the formula $1 / \tan(X)$. Note: Do not call Cotan with X = 0.

Sin Sin returns the sine of the angle X in radians.

Tan Tan returns the tangent of X. $\tan(X) = \sin(X) / \cos(X)$.

EXAMPLE: The following program calculates and prints the Tangent of 2.5

```

var                               {Start of variable declarations}
  Value: real;                    {Value is declared as a Real variable type}
begin                             {Start of Main programming code}
  Value:= Tan(2.5);               {Value is assigned the Tangent of 2.5}
  writeln(Value);                 {Value is printed to the output window}
end;                               {End of program}

```

Arrays

DESCRIPTION: Use *Arrays* to store and manipulate groups of numbers. An Array is an ordered collection of numbers, with each number referenced by its position in the Array. When an Array is created, the elements are all set to zero values. You can fill the Array elements with any number and manipulate the data as needed. The size of the Array will be the greater of 2100 or the number of bars on the currently displayed chart.

Arrays must be declared and created at the start of the program, and then freed at the end of the program. Failing to free an Array variable after using the program will cause a memory leak (the computer memory is not released). Several Properties and Functions can be used with an Array to add, delete, and retrieve numbers at specified positions in the array, rearrange the array elements, sort the array elements in ascending or descending order, and calculate math functions on the array data.

NOTE: If you don't want to declare, create, and free your own Array, Ensign provides one predefined array named **vArray**. This Array is a global ESPL array that is automatically declared, created, and freed by Ensign (see the documentation for **vArray**). If you need to use multiple arrays, then you will have to create and use your own arrays.

Declaring and Creating an Array

Before an Array can be used it must be declared as an TArray variable type, and then created by using the *Create* command. Append the *Create* command after the **TArray** statement to create the array (example: **TArray.Create**). The following program declares an array named *TestArray*. *TestArray* is then created, filled with random numbers, sorted, printed, and then freed at the end. This is a good example of using an Array to hold and manipulate a group of numbers.

```
var                                     {Start of variable declarations}
  TestArray: TArray;                   {TestArray is declared as a TArray}
  X: integer;                           {X is declared as an integer}
begin                                   {Start of Main programming code}
  TestArray:=TArray.Create;             {TestArray is Created}
  for X:=1 to 10 do TestArray.Values[X]:=Random(100); {TestArray filled}
  TestArray.Sort(10,10,True);           {TestArray is sorted}
  for X:=1 to 10 do writeln(TestArray.Values[X]);   {TestArray is printed}
  TestArray.Free;                       {TestArray is freed}
end;                                     {End of program}
```

PROPERTIES, FUNCTIONS, AND METHODS: The following list of commands can be used to control arrays. The command is appended to the Array name to obtain the desired action (example, *TestArray.Values[5]*).

<i>Values</i>	References a specified position in the Array. Used to set and retrieve values from the Array.
<i>Average</i>	Returns the simple moving average of a range of Array elements.
<i>ExpAverage</i>	Returns an exponential smoothed average for a range of Array elements.
<i>Highest</i>	Returns the highest number and its Position in a range of Array elements.
<i>Lowest</i>	Returns the lowest number and its Position in a range of elements.
<i>Regression</i>	Returns the linear regression line value for a range of elements.
<i>StdDev</i>	Returns the standard deviation for a range of elements.
<i>Summation</i>	Returns the sum of the values for a range of elements.

<i>Create</i>	Creates and initializes an Array.
<i>Delete</i>	Deletes a specified element from the Array.
<i>Exchange</i>	Swaps the position of two elements in the Array.
<i>Free</i>	Destroys an Array and frees its associated computer memory.
<i>IndexOf</i>	Returns the Position of a specified number of the array.
<i>Insert</i>	Inserts a number at a specified Position.
<i>Plot</i>	Opens a chart and plots the Array values on the chart.
<i>Sort</i>	Sorts the array elements in ascending or descending order.

SYNTAX: The following syntax is used with each Array command.

Values: `ArrayName.Values[Position: Integer]: real;`
The *Values* property is used to store and retrieve a number from the specified position in the Array. Square brackets and a position number are used to specify the *Position* in the array. Use the *Values* command to read or modify the number at the specified position. *Position* specifies the position of the number in the Array, where 0 is the position of the first number, 1 is the position of the second number, and so on.
 Example: `TestArray.Values[5]` references the 6th element in TestArray.

For the following seven functions, the default for *Position* is 2100. The default for *Count* is *Position*. Parentheses are used to enclose the *Position* and *Count* values.

Average: `ArrayName.Average([Position: integer, Count: integer]): real;`
Average returns a simple moving average value for *Count* number of elements ending at the specified *Position*. Example: `TestArray.Average(19, 20)` returns the average of the first 20 array values. Remember that the count starts at zero, so 19 is the 20th position in the array.

ExpAverage: `ArrayName.ExpAverage([Position: integer, Period: integer [, Start: integer]): real;`
The *ExpAverage* function returns the exponential smoothed average from *Start* through *Position*. If *Start* is omitted, then the default for *Start* is $Position - Period + 1$.
The exponential smoothing factor = $2 / (Period + 1)$.
ExpAverage is initialized with `ArrayValue[Start]`.
 $ExpAverage = ArrayValue * factor + (1-factor) * previous\ ExpAverage$.
Example: `TestArray.ExpAverage(50, 10)` returns a 10 period ave ending at position 50.

Highest: `ArrayName.Highest(Position, Count, var ArrayIndex, Rank: integer): real;`
Highest returns the highest number from *Count* number of elements ending at the specified *Position*. *ArrayIndex* is an integer value that is returned from this command. It contains the array position where the highest number occurred. If two array elements contain the same high value, *ArrayIndex* will return the position of the first occurrence (closest to position 0). *Rank* can be entered as a value from 1 to 5. The *Rank* specifies that the return value be the 1st, 2nd, 3rd, 4th, or 5th highest value in the specified array elements.
Example: `TestArray.Highest(20, 10, X, 2)` returns the 2nd highest value in elements 11 to 20.

Lowest: `ArrayName.Lowest(Position, Count, var ArrayIndex, Rank: integer): real;`
Lowest returns the lowest number from *Count* number of elements ending at the specified *Position*. *ArrayIndex* is an integer value that is returned from this command. It contains the array position where the lowest number occurred. If two array elements contain the same low value, *ArrayIndex* will return the position of the first occurrence (closest to position 0). *Rank* can be entered as a value from 1 to 5. The *Rank* specifies that the return vale be the 1st, 2nd, 3rd, 4th, or 5th lowest value in the specified array elements.
Example: `TestArray.Lowest(20, 10, X, 1)` returns the lowest value in elements 11 to 20.

Regression: `ArrayName.Regression(Position, Count: integer, var Slope: real, var StdError: real): real;`
A simple linear regression line is determined using *Count* number of elements ending at *Position*. *Regression* returns the value of the linear regression line at *Position*. The slope of the linear regression line is returned in the *Slope* variable, and the Standard Error of Estimate is returned in the *StdError* variable. These variables should be declared before calling this command.
Example: `TestArray.Regression(20, 10, S, SE)` returns the linear regression value from elements 11 through 20. The slope is returned in *S*, and the standard error in returned in *SE*.

StdDev: `ArrayName.StdDev([Position, Count: integer, Flag: boolean]): real;`
StdDev returns a standard deviation for *Count* number of elements ending at *Position*. When *Flag* is True, it uses the formula for a specific population, and when False, it uses the formula for a sample population. The default is for a specific population. Example: `TestArray.StdDev(20, 10, True)` returns the Standard Deviation of array elements 11 through 20.

Summation: ArrayName.*Summation*([*Position*, *Count*: integer]): real;
Summation returns the sum of *Count* number of elements ending at *Position*.
Example: TestArray.*Summation*(20, 21) returns the sum of all the elements from 0 to 20.

Create: ArrayName := TArray.*Create*; Call TArray.*Create* to create an array of real. The size of the array is fixed and cannot be redimensioned. All elements are initialized to a value of zero. Trying to use an array before it is created will cause an Access Violation error. The ArrayName should be declared before creating it.

```

var                               {Start of variable declarations}
  TestArray: TArray;              {TestArray is declared as a TArray}
begin                             {Start of Main programming code}
  TestArray:=TArray.Create;      {TestArray is Created}

```

Delete: ArrayName.*Delete*(*Position*: Integer);
Delete removes a specified number from the array. Subsequent elements in the array are moved up to fill in the number deleted at position Index.
Example: TestArray.*Delete*(5); {Deletes array element at Position 5}

Exchange: ArrayName.*Exchange*(*Position1*, *Position2*: Integer);
Call *Exchange* to exchange the position of two numbers in the array. The numbers are specified by their position values. The following example exchanges the array values in Position 5 and Position 6.
Example: TestArray.*Exchange*(5, 6);

IndexOf: ArrayName.*IndexOf*(*Number*: real): Integer;
Call *IndexOf* to locate the first occurrence of *Number* in the Array. *IndexOf* returns the Position of the number in the array. Example, if *Number* is located in the Position 5 of the array, then a 5 will be returned. If *Number* is not found in the array, then a -1 is returned. If *Number* appears in the array multiple times, then only the position of the first occurrence is returned.
Example: TestArray.*IndexOf*(750); {Locate the Position index of the number 750}

Insert: ArrayName.*Insert*(*Position*: Integer; *Number*: real);
Use *Insert* to insert *Number* into the array at the specified *Position* index. If *Position* equals 0, then *Number* is inserted at the beginning of the array. Subsequent array elements are moved down in the array to make room for the inserted number.
Example: TestArray.*Insert*(5, 750); {Inserts the number 750 at array position 5}

Plot: ArrayName.*Plot*(*FileName*: string, *Count*: integer);
Call *Plot* to open a Chart and Plot the array data on the Chart. The date for the data set will be today's date. The index is stored in the Time field for each data point. *Count* is the number of data points, with the first data point beginning at position 1. When the Chart is closed the data set will be saved to *FileName*.
Example: ArrayName.*Plot*('C:\ENSIGNHIST\ARRAY.D', 100); {Plots 100 array elements}

Sort: ArrayName.*Sort*([*Position*: Integer, *Count*: integer, *Ascending*: boolean]);
Call *Sort* to sort the numbers in an array. *Sort* will sort *Count* number of elements ending at *Position*. The default for *Count* is *Position*. *Ascending* will sort the array into ascending order when True, and descending order when False. The default is ascending order.
Example: TestArray.*Sort*(10,10,True); {TestArray is sorted}

AutoESPL

SYNTAX: **AutoESPL**: integer;

DESCRIPTION: **AutoESPL** is a global variable that can be read or set the ESPL variable value that is used for the ESPL Project Autorun feature. This value is shown on the Setup | System | ESPL & DYO form in the spinner labeled ESPL variable value.

EXAMPLE:

```
begin
  writeln(AutoESPL);
  AutoESPL := 5;
end;
```

AutoRefresh

SYNTAX: **AutoRefresh**: boolean;

DESCRIPTION: **AutoRefresh** is a global variable that can be set to a value of True or False. Use the **AutoRefresh** variable to programmatically enable or disable the 'Auto Refresh On Chart Open for:' check box on the Setup | Charts property form. If True, charts will automatically request a refresh of data when a chart opens. This is a global setting.

EXAMPLE:

```
procedure OpenAChart;
begin
  AutoRefresh := True;
  writeln(AutoRefresh);
  Chart('IBM.D');
end;

begin
  if ESPL=1 then OpenAChart;
end;
```

Ave ExpAve Sum

SYNTAX: **Ave**(n1: variant [, n2: variant n100: variant]): real;
ExpAve(alpha: variant, n1: variant [, n2: variant n99: variant]): real;
Sum(n1: variant [, n2: variant n100: variant]): variant;

DESCRIPTION: The **Ave** function returns the simple average of the parameters. $Ave = Sum/Count$. The **ExpAve** function returns the exponential smoothed average of the parameters. The exponential smoothing factor is passed as the 1st parameter. **ExpAve** is initialized with n1. $ExpAve = n * factor + (1-factor) * previous\ ExpAve$. The exponential smoothing factor equals Alpha when alpha is between 0 and 1, otherwise the exponential smoothing factor is equal to $2 / (Alpha + 1)$. The **Sum** function returns the sum of the parameters.

PARAMETERS: The parameter list may contain up to 100 numeric entries.

EXAMPLE:

```
begin
```

```

writeln(Ave(5,7,9,11,3));      {Prints      7 = (5 + 7 + 9 + 11 + 3) / 5 }
writeln(ExpAve(0.2,8,6));    {Prints 7.60 = (6 * 0.2) + (0.8 * 8)}
writeln(Sum(5,3,9,-1));     {Prints     16 = 5 + 3 + 9 + -1}
end;

```

Average

ExpAverage

Summation

SYNTAX: **Average**(Type: integer, Index: integer, Period: integer [, Dataset: integer]): real;
ExpAverage(Type: integer, Index: integer, Period: integer [, Start: integer, Dataset: integer]): real;
Summation(Type: integer, Index: integer, Period: integer [, Dataset: integer]): real;

DESCRIPTION: The **Average**, **ExpAverage**, and **Summation** commands are used to perform math calculations on chart data. Since there may be multiple Chart windows open at the same time, it is important to first identify the Chart that the functions will apply to. A global ESPL variable named *Window* is used to specify which chart to use. The *Window* variable can be manually assigned a window pointer value (if you have been keeping track of the window handles), or you can use the **FindWindow** or **Chart** functions to set the *Window* variable.

The **Average** function is used to calculate a simple average of several Chart data points. The function returns the simple average of *Period* number of chart data points which end at *Index*. Average = Summation/Period.

The **Summation** function returns the sum of *Period* number of Chart data points which end at *Index*.

The **ExpAverage** function returns the exponential smoothed average from *Start* through *Index*.

The exponential smoothing factor = $2 / (\text{Period} + 1)$.

ExpAverage is initialized with Price[Start].

ExpAverage = Price * factor + (1-factor) *previous ExpAverage.

PARAMETERS:

Type: *Type* is one of the following predefined constants:

eArray	eClose	eHigh	eLast	eLow	eMidPoint
eMid3	eMid4	eNet	eOpen	eOpenInterest	ePercent
eRange	eTrueHigh	eTrueLow	eTrueRange	eVolume	
1	2	3	4		

Refer to the **Bar** function for a complete description of these constants.

Index: *Index* is the Chart bar array position (a number between 1 and the number of bars in the chart file). Both the Host chart and optional Overlay charts use the same indexing.

Period: *Period* is the number of data points to use in the calculation. The chart data points from (*Index* - *Period* + 1) through and including (*Index*) will be used in the calculation.

Start: *Start* is only used with the **ExpAverage** command. The default for *Start* is (*Index* - *Period* + 1).

DataSet: *DataSet* is used if the chart contains another study or an overlay chart. In this case, the functions can be applied to another study, or to the overlay chart's data. The default is to use the host chart's data. To access the overlay chart's data pass the overlay's object number as the *DataSet* parameter, or pass a number 1, 2, 3, etc. (meaning the 1st, 2nd, or 3rd overlay). NOTE: The object number for studies and overlays can be obtained with the **FindStudy** command.

EXAMPLE:

```
begin
  Chart('IBM.D');
  writeln(Average(eLow, BarEnd, 10));    {Print Average of the last 10 Lows}
  writeln(Summation(eHigh, BarEnd, 20)); {Print Sum of the last 20 Highs}
  writeln(ExpAverage(eVolume, BarEnd, 20)); {Print Ave Volumes last 20 bars}
end;
```

Bar ChartBar

SYNTAX: **Bar**(*Type*, *Index*: integer [, *Dummy*, *Dataset*: integer]): variant;
Bar(*eIndex*, *Date* [, *Time*, *Dataset*: integer, *NearestMatch*: boolean, *Seconds* :integer]): integer;

ChartBar(*ChartName*: string, *Type*, *Index* [, *Dummy*, *Dataset*: integer]): variant;
ChartBar(*Window*, *Type*, *Index* [, *Dummy*, *Dataset*: integer]): variant;

ChartBar(*ChartName*: string, *eIndex*, *Date* [, *Time*, *Dataset*: integer, *NearestMatch*: boolean]): integer;
ChartBar(*Window*: integer, *eIndex*, *Date* [, *Time*, *Dataset*: integer, *NearestMatch*: boolean]): integer;

DESCRIPTION: **Bar** and **ChartBar** are used to retrieve price values from the referenced Bars of a Chart. Since there may be multiple Chart windows open at the same time, it is important to first identify the Chart that the functions will apply to. A global ESPL variable named *Window* is used to specify which chart to use. The *Window* variable can be manually assigned a window pointer value (if you have been keeping track of the window handles), or you can use the **FindWindow** or **Chart** functions to set the *Window* variable.

Bar(*eIndex*, *Date*, *Time*) is used to find the bar Index position for a given date and time. The function returns a zero if the bar is not found. If the time is omitted then the index for the last bar of the day will be returned. For Tick charts, **Bar**(*eIndex*, *Date*, *Time*) returns the first occurrence for duplicate time stamps.

ChartBar functions are identical to the **Bar** functions, except that a Chart must be identified in the 1st parameter. **ChartBar** uses either the name of the Chart or a Window handle number to know which chart data-set to use. **ChartBar** allows you to access chart data from any open chart window. Use **FindWindow** to get a Window handle number.

PARAMETERS:

Type: *Type* is one of the following predefined constants:

eArray	Returns the vArray[Index] value
eClose	Returns the bar's Close price
eColor	Returns the bar's color value, unless it is one of the following set by a ColorBar study: 0 = Normal 1 = Bullish color 2 = Bearish color 3 = ESPL color 4 = Volume color 5 = OpenInt color 6 = Grid color 254 = background color (hidden).
eColorValue	Returns the bar's color value. eColor could return 1 for Bullish color, whereas eColorValue will return the color value.
eDate	Returns the bar's date value in the format of yyymmdd. 100 represents the year 2000.

For years 2000+ $yyy = 100 + \text{last 2 year digits}$ (example 106 = 2006, 1061225)
 For years 1999 and prior the year will be returned as 97, 98, 99 (example: 991225);

eDateTime	Returns the bar's Date and Time value as a TDateTime variable.
eDay	Returns the Calendar Day of the bar
eHigh	Returns the bar's High price
eIndex	Returns the bar Position index for a given date and time
eLast	Returns the bar's Close price (last price)
eLow	Returns the bar's Low price
eMidPoint	Returns the bar's $(\text{High} + \text{Low}) / 2$
eMid3	Returns the bar's $(\text{High} + \text{Low} + \text{Last}) / 3$
eMid4	Returns the bar's $(\text{Open} + \text{High} + \text{Low} + \text{Last}) / 4$
eMonth	Returns the bar's Calendar Month
eNet	Returns the bar's Net change (Last - PriorLast)
eOpen	Returns the bar's Open price
eOpenInterest	Returns the bar's Open Interest value
ePercent	Returns percent value Close is, in the High/Low range $(100 * (\text{Last} - \text{Low}) / (\text{High} - \text{Low}))$
eRange	Returns the High Low range of the bar (High – Low)
eRawTime	Returns the bar's raw time stamp. For intra-day bars this is seconds from 1970.
eSecond	Returns the bar's Seconds time stamp value. Example, 15 equals 15 seconds.
eTickCount	Returns the Tick Count for the bar. Useful with Constant Tick bars.
eTime	Returns the bar's Time value in the format of HHMM
eTrueHigh	Returns the higher of either the High price or previous bar's Close
eTrueLow	Returns the lesser of either the Low price or previous bar's Close
eTrueRange	Returns the True High - True Low range
eVolume	Returns the bar's Volume
eYear	Returns the bar's Calendar Year. 100 is used to represent the year 2000.
1	Returns indexed values from the 1st User Study Array
2	Returns indexed values from the 2nd User Study Array
3	Returns indexed values from the 3rd User Study Array
4	Returns indexed values from the 4th User Study Array
eAskVol	Returns the Ask Volume
eBidVol	Returns the Bid Volume
eAskRatio	Returns the Ask Ratio
eBidRatio	Returns the Bid Ratio
eBuyPress	Returns the Buy Pressure
eSellPress	Returns the Sell Pressure
eBuyRatio	Returns the Buy Ratio
eSellRatio	Returns the Sell Ratio

Note: Buy Pressure and Sell Pressure are similar to Ask Volume and Bid Volume. However, the Pressure values are derived from bar prices using proprietary formulas.

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart. If *Index* is less than or equal to zero, the function will use index as an offset from the last bar on the chart. If *Index* is out of range, the function will return zero. Both a host chart and its optional overlay chart use the same indexing.

Date: *Date* is the bar's date as a integer in the format of *yyymmdd*.
 Example: 1001231 represents December 31, 2000
 98= 1998, 99= 1999, 100= 2000, 101= 2001, 102= 2002

Time: *Time* is the bar's time stamp as a long integer in the format of *hhmm*, example: 830
 For daily, weekly, and monthly charts, the *Time* value should be zero, or omitted.

Seconds: *Seconds* is the bar's Seconds time stamp as a long integer. Example: 15 equals 15-seconds.

For daily, weekly, and monthly charts, the *Seconds* value should be zero, or omitted.

Dummy: A *Dummy* parameter that is not used. Enter a zero value. Used only as a place holder.

DataSet: *DataSet* is used if the chart contains an overlay chart. In this case, two chart datasets can be accessed. The default is to use the host chart's data. To access the overlay chart's data pass the overlay's object number as the *DataSet* parameter, or pass a number 1, 2, 3, etc. (meaning the 1st, 2nd, or 3rd overlay). NOTE: The overlay object number can be obtained with the **FindStudy** command.

NearestMatch is an optional flag. Set the value to TRUE to locate and return the nearest or exact bar. Set the value to FALSE (or omit the parameter) to only find the exact match. If the exact match is not found, then the function will return a zero. This parameter allows you to find the nearest *eIndex* bar to the specified Time and Date (even on a different chart) if the exact matching time and date are not found.

EXAMPLE:

```
var                               {Start of Variable declarations}
  WindowX:integer;                {Declare WindowX as an integer variable}
begin                             {Start of Main programming code}
  Output(eClear);
  Chart('MSFT.D');                {Open a MSFT Daily chart}
  Chart('INDU.D');                {Open an INDU Daily chart}
  Chart('IBM.D');                 {Open an IBM Daily chart}
  writeln(Bar(eIndex, 1000315));   {Print Index of bar March 15, 2000 date}
  writeln(Bar(eDate,BarEnd));     {Print the Date of the last IBM bar}
  writeln(Bar(eTrueRange, BarEnd)); {Print TrueRange of last IBM bar}
  writeln(ChartBar('INDU.D', eHigh, 500)); {Print High at Index 500}
  WindowX:=FindWindow(eChart, 'MSFT.D'); {Find MSFT chart}
  writeln(ChartBar(WindowX, eLow, BarEnd)); {Print Low of last MSFT bar}
end;                              {End of Program}
```

BarBegin BarEnd BarLeft BarRight BarBeginLeft

SYNTAX: *BarBegin*: integer;
 BarEnd: integer;
 BarLeft: integer;
 BarRight: integer;
 BarBeginLeft: integer;

DESCRIPTION: These five global chart variables contain Bar Index positions for a chart. They are frequently used in User-defined Studies and User-defined Color Bar programs.

BarEnd: The Bar Index position for the last bar of the Chart.
BarLeft: The Bar Index position for the bar on the left edge of the Chart.
BarRight: The Bar Index position for the last Visible bar (useful if the chart has been shifted to the right).
BarBegin: Initially set to 2 (second bar of the chart data). On subsequent calls it is set to *BarEnd*.
BarBeginLeft: Initially set to *BarLeft*. On subsequent calls it is set to *BarEnd*.

NOTE: **FOR** loops are often used to loop through all the bars on a chart. Calculations or studies are performed on the chart data. The first time a User-defined Study runs, Ensign will set *BarBegin* to 2, and *BarEnd* to the last bar index. These variables can then be used in the **FOR** loop, to loop from the first bar to the last bar of the chart.

On a real-time chart, the ESPL programming code is called at the completion of every new bar (or every tick). Rather than loop through every bar again, calculations for just the most recent bar are normally performed. For this reason, on all subsequent calls to the programming code, Ensign will set *BarBegin* and *BarBeginLeft* to be the index of the last bar. In other words, *BarBegin* and *BarBeginLeft* will equal *BarEnd*. Programming calculations should loop from *BarBegin* or *BarBeginLeft* to *BarEnd*. *BarLeft* is the index position of the first visible bar on the chart. Use *BarBeginLeft* instead of *BarBegin* when you only want to loop through the visible bars of the chart.

EXAMPLE: This program can be run by opening a chart, and then clicking ESPL button **100** on the Run ESPL panel. This starts a User-defined study and sets the *ESPL* variable to 100. The ESPL program checks to see if *ESPL* = 100 and then calls the 'PrintPrices' procedure if true. The 'PrintPrices' procedure outputs the Date, plus the High and Low prices of each bar to the output window. At the completion of every new bar on the chart, the ESPL programming code will get called again. On these subsequent calls to the program, *BarBegin* will equal *BarEnd*, so that only the new bar will get added to the printed list (the loop will be from *BarEnd* to *BarEnd*...the last bar only). In this example, the program loops through all the chart bars on the initial run; on all subsequent calls to the program (when a new bar completes on the chart) just the last bar is printed.

```
procedure PrintPrices;           {Declare a procedure named PrintPrices}
var
  LoopCount: integer;           {Declare LoopCount as an integer}
begin
  for LoopCount := BarBegin to BarEnd do      {Loop from BarBegin to BarEnd}
  begin                                       {Start of Loop Programming code}
    writeln('Date=', Bar(eDate,LoopCount));   {Print the Date of each bar}
    writeln('High=', Bar(eHigh,LoopCount));   {Print the High of each bar}
    writeln('Low =', Bar(eLow,LoopCount));    {Print the Low of each bar}
    writeln('');                             {Print a Blank line to separate days}
  end;                                       {End of Loop Programming code}
end;

                                     {***Main Programming Code***}
begin                                   {Program Execution Starts here}
  if ESPL=100 then PrintPrices;
end;
```

Beep

SYNTAX: **Beep**;

DESCRIPTION: The **Beep** statement is used to make a beep sound. The statement activates the Windows API MessageBeep function.

EXAMPLE:

```
begin
  Beep;
end;
```

Begin...End

SYNTAX: **Begin**
 {multiple statements}
 End;

DESCRIPTION: The **Begin** and **End** statements are used to group together and to block sections of programming code. The main programming code for any program starts with the **Begin** statement and ends with the **End** statement. The **Begin** statement does not require a semi-colon. Grouping several lines together in a **Begin...End** block forms a compound statement. Compound statements are frequently used in **If...Then...Else** statements, **For** loops, and **While** statements.

For example, if a program requires several things to happen in an **If...Then...Else** statement, then use a **Begin...End** block to enclose all the necessary programming statements. Each statement in the **Begin...End** block should end with a semi-colon.

EXAMPLE: The following example uses several blocks of code. When the program is run, it first tests to see if the time is past 09:30 AM. If so, it then loads an IBM.60 chart and checks to see if the current price is higher than the closing price of the previous hourly bar. Depending on the results, the program will beep and print an appropriate message. The chart is then closed.

```
var
  s: string;
begin
  if TimeStr > '09:30' then
  begin
    Chart('IBM.60');
    if Last(BarEnd) > Last(BarEnd-1) then
    begin
      beep;
      writeln('IBM is Higher this hour');
    end
    else
    begin
      beep;
      writeln('IBM is Lower this hour');
    end;
    mnuCloseWindow.click;
  end;
end;
```

{Start of Main programming code}
{If time > 9:30 AM then proceed}
{Start of 1st If...Then Block}
{Open an IBM 60-minute chart}
{If bar > previous bar then do}
{Start of 2nd Block}
{Make a beep sound}
{Print message}
{End of 2nd Block}
{else otherwise do 3rd Block code}
{Start of 3rd Block}
{Make a beep sound}
{Print message}
{End of 3rd Block}
{Close the Chart}
{End of 1st If...Then Block}
{End of Program}

NOTE: Use Tabs to indent the Blocks of code. It makes the programming code much easier to read and follow. The **Else** statement included in the **If...Then...Else** statement should not be preceded by a semi-colon. That is why the first **End** statement in this example is not followed with a semi-colon.

Bullet

SYNTAX: **Bullet**(*Index*: integer, *Position*: integer [, *Color*: Integer]): integer;

DESCRIPTION: **Bullet** will add bullet markers to a chart similar to the Alert Draw Tool. **Bullet** returns the color of the chart pixels before a bullet is drawn. The return value can be used to test for the presence of an existing bullet. **Bullet** returns 0 when the pixels are the chart background color. A bullet will not be drawn when the *Color* parameter is omitted. Use *eNone* as the *Color* parameter to erase a bullet. The **Bullets** are not permanent objects and will not remain on the chart if the chart is moved or closed. The ESPL program must draw the bullets as needed.

PARAMETERS:

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart.

Position: *Position* is one of these predefined constants for the location of the bullet:

eHigh	eLast	eLow	eMidPoint	eOpen			
1	2	3	4	5	6	7	8

1, 2, 3 and 4 are the 1st, 2nd, 3rd, and 4th Top rows where bullets are placed.
5, 6, 7, and 8 are the 1st, 2nd, 3rd, and 4th Bottom rows where bullets are placed.

Color: *Color* may be one of the following numbers for the bar color. These values will use the colors that are selected on the SetUp | Chart form:

- 0 = Normal
- 1 = Bullish color
- 2 = Bearish color
- 3 = Big Cross color
- 4 = Volume color
- 5 = OpenInt color
- 6 = Grid color
- 254 = background color (hidden) = eNone.

Setting a bullet's color to eNone will remove a bullet.

Color may be one of these predefined [color constants](#) or a hex BlueGreenRed number:

Note: Because cBlack has a value of 0, using it would color the bar with the Normal color instead of black. Uses the Window variable as set by the FindWindow or Chart function.

EXAMPLE: The following program will draw a Red bullet on the top row, if a bullet does not already exist there.

```
begin
  FindWindow(eChart);
  if Bullet(BarEnd,1)=0 then Bullet(BarEnd,1,c1Red);
end;
```

Buttons

DESCRIPTION: Many of the program Buttons in Ensign can be accessed with the ESPL language. Button properties can be read and set. Buttons can also be programmatically clicked (as if the button had been clicked with the mouse). Buttons can also be disabled, resized, and filled with unique captions and hints. This allows ESPL programs to utilize the power and features of Ensign buttons. To click a button, list the button name followed by the 'Click' command (example: `btnQuote.click` will open a quote page).

PROPERTIES AND METHODS:

Click: *Click* simulates a mouse click, as if the user had clicked on the button.

Example: `btnNews.click;` will open the News page

Caption: *Caption* specifies the text on the button. Not all buttons have captions. The example changes the 'ESPL' to an 'X' on then ESPL button. Example: `btnESPL.caption := 'X';`

Checked: *Checked* can be used to set or read the up or down state of a button.
Example: `btnHelp.Checked := True;` will depress the button.

Cursor: Specifies the mouse image when the cursor passes over a button. The example will cause the mouse cursor to change to an 'HourGlass' when moved over the **Exit** button. The available mouse cursors are shown below.

crDefault Uses the default Windows mouse cursor.
 crArrow Uses an Arrow cursor.
 crCross Uses a Cross cursor.
 crIBeam Uses an I-Beam cursor.
 crSize Uses an Arrow cursor.
 crDrag Uses the Drag cursor.
 crHourGlass Uses the HourGlass cursor.

Example: `btnESPL.cursor := crHourGlass;`

Enabled: This allows you enable or disable a button. If a button is disabled, then it will not respond to any mouse clicks or to the keyboard. Set *Enabled* equal to either `False` or `True` to disable or enable the button.

Example: `btnESPL.enabled := False;` causes the button to be disabled.

ImageIndex: Change the button image. See the [Image List](#) in the Appendix for a list of index values.

Example: `btnLive.ImageIndex := 148;` sets the Live feed button on the main toolbar to the Green ball image

Visible: This allows you to actually hide a button. `True` makes the button visible. `False` hides the button.

Example: `btnChart.visible := False;` hides the **Charts** button.

Hint: Allows you to change the Hint text that appears when the mouse is over a button.

Example: `btnQuote.hint := 'Click here to Open a Quote Page';`

ShowHint: Allows you to enable or disable hints for a button. When `True`, the button will display the Hint.

Example: `btnAlert.ShowHint := False;` disables hints for the **Alerts** button.

To move or resize a button, use the following button properties.

Height: *Height* is the vertical size of the button in pixels.

Width: *Width* is the horizontal size of the button in pixels.

Top: The vertical coordinate of the top edge of the button.

Left: The horizontal coordinate of the left edge of the button.

Example: `btnQuote.height := 12;` reduces the height of the Quote button to 12 pixels.

AVAILABLE TOOLBAR COMPONENT NAMES:

Main Ribbon	Setup Ribbon	Window Ribbon	Help Ribbon
btnLayout	btnFeeds	btnCascade	btnStaff
btnChart	btnSystem	btnTileHorz	btnNewUser
btnStack	btnCharts	btnTileVert	btnContactUs
btn1 through btn9	btnPackage	btnClose	btnRunRemote
btnQuote	btnInternet	btnFind	btnJoinWebinar
btnOption	btnTheme	btnWindowColor	btnHome
btnNews	btnPrinter	btnWindowFont	btnDocs
btnAccount	btnCustomSymbols	btnPrint	btnHelp
btnAlert	btnOptimizeTrades	btnImageTo	btnVideo

btnOrderEntry	btnChartScanner	btnExit	btnWhatsNew
btnSpreadSheet	btnPlayback	btnAboutEnsign	btnDownloads
btnChatRooms	btnClock		btnUpgrade
btnDatabase	btnSymbolProperty		btnSymbols
btnESPL	btnCrossRef		btnHardwareReport
btnRunESPL	btnHolidaySchedule		btnReports
btnOutputWindow			btnLog
Main Toolbar	Labels	Forms	
btnLive	lblEnsign	frmMain	
btnPin	lblCaption	frmRunESPL	
btnMinimize			
btnMaximize			
btnExitEnsign			
Run ESPL Form			
btnESPL#	Where # is the ESPL tag: 0..9, 100..109, 200..209, 300..309, 400..403, 500..503		

EXAMPLE: The following program clicks the **Quote** page button, prints the quote page, and then closes the window.

```
begin                                {Start of Main programming code}
  btnQuote.click;                    {Click Quotes button to open a quote page}
  btnPrint.click;                    {Click Print button to print the quote page}
  btnClose.Click;                   {Close all child windows}
  btnESPL1.Caption := 'Reset';      {Change this ESPL button's caption}
end;                                  {End of program}
```

EXAMPLE: The following program will change the screen to show Layer 2, and show the RUN ESPL form.

```
begin
  btn2.click;                        {Display layer 2}
  frmRunESPL.Visible := true;       {Display the Run ESPL form}
end;
```

Callback

SYNTAX: **Callback**(*Index*: integer);

DESCRIPTION: The **Callback** statement is only used with a User study. When a User study executes, it sets BarBegin and BarEnd prior to making the call. Normally a study would be designed to iterate through all bars between these two indexes using a FOR loop. If the calculation time is too long, one might consider calculating for 1 bar, and then issue a **Callback** statement and pass as the parameter the *Index* where the calculation should resume. *Index* will be used to set

the value for BarBegin. **CallBack** posts a windows message to the Ensign program to execute the ESPL user study script again. The ESPL script will terminate so other processes can run, such as processing the data feed. Then the windows message will rerun the ESPL script for the user study.

PARAMETERS:

Index: Specifies the value that BarBegin will be set to prior to executing the User study's script.

EXAMPLE: Open a chart, and click button 100 on the Run ESPL panel to add ESPL User study 100.

```
begin                                     {Start of Main programming code}
  if ESPL = 100 then begin                {process calls from the ESPL study}
    if BarBegin < BarEnd then begin      {repeat while there are more bars to calculate}
      writeln(BarBegin);                  {show some action}
      CallBack(BarBegin + 1);            {post windows message to execute again}
    end;                                  {end of BarBegin < BarEnd block}
  end;                                    {end of ESPL = 100 block}
end;                                       {End of program}
```

Chart

SYNTAX: **Chart**(*FileName*: string [, *Feed*: constant]): integer;

DESCRIPTION: The **Chart** function is used to open and display a chart. The function returns the chart's child window position, and internally sets the global ESPL *Window* variable equal to the chart child window. NOTE: You can manually set the ESPL *Window* variable to point to a child window, but the ordering of windows can change when any window gets focus, or new windows are opened. You shouldn't try to remember a *Window* number for a specific chart, in order to identify that chart later in your code. Instead, use the **FindWindow** command. The Feed for the chart symbol can be optionally specified.

PARAMETERS:

FileName: Specifies the Chart file to open and display. Examples: 'IBM.D', 'AAPL.1', 'INDU.W', etc.

Feed: *Feed* is one of these predefined constants. The default is the value assigned to the FEED global variable.

eFXCM	eIB	eSignal	eIQFeed	eNinja	eOpenECry
eTraderBytes	eTransAct	eGlobal	eDBFX	eATCBrokers	eCustom

EXAMPLE: The following example opens an IBM daily chart and prints the last 10 Dates and Closing prices of the chart.

```
var                                     {Start of Variable declarations}
  a: integer;                           {Declares 'a' as an integer variable}
begin                                    {Start of Main programming code}
  Chart('IBM.D');                        {Open the IBM daily chart}
  for a := BarEnd-9 to BarEnd do writeln(Bar(eDate,a), ' ', Last(a));
  mnuCloseWindow.click;                  {Close the chart window}
end;                                       {End of program}
```

ChartLoad

SYNTAX: **ChartLoad**(*FileName*: string [, *Feed*: constant]): boolean;

DESCRIPTION: **ChartLoad** is used to open a chart (without painting the bars and studies in the chart window). Use **ChartLoad** when you need to load several charts in a row, and perform calculations on the chart data. **ChartLoad** is very fast because the chart bars are not painted in the chart window (except the first chart requested). The chart data and chart studies are loaded into memory and can be accessed the same as with any chart. If several charts need to be tested, then **ChartLoad** should be used instead of the **Chart** command. **ChartLoad** allows you to access all of the chart and study data without actually painting the data to the screen. If no charts are open, then **ChartLoad** will open an initial chart window. Once a chart window is open, subsequent calls to **ChartLoad** will load the new chart into the existing window. This increases the speed for loading several charts in a row. **ChartLoad** returns a True value if the chart file was able to load properly. The Feed for the chart symbol can be optionally specified.

PARAMETERS:

FileName: *FileName* is the chart file that will be loaded. If necessary, *FileName* may include the path to the file. However, if no path is provided Ensign will determine the path based on the file extension.

Feed: *Feed* is one of these predefined constants. The default is the value assigned to the FEED global variable.

eFXCM	eIB	eSignal	eIQFeed	eNinja	eOpenECry
eTraderBytes	eTransAct	eGlobal	eDBFX	eATCBrokers	eCustom

EXAMPLE: The following example scans through all the symbols in the NASDAQ stock market group. If a symbol's current price is between 55 and 60 dollars, then a chart file is loaded. Each chart is then tested to see if the current price is Higher than the close price of 10 bars ago. All symbols which meet these requirements are printed to the output window.

```
begin                                     {Start of Main programming code}
  Find(eSignal);                          {Locate the eSignal quote symbols}
  repeat                                  {begin Repeat...Until loop}
    if GetData(eLast) <= 60 then          {test each symbol to see if Last <= 60}
      if GetData(eLast) >= 55 then begin  {test each symbol to see if Last >= 55}
        ChartLoad(GetData(eSymbol)+ '.D'); {Load chart for the current symbol}
        if Last(BarEnd) > Last(BarEnd-10) then writeln(GetData(eSymbol));
      end;                                 {end of If..then block}
    until not Find(eNext);                {Get next symbol, loop back to repeat}
    mnuCloseWindow.click;                 {Close the chart window}
end;                                       {End of program}
```

ChartRefresh

SYNTAX: **ChartRefresh**([*True*, *Window*, *Recalc*: integer, *RecalcESPL*: boolean]): boolean;
CharRefresh(*False*): boolean;

DESCRIPTION: The **ChartRefresh** function is used to control the recalculation and refreshing of chart lines and tools. The **ChartRefresh**(*True*) command causes a chart to redraw or refresh itself. The *Window* parameter can be used to refresh a specific study sub-window panel (rather than the whole chart). All the lines and tools on the chart will be refreshed (redrawn). The **ChartRefresh**(*False*) command disables chart refreshing. This may be desired if multiple lines are being drawn on a chart (causing a noticeable blink for each item). The refresh should be disabled until all the lines are drawn. Then the refresh can be enabled again with the **ChartRefresh**(*True*) command. Enter any value in the 3rd parameter (*Recalc*) to cause all studies on the chart to recalculate and repaint (except ESPL studies). Enter *True* in the 4th parameter (*Recalc ESPL*) to recalculate and repaint all studies on the chart, including ESPL studies. Be careful when using this 4th parameter that you don't design your programming code so that it goes into an endless loop. The 4th parameter should only be used when you are manually clicking a button to cause a *Recalc* of all studies.

NOTE: **ChartRefresh**() without a parameter defaults to **ChartRefresh**(*True*). The **ChartRefresh** command could be used when using the **AddLine** command to add many lines to a chart. The chart refresh should be disabled first, then the

lines should be drawn, then the refresh should be turned on again. **ChartRefresh(True)** enables the chart refresh and forces the chart and its lines to redraw. **ChartRefresh(True)** should be used after using the **SetBar(eColor...)** function. NOTE: Try not to create a circular loop where **ChartRefresh** causes the chart to recalculate, which causes **ChartRefresh** to execute again, and the process repeats endlessly. It would be inappropriate to use **ChartRefresh** in an ESPL Color Bar study for this reason.

PARAMETERS:

- Window:* The *Window* parameter is used to refresh a specific study sub-window panel. For example, `ChartRefresh(True,2)` will refresh study sub-window 2. Only the indicated sub-window will be refreshed. The study is not recalculated, but is redrawn using current values. The following can be used:
- 0= Refresh the main Chart window
 - 1= Refresh study sub-window 1
 - 2= Refresh study sub-window 2
 - 3= Refresh study sub-window 3
 - 4= Refresh study sub-window 4
 - 5= Refresh study sub-window 5
 - 6= Refresh study sub-window 6
 - 7= Refresh study sub-window 7
 - 8= Refresh study sub-window 8
 - 9= Refresh the Volume sub-window
- Recalc:* If any parameter is entered in this position, then all Chart studies will recalculate and repaint (except ESPL studies).
- Recalc ESPL:* If a True a value is entered in this 4th parameter position, then all Chart studies will be recalculated, including any ESPL studies on the chart.

EXAMPLE: The following program opens an IBM daily chart and adds an RSI study using default settings (assumed to be displayed in the 2nd study sub-window). A study parameter for the RSI is changed and the study is recalculated and refreshed. A **FOR** loop is then used to recolor the bars on the chart using the **SetBar** command. The **Random** command is used to generate a random color value for each bar. After the loop is finished, the **ChartRefresh** command is used to force a chart redraw (allowing you to see the new bar colors).

```

var                                     {Start of Variable declarations}
  i, ID: integer;                       {i and ID are declared as an Integer}
  Color: real;                           {Color is declared as a Real}
begin                                    {Start of Main Programming code}
  Chart('IBM.D');                        {Open an IBM daily chart}
  ID:= AddStudy(eRSI,0);                  {Add an RSI Study to the chart}
  SetStudy(ID,8,10);                      {Change the RSI 'Bar' Parameter to 10}
  ChartRefresh(True,2,True);              {Recalc RSI study in the 2nd sub-window}
  for i:=BarLeft to BarEnd do begin      {Loop through the visible bars}
    Color:= Random(10000000);            {Generate a random color value}
    SetBar(eColor,i,Color);              {Change the bar's color}
  end;                                     {end of loop block}
  ChartRefresh(True);                     {Redraw and refresh the chart}
end;                                       {End of program}

```

ChartReplace

SYNTAX: **ChartReplace(FileName: string): boolean;**

DESCRIPTION: **ChartReplace** is used to replace the currently opened chart, with a different chart. The new chart is loaded into the previously opened chart window and displays the bars and studies. **ChartReplace** can be used to change the symbol for all charts whose Symbol Color Box setting is the same.

PARAMETERS:

FileName: *FileName* is the chart file that will be loaded. If necessary, *FileName* may include the path to the file.

EXAMPLE: The following example opens an IBM daily chart, pauses for 10 seconds, replaces the chart with a MSFT daily chart, pauses for 10 more seconds, and then closes the chart window.

```
begin                                {Start of Main programming code}
  Chart('IBM.D');                    {Open an IBM daily chart}
  Pause(10);                          {Pause 10 seconds while you view chart}
  ChartReplace('MSFT.D');            {Replace the IBM chart window with MSFT}
  Pause(10);                          {Pause another 10 seconds}
  mnuCloseWindow.click;              {Close the active window}
end;                                  {End of program}
```

ChartSave

SYNTAX: **ChartSave**;

DESCRIPTION: **ChartSave** causes the active chart window to Save its current data contents (bar price data) to the computer hard disk. Chart data is normally saved whenever a chart is loaded or closed, however, **ChartSave** causes the chart data to be saved while the chart is still open. This may be desired if the chart file is being accessed by another software program, and requires up-to-the-minute current data. If a chart properties file already exists, the chart properties data will also be saved.

PARAMETERS: None.

EXAMPLE: The following example assumes that an IBM 5-minute chart is already open on the screen. The program finds the chart, and causes the chart to Save its data.

```
begin                                {Start of Main programming code}
  FindWindow(eChart, 'IBM.5');        {Find the IBM 5-minute chart}
  SetMyFocus;                          {Set the focus to that chart window}
  ChartSave;                            {Save the Chart data while chart is still open}
end;                                    {End of program}
```

Chat ChatRoom

SYNTAX: **Chat**(*Text*: string, [*Icon*:integer]): boolean;
ChatRoom(*Room*: integer);

DESCRIPTION: The **Chat** function is used to print text into the Chat window. The function can optionally be used to change the Icon that appears next to your Chat Nickname. The function will return a True value if the text was posted successfully, otherwise a False value will be returned. It is not necessary to use the **FindWindow** command before posting text to the Chat room window.

The **ChatRoom** command is used to change to a different Chat room. Enter a Room number from 1 through 74. The

Chat rooms can be seen by clicking the drop-down list box on the right side of the Chat window. Each room has a name and number.

PARAMETERS:

Text: Enter the chat text that will appear in the room
Icon: Enter a value between 0 and 62. These values correspond to the 'My Icon' list in the Chat setup screen. This allows you to change the Icon that appears next to your Chat Nickname.
Room: Enter a chat room number from 0 to 99.

EXAMPLE: The following example can open a Chat window, switch to the 'New Users' room, and print some text into the room, while changing the Chat Icon next to the Nickname.

```
begin
  btnChat.click;    {Open Chat Screen}
  Pause(2);        {Wait until it opens}
  ChatRoom(1);     {Change to the New Users room}
  Pause(2);        {Wait until it changes}
  Chat('I have taken profits.', 18);    {Post a message, change icon}
end;
```

ChDir MkDir Rmdir

SYNTAX: **ChDir**(*Path*: string): boolean;
MkDir(*Path*: string): boolean;
Rmdir(*Path*: string): boolean;

DESCRIPTION: The **ChDir** function changes the current directory to the specified *Path*. If *Path* includes a drive letter, then the current drive is also changed. The **MkDir** function creates a new sub-directory with the specified *Path*. The last item in the *Path* cannot be an existing file name. The **Rmdir** function deletes the sub-directory specified in *Path*. If the path does not exist, is non-empty, or is the currently logged directory, the function will fail. These functions return True when successful, and False when an error occurs.

PARAMETERS:

Path: Specifies the directory *Path* on the hard disk. Example: 'C:\ENSIGN10\ESPL'

EXAMPLE: The following program changes the current directory. A new directory is created. The directory is then deleted.

```
begin
  ChDir('C:\ENSIGN10\ESPL');
  MkDir('C:\ENSIGN10\TEMP');
  Rmdir('C:\ENSIGN10\TEMP');
end;
```

ChildCount

SYNTAX: **ChildCount**: integer;

DESCRIPTION: The **ChildCount** function is used to retrieve a count for the number of child windows that are open. All of the Chart windows, Quote windows, Alert windows, News windows, etc., are child windows within the main Ensign application. The **ChildCount** function is normally used to determine the number of child windows that are open. This number can be used in loops to access information from each child window.

EXAMPLE: The following program uses a **FOR** loop to access information for each child window that is open. The form name for each child window is printed. The mouse cursor for chart windows is changed to a CROSS. After running this program the mouse cursor will change to a CROSS when moving over the chart windows. The **Child** function and a variable of type *TForm* are used to access and set the child window information.

```

var                                     {Start of Variable declarations}
  ChildForm: TForm;                    {ChildForm is declared as a TForm variable}
  j: integer;                           {J is declared as an Integer variable}
begin                                   {Start of Main Programming code}
  for j := 0 to pred(ChildCount) do     {Loop through all child windows}
  begin                                  {block start}
    ChildForm := Child(i);              {ChildForm is assigned to a child window}
    writeln(ChildForm.Name);            {Print the form name of the child window}
    if pos('Chart',ChildForm.Name) > 0 then begin
      ChildForm.Cursor:= crCross;      {Change the form's cursor style}
      Window := j + 1;                  {Window needs to be 1 more than the index}
      SetMyFocus;                       {Make the Window selection the chart active}
      Remove(eStudy);                  {Remove all studies from the active chart}
    end;                                 {end of block}
  end;                                   {end of block}
end;                                     {End of program}

```

Child

SYNTAX: **Child**(*index*: integer): TForm;

DESCRIPTION: The **Child** array holds the child windows that are open. All of the Chart windows, Quote windows, Alert windows, News windows, etc., are child windows within the main Ensign application. **ChildCount** is the number of members in the array. See the **ChildCount** example.

Choose

SYNTAX: **Choose**(*Title*: string, *Selection1*: string [, *Selection2*: string, .. *SelectionN*: string]): integer;
Choose(*Title*: string, *List.Text*: TStringList): integer;
Choose(*Title*: string, *Top*, *Left*, *Height*: integer, *Selection1*: string [,...*SelectionN*:string]): integer;
Choose(*Title*: string, *Top*, *Left*, *Height*: integer, *Selections.Text*: StringList): integer;

DESCRIPTION: The **Choose** function is used to open a small window containing a list of custom selections. The window contains the list of selections, plus a **Cancel** button, and a **Close** button. The user makes a selection from the list and then clicks the **Close** button. A selection can also be made by double-clicking on an entry. Or, the user can click the **Cancel** button. The **Choose** command returns the list position number, for the item that was chosen. The command returns a 0 if the user clicks the **Cancel** button, or clicks the **Close** button without making a selection. The predefined global string variable named *IT* is assigned the contents of the selected item from the choose list. The *IT* variable can then be used to print or use the selected string item.

The screen location of the list box can be optionally specified by including the *Top*, *Left*, and *Height* parameters (before the *Selection* entries). The *Top*, *Left*, and *Height* parameters specify how many screen pixels to use for each item. The *Top*

coordinate is the 2nd parameter and specifies the top of the selection box (counting down from the top of the screen). The *Left* coordinate is the 3rd parameter and specifies the left edge of the box. The *Height* of the box is the 4th parameter.

The **Choose** function can be used to present a list of items to choose from. The user can choose a particular item from the list, and then the ESPL program can proceed based on the selection. Example: The **Choose** command could be used to display a listing of the programs contained in the ESPL script file. The user could choose which program to run from the list, and the *ESPL* variable could be assigned the result. The corresponding ESPL program could then be run.

PARAMETERS:

Title: The *Title* parameter is the text that will appear as the Heading of the **Choose** window.
Selection1: Enter the text that will be used in the selection list.
Selection2..N: Enter as many other text items, separated by commas, to complete the list of selection items.
List.Text: A `StringList` can also be used to populate the list of selection items. Prepare a `StringList` to contain the list of selection items, and then use the `.TEXT` property command to assign the list into the **Choose** window.
Top,Left,Height: The *Top*, *Left*, and *Height* parameters specify the optional screen location of the list box. See the following:

```
Choose('Caption String','First String');           {Opens in default position}
Choose('Caption String',100,'First String');       {Top is 100 pixels down}
Choose('Caption String',100,50,'First String');     {100 down and 50 left}
Choose('Caption String',100,50,400,'First String'); {100 down, 50 left, 400 height}
```

EXAMPLE 2: The following program displays a list of five stock symbols. A daily chart is opened for the selected symbol.

```
var i: integer;
begin
  i := Choose('Select a Daily Chart','MSFT','IBM','INTC','YHOO','CSCO');
  if i = 0 then
    ShowMessage('You clicked Cancel, or failed to make a selection.')
  else
    Chart(IT + '.D'); {Open Chart. IT contains the Symbol that was selected.}
end;
```

EXAMPLE2: This program displays a list of 3 programs that can be run from the ESPL script file. The global `StringList` variable `'sList'` is used to prepare the selection list. This is a good example of using the **Choose** function to run programs.

```
procedure Program1;
begin
  writeln('Program 1 was run');
end;

procedure Program2;
begin
  writeln('Program 2 was run');
end;

procedure Program3;
begin
  writeln('Program 3 was run');
end;

begin                                     {Start of Main Programming Code}
```

```

sList.clear;                               {Clear the sList StringList}
sList.CommaText:= 'Program1,Program2,Program3'; {Create list selections}
ESPL := Choose('Make a Selection',sList.Text);{Display list, assign ESPL }
if ESPL = 1 then Program1;                   {If ESPL=1 then run Program1}
if ESPL = 2 then Program2;                   {If ESPL=2 then run Program2}
if ESPL = 3 then Program3;                   {If ESPL=3 then run Program3}
end;                                         {End of program}

```

Chr Ord

SYNTAX: **Chr**(*Number*: integer): char;
 Ord(*Character*: char): integer;

DESCRIPTION:

The **Chr** function returns the alphabet Character equal to the ordinal value of the *Number* parameter.

The **Ord** function returns the ordinal value (number) of the *Character* parameter.

NOTE: Individual text and alphabet characters have an ordinal value. These values are used by computers to represent the characters numerically in bytes. For example, the ordinal values of the Capital letters A through Z are 65 through 90. Example, **Ord**('A') equals 65. **Ord**('B') equals 66, and so forth... **Chr**(65) equals 'A', **Chr**(90) equals 'Z'. Using these two functions allows you to determine the ordinal value of any Character, or to determine the Character that represents an ordinal number.

PARAMETERS:

Number: *Number* is a decimal value from 0 through 255. This value (ordinal value) represents a character to the computer. The values from 0-31 are Control characters (like the ESC key = 27). The values from 32-64 represent the space character=32, plus numbers (0-9) and other misc. characters. The values from 65-90 are the capitalized alphabet letters (A-Z). The values from 91-96 are various punctuation characters. The values from 97-122 are the small alphabet letters (a-z). The values from 123-255 are various punctuation characters and draw characters.

Character: *Character* can be any alphabet letter (either capitalized or not), plus all the other characters generally available on a computer keyboard.

EXAMPLE: The following example counts from 65 to 90 and prints the alphabet Character represented by the count value. The small alphabet Characters are also printed by adding 32 to the value of *Count* (resulting in 97-122). Lastly, a loop examines each character in the word 'HELLO' and prints the ordinal value of the character.

```

var                               {Start of Variable Declarations}
  Count: integer;                  {Declare Count as an integer}
begin                               {Start of Main programming code}
  for Count:=65 to 90 do writeln(Chr(Count), ' ', Chr(Count + 32));
  for Count:=1 to 5 do writeln(Ord(Copy('HELLO',Count,1)));
end;                               {End of program}

```

Clipboard

SYNTAX: **Clipboard**(*Text* :string);

DESCRIPTION: The **Clipboard** command is used to copy *Text* to the Windows clipboard. The *Text* can then be pasted into another application or window. Some customers use the **Clipboard** command to enable the computer to talk to them. There are programs that automatically SAY whatever is copied to the clipboard. For example, news alert titles can be copied to the clipboard. The computer would read and say the news title line.

PARAMETERS:

Text: The *Text* parameter is a string value that is copied to the Windows clipboard.

EXAMPLE: The following example copies the price of a symbol to the clipboard. Each time the price changes on a chart, the computer will say the price. Click the Run ESPL button, and then click ESPL button 100 to apply the study to the active chart. The program below also allows news story title alerts to be read. Click button 1 on the Run ESPL panel to activate the news reader. Click button 2 to stop the news reader. Set news alerts in Ensign to trigger the title lines to be read. You can download free talking software from Internet web sites.

```
procedure SayChartPrice;           {Copy chart price to the clipboard}
var
  s:string;
  Price:integer;
begin
  SetUser(eName, 'SayPrice');
  SetUser(eClose, False);
  Price:=Last(BarEnd);
  if (Price mod 100) = 0 then s:=IntToStr(Price) else s:=IntToStr(Price mod 100);
  Clipboard(s);
end;

begin
  if ESPL=1 then AlertEvent(eNews, 69); // Turn on News Title Alert Event
  if ESPL=2 then AlertEvent(eNews, 0); // Turn off the News Alert Event
  if ESPL=69 then Clipboard(IT); // Copy News title to Clipboard
  if ESPL=100 then SayChartPrice; // Run the SayChartPrices procedure
end;
```

AssignFile
Append
Reset
Rewrite
CloseFile
ReadLn
WriteFile
WriteLnFile
EOF
DeleteFile
FileExists
DirectoryExists
RenameFile

SYNTAX: **AssignFile**(*FileHandle*: Textfile, *FileName*: string);
 Append(*FileHandle*);
 Reset(*FileHandle*);
 Rewrite(*FileHandle*);
 Closefile(*FileHandle*);

 ReadLn(*FileHandle*): string;
 WriteFile(*FileHandle*, *Expression*);
 WriteLnFile(*FileHandle*, *Expression*);
 EOF(*FileHandle*): boolean;

 DeleteFile(*FileName*: string): boolean;
 FileExists(*FileName*: string): boolean;
 DirectoryExists(*DirectoryName*: string): boolean;
 RenameFile (*OldName*, *NewName*: string): boolean;

DESCRIPTION: Use the following File commands to open, close, delete, read, rename, and write data to an ASCII text file. Only one file can be open at a time.

AssignFile: **AssignFile** is used to open the specified text file. A *FileHandle* is created for use of the file. The variable used for the *FileHandle* should be previously declared as type: TextFile. After the *FileHandle* is created, then use the **Append**, **Reset**, or **Rewrite** command to set the file mode for input or output.

Append: This command can be executed after **Assignfile** obtains a *FileHandle*. Specifies that the File represented by the *FileHandle* is in Output mode. Written text will be appended to the end of the file.

Reset: This command can be executed after **Assignfile** obtains a *FileHandle*. Specifies that the File represented by the *FileHandle* is in Read-only mode. The file can only be read.

Rewrite: This command can be executed after **Assignfile** obtains a *FileHandle*. Specifies that the File represented by the *FileHandle* is in Output mode.

If a previous file existed with the same name, it is overwritten.

- CloseFile:** **CloseFile** closes the File represented by the *FileHandle*.
- Readln:** **Readln** is used to read one line of text from the File represented by the *FileHandle*. Use multiple **Readln** calls to read successive lines from the file. **Readln** reads one line of text and then points itself to the next line of text in the file. The **EOF** command will return a `True` value when the end of the file has been reached. A **While** loop is often used to loop through the whole file, reading each line. See the example below.
- WriteFile:** **WriteFile** writes the *Expression* to the File represented by the *FileHandle*. A line feed is not included with the text. Successive calls using the **WriteFile** command will continue to write data to the same line of text in the file. Use the **WritelnFile** command to complete a line of written text with a line feed. The *Expression* can be Numeric, String, or Boolean. Use the **Str**, **Format**, and **Align** functions to format a numeric expression with decimal places.
- WritelnFile:** This command is exactly the same as **WriteFile**, except that the written *Expression* is concluded with a line feed. This causes the next **WriteFile** or **WritelnFile** statement to write on the next line in the file. Use **WritelnFile**(*FileHandle*, ''); to print a blank line in the text file.
- EOF:** **EOF** returns a `True` value if the file pointer of the currently open text file is at the end of file. This function is used when reading each line of a file. When the last line is read, then the **EOF** function will return `True`.
- DeleteFile:** This function erases the specified *FileName* from the disk. If the file cannot be deleted or does not exist, the function returns a `False` value. The function returns a `True` value if the file was successfully deleted.
- FileExists:** **FileExists** is used to verify if a specified file already exists on the computer. The path and *FileName* are specified as parameters. The function returns a `True` value if the specified file exists, and `False` otherwise.
- DirectoryExists:** **DirectoryExists** is used to verify if a specified directory already exists on the computer. The path and *DirectoryName* are specified as parameters. The function returns a `True` value if the specified directory exists, and `False` otherwise.
- RenameFile:** **RenameFile** is used to rename a file on the computer hard disk. The function changes the name of the *OldName* file to be the *NewName*. Enter the complete path and file name if necessary. The function returns a `True` value if the file was successfully renamed, otherwise it returns a `False` value.

PARAMETERS:

- FileName:* *FileName* specifies the file to open, delete, or verify if it exists. Include the path of the file on the hard disk. If necessary, use the *sPath* global ESPL variable that points to the \ENSIGN10 folder.
- FileHandle:* *FileHandle* is a variable of type `TextFile` used to access an opened file. See the example below where 'f' is declared as the filehandle. Make sure that you declare this variable before using with other commands.
- OldName:* The file name and location of the file to be renamed.
- NewName:* Rename the *OldName* file to be this *NewName*.

EXAMPLE: The following example demonstrates all of the file commands documented above with some simple tasks.

```
var                                     {Start of Variable Declarations}
  i: integer;                           {Declare i as an integer}
  f: TextFile;                           {Declare a File Handle}
begin                                    {Start of Main programming code}
  AssignFile(f, sPath + 'TEST.TXT');     {Open a new file named TEST.TXT}
  Rewrite(f);                             {Output mode, overwrites previous file}
  for i := 1 to 9 do writefile(f,i);     {Write the numbers 1-9 in a row}
  writelnfile(f, '');                     {Write a blank line}
  writelnfile(f, 'Nine Numbers');        {Write the text 'Nine Numbers'}
  CloseFile(f);                           {Close the file}

  {Check to see if TEST.TXT exists, and then Rename it to be TEST.TMP}

  if FileExists(sPath + 'TEST.TXT') then
    RenameFile(sPath + 'TEST.TXT', sPath + 'TEST.TMP');
  Output(eClear);                          {Clear output window}
  AssignFile(f, sPath + 'TEST.TMP');       {Open TEST.TMP}
  Reset(f);                                {Set file mode to Read-only}
  writeln('Contents of TEST.TMP');         {Print to the output window}
  while not EOF(f) do writeln(ReadLn(f));  {Read, then Print each row of file}
  CloseFile(f);                             {Close the file}
  DeleteFile(sPath + 'TEST.TMP');         {Delete the file named TEST.TMP}
end;                                       {End of program}
```

EXAMPLE: This ESPL program demonstrates the usage of the these FILE commands: Append, AssignFile, Closefile, EOF, FileExists, Readln, Reset, Rewrite, WriteFile, WriteLnFile

```
uses
  Classes, SysUtils; {Make sure that SysUtils is in the Uses list}
var
  FileHandle: File; {Define a File Handle variable}
  MyString: string;
begin
  MyString:='This is the Text that will be written to the file';

  {Test to see if a File exists}
  if FileExists('C:\Ensign10\Test.txt') then writeln('Yes');

  {Associate the name of an external file with the File handle variable}
  AssignFile(FileHandle, 'C:\Ensign10\Test.txt');

  {Open the File for Writing, owerwrite the prior file if it existed}
  Rewrite(FileHandle);
```

```

{Write the string to the file with a line feed}
WriteLnFile(FileHandle, MyString);

{Write five strings in a row without a line feed}
WriteFile(FileHandle, '1 ');
WriteFile(FileHandle, '2 ');
WriteFile(FileHandle, '3 ');
WriteFile(FileHandle, '4 ');
WriteFile(FileHandle, '5 ');

{Write some strings to the file with a line feed}
WriteLnFile(FileHandle, '- A B C D E ');
WriteLnFile(FileHandle, 'Numbers and Letters');

{Now Open the File, with the file pointer at beginning of File}
Reset(FileHandle);

{Read and print lines from the File until the End of File}
while not EOF(FileHandle) do
begin
  MyString:=ReadLn(FileHandle);
  writeln(MyString);
end;

{Now Open the File, with the file pointer at the end of the File}
Append(FileHandle);

{Append a string to the end of the file with a line feed}
WriteLnFile(FileHandle, 'This is the last line of the File!!!');

{Close the File}
CloseFile(FileHandle);
end;

```

The following will appear in the Text.txt file after running this program:

```

This is the Text that will be written to the file
1 2 3 4 5 - A B C D E
Numbers and Letters
This is the last line of the File!!!

```

ColorBars

SYNTAX: **ColorBars**(Name: integer): boolean;

DESCRIPTION: The **ColorBars** function applies the *Named* ColorBar study to the active chart. The function returns a True value if the study was successfully applied to the chart, otherwise False.

PARAMETER:

Name: The *Name* parameter may be one of the following ColorBar predefined constants:

eBar	eCandlestick	eCloseVsOpen	eDay	eDunnigan	eFullMoons
eGap	eGapOpen	eHour	eIsland	eKeyReversal	eMajorTrend
eMinorTrend	eMinute	eMonth	eMoonPhases	eNet	eNone
eNormal	eOuter10	eOuter25	eOutside	eSmallTrend	eTrend

eTurningPoint eVolumeSize eWeekly eWeek

NOTE: ColorBars(eNone) is used to prevent Layouts from clearing bar coloring applied by ESPL programs.

EXAMPLE: The following example opens an IBM daily chart and applies the 'eTrend' ColorBar study to the chart. It then pauses for 5 seconds and sets the bar colors back to normal. See the Ensign ColorBar documentation for details on each ColorBar study.

```
begin                                {Start of Main Programming code}
  Chart('IBM.D');                    {Open an IBM daily chart}
  ColorBars(eTrend);                  {Apply 'eTrend' ColorBar study to chart}
  Pause(5);                           {Pause for 5 seconds}
  ColorBars(eNormal);                 {Reset the bar colors back to normal}
end;                                   {End of program}
```

ConvertPrice FormatPrice

SYNTAX: **ConvertPrice**(Price: real [, Scale: integer]): real;
 FormatPrice(Price: real [, Scale: integer]): string;

DESCRIPTION:

ConvertPrice: The **ConvertPrice** function converts a quote price from its Calculation format (decimal) to its Display format (points). For bond futures, the Calculation format is 125.25 (decimal), and the Display format is 12508 (points). For stocks, the Calculation format is 12525 (pennies), and the Display format is 125.25 (dollars).

FormatPrice: The **FormatPrice** function converts a Display format quote price into a string. The string is formatted in the same manner that prices are shown on quote pages. The returned string is typically 9 characters in length. A bond future display format of 12508 will be returned as a string ' 125-08'. A stock display format of 125.25 will return a string with fractions of ' 125 1/4'.

PARAMETERS:

Price: *Price* is the quote value to be converted or formatted.

Scale: *Scale* is a scaling factor used to convert and format a price between decimal and points. The default is the scale factor from the currently decoded database record. Scale can be obtained for a chart price by using the following command: **GetVariable**(eScaleFactor); Scale can be one of the following values:

5	Convert 100,000ths	
4	Convert 10,000ths	Currencies
3	Convert 1,000ths	Meats
2	Convert 100ths	S&P and stocks
1	Convert 10ths	Gold
0	No conversion, Integer	Cocoa.
-1	Convert 8ths	Grains.
-2	Convert 16ths	
-3	Convert 32nds	Bond futures
-4	Convert 64ths	Bond futures options
-5	Convert 128ths	
-6	Convert half 32nds	TY, FV
-7	Convert quarter 32nds	TU

EXAMPLE: The following example retrieves an IBM quote price (assuming a value of 104 7/16). The price is then printed to the output window in a Calculation price format and as a String formatted price. Next, a price of 94.25 is converted into 32nds and printed as an actual value, Display value, and String value.

```

var                                     {Start of Variable Declarations}
  price: real;                          {Declares Price as a Real variable}
begin                                  {Start of Main Programming code}
  Find(eSignal, 'IBM');                 {Finds and decodes the IBM quote record}
  price := GetData(eLast);              {Assigns Price the Last price}
  writeln(price, ' ', FormatPrice(price)); {Outputs 104.44 104 7/16}
  price := 94.25;                       {Assign a new value of 94.25 to Price}
  writeln(price);                       {Prints 94.25}
  writeln(ConvertPrice(price, -3));      {Converts into 32nds, prints 9408.00}
  writeln(FormatPrice(price, -3));      {Converts to String}
end;                                    {End of program}

```

Copy

SYNTAX: **Copy**(*Text*: string, *Start*: integer, *Count*: integer): string;

DESCRIPTION: The **Copy** function is used to copy and return a subset of text from an original text string. You must specify the *Start* character, and the *Count* of characters to copy and return from the start point. The returned subset string value can be printed or assigned to a string variable.

PARAMETERS:

Text: The *Text* parameter is the original string that will be copied from.
Start: Specifies which character in the *Text* to start copying from.
Count: Specifies how many characters to copy from the starting character.

EXAMPLE: The following example copies and returns the characters 'IBM' from the original text string 'Buy IBM Today'. In this example, the copy command starts at the 5th character and copies 3 characters.

```

var                                     {Start of Variable Declarations}
  NewString, OriginalString: string;    {Declares two variables as strings}
begin                                  {Start of Main Programming code}
  OriginalString := 'Buy IBM Today';    {Assigns original text contents}
  NewString := Copy(OriginalString, 5, 3); {Copies 3 characters}
  writeln(OriginalString);              {Prints the original text}
  writeln(NewString);                   {Prints the copied text}
end;                                    {End of program}

```

CopyFile

SYNTAX: **CopyFile**(*SourceFile*, *TargetFile*: string, *Preserve*: boolean);

DESCRIPTION: The **CopyFile** command is used to copy a file on the computer. The *SourceFile* will be copied to the *TargetFile* destination. Specify the path and filename for each entry. The *Preserve* parameter is used to indicate whether an existing file should be overwritten or not. If *Preserve* is False, and the *TargetFile* already exists, then the *TargetFile* will be overwritten. If *Preserve* is True, and the *TargetFile* already exists, then the **CopyFile** command will fail and return a False value.

PARAMETERS:

SourceFile: Specifies the path and filename for the file to copy (example: 'C:\ENSIGN\CUSTOM.QUO');
TargetFile: Specifies the path and filename for the file to copy to (example: sPath + 'HIST\INDU.D');
Preserve: Enter a True value to preserve existing files. Enter False to allow an existing file to be overwritten.

EXAMPLE: The following program will copy the CUSTOM.QUO file to a floppy disk in the A: drive.

```
begin
  CopyFile(sPath + 'CUSTOM.QUO', 'A:\CUSTOM.QUO', False);
end;
```

CreateProcess

SYNTAX: **CreateProcess**(*Program*: string, [*Wait*: boolean]): integer;

DESCRIPTION: The **CreateProcess** function is used to RUN another windows program. This allows you to start a completely different application using the ESPL programming language. For example, the ESPL language could be used to start a Spreadsheet program, Word Processor, Analysis program, etc. If the function succeeds, the return value will be a number greater than 31. If the function fails, the return value will be one of the following error values:

- 0 - The system is out of memory or resources.
- 2 - The specified file was not found.

PARAMETERS:

Program: The *Program* parameter is a text string containing the path and FileName of the program to run. If the parameter does not include a directory path, Windows searches for the executable file in this sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

Wait: Waits until the other program closes before continuing, or just carries on leaving the newly started program to its own devices. Default is False, do not wait.

EXAMPLE: The following program runs the eSignal Turbo Data Manager program.

```
begin
  CreateProcess('C:\Program Files (x86)\eSignal\winros.exe', false);
end;
```

Date DateStr

SYNTAX: **Date**: TDateTime;
DateStr: string;

DESCRIPTION: The **Date** function returns the current date. The *TDateTime* variable type is used to return the date. Note: You can subtract two *TDateTime* values to find the number of days between the two dates. The **DateStr** function returns the current date as a string variable in the format mm-dd-yy.

EXAMPLE: The following example prints the current date (assuming a date of May 22, 2000). The example also illustrates how the **Date** function can handle addition and subtraction calculations.

```
begin                                {Start of Main Programming code}
  writeln('The Date today is ',DateStr); {The date prints as 05-22-00}
  writeln(Date-1);                    {Prints 5/21/2000}
  writeln(Date);                       {Prints 5/22/2000}
  writeln(Date+1);                     {Prints 5/23/2000}
end;                                   {End of program}
```

DateToLong

LongToDate

LongToTime

TimeToLong

TimeToString

DwordToDate

TDateToDword

SYNTAX: **DateToLong**(Date: TDateTime): integer;
 LongToDate(Number: integer): TDateTime;
 LongToTime(Number: integer): TDateTime;
 TimeToLong(Date: TDateTime): integer;
 TimeToString(Date: TDateTime): string;
 DwordToDate(Number: integer): TDateTime;
 TDateToDword(Date: TDateTime): integer;

DESCRIPTION: The **DateToLong** function converts a *TDateTime* date into an integer number. The integer format is the format used by Ensign to store dates in the chart files.

The following numbers represent the years.

99 = 1999
100 = 2000
101 = 2001
102 = 2002
103 = 2003
104 = 2004 ... etc.

LongToDate converts a chart bar date to a TDateTime date.
LongToTime converts a chart bar time to a TDateTime time.
TimeToLong converts a TDateTime time into a chart bar time.
TimeToString converts a TDateTime time into string, which can be concatenated with other strings.

DwordToDate converts a double word value into a TDateTime date and time. The double word value represents the number of seconds since 1970. The double word variable is used for all Intraday chart time frames.
 Example: 1083283202 equals 4/30/2004 12:00:02 AM

TDateToDword converts a TDateTime into a double word value. The double word value represents the number of seconds since 1970. The double word variable is used for all Intraday chart time frames.
 Example: 4/30/2004 7:30:00 AM equals 1083310200

EXAMPLE: The following examples assumes a date of July 22, 2002. The integer date of 3 different years is printed. A bar chart date value of 1020722 is converted to a `TDateTime` date. A bar chart time value of 1515 is converted into a `TDateTime` time. The current time is converted to a chart bar time.

```
begin
  writeln('Last year ',DateToLong(Date-366));           {Prints 1010722}
  writeln('Today is ',DateToLong(Date));               {Prints 1020722}
  writeln('Next year ',DateToLong(Date+365));         {Prints 1030722}
  writeln(LongToDate(1020722));                        {Prints 07/22/2002}
  writeln(LongToTime(1515));                           {Prints 3:15:00 PM}
  writeln(TimeToString(LongToTime(1515)));           {Prints 3:15:00 PM}
  writeln(TimeToLong(Now));                            {Prints 1515}
end;
```

DateToStr

SYNTAX: **DateToStr**(Date: TDateTime): string;

DESCRIPTION: The **DateToStr** function converts a `TDateTime` date into a string. The string representation of a date value can be used for print and display purposes.

EXAMPLE: The following example assumes a current date of July 22, 2002. The example prints the date of 7 days ago.

```
begin
  writeln('Last week the Date was ', DateToStr(Date-7));
  {Date prints as 7/15/2002}
end;
```

DateToString

SYNTAX: **DateToString**(Date: integer): string;

DESCRIPTION: The **DateToString** function converts an integer date into a string. Example, the integer date of 1021126 is returned as a string value of '11-26-02'. NOTE: The date returned from the **Bar**(eDate,index) function is an integer number.

EXAMPLE: The following example opens an IBM daily chart and prints the date of the first bar of the chart (in both the integer and string formats). The example assumes that the first date of the chart is May 26, 1993.

```
var
  iDate: integer;           {Start of Variable Declarations}
  sDate: string;           {Declares iDate as an integer}
begin
  Chart('IBM.D');         {Declares sDate as a string}
  iDate:= Bar(eDate,1);   {Start of Main Programming code}
  sDate:= DateToString(iDate); {Opens an IBM daily chart}
  writeln(iDate,' ',sDate); {Retrieves the date of the 1st bar}
                           {Converts the integer date into a string}
                           {Prints 930526 05-26-93}
end;                       {End of program}
```

DayOfWeek

SYNTAX: **DayOfWeek**([*Date*: TDateTime]): integer;

DESCRIPTION: The **DayOfWeek** function returns the day of the week of the specified date. The returned value is a number between 1 and 7 (where Sunday=1, Monday=2, Tuesday=3, Wednesday=4, Thursday=5, Friday=6, and Saturday=7). If the *Date* is omitted, then the current date is used.

EXAMPLE: The following example assumes a date of May 22, 2000 (which is a Monday).

```
begin
  writeln(DayOfWeek(Date));           {Prints a 2 (Monday)}
  writeln(DayOfWeek(Date+1));        {Prints a 3 (Tuesday)}
end;
```

Dec Inc

SYNTAX: **Dec**(*Number*: integer);
 Inc(*Number*: integer);

DESCRIPTION: The **Dec** statement decrements the value of *Number* by one. The **Inc** statement is used to increment the value of *Number* by one. These statement can be used to increment or decrement variables that might be used as counters or pointers.

EXAMPLE: The following example sets a variable equal to 10, and then uses the **Dec** and **Inc** statements to change the value of *Count*. The **Dec** and **Inc** statements are quicker and more efficient than using the following statements (*Count:=Count-1;* or *Count:=Count+1;*).

```
var           {Start of Variable Declarations}
  Count: integer; {Declares Count as an integer variable}
begin       {Start of Main Programming code}
  Count := 10; {Set Count equal to 10}
  dec(Count); {Count now equals 9}
  writeln(Count); {Prints 9 }
  inc(Count); {Count now equals 10 again}
  writeln(Count); {Prints 10}
end;        {End of program}
```

DecodeDate

SYNTAX: **DecodeDate**(*Date*: TDateTime, var *Year*, var *Month*, var *Day*: integer);

DESCRIPTION: The **DecodeDate** statement is passed a *Date*, and then returns the *Year*, *Month*, and *Day* in separate variables. This allows a date to be quickly broken down into its separate parts. If the supplied *Date* value is less than or equal to zero, then the returned *Year*, *Month*, and *Day* values are all set to zero. The *Year* will contain a number between 1900 and 2099, etc. *Month* will contain a value between 1 and 12. Valid *Day* values are 1 through 28, 29, 30, or 31 (depending on the Month).

PARAMETERS:

Date: This value is passed to the **DecodeDate** statement. A *TDateTime* variable type is expected. This date value is then broken down into the *Year*, *Month*, and *Day* values.

Year: This variable receives back the value of the Year.
Month: This variable receives back the value of the Month.
Day: This variable receives back the value of the Day.

EXAMPLE: The following example is passed the current Date. The Year, Month, and Day are returned. The values are then printed.

```
var                                {Start of Variable Declarations}
  Yr, Mth, Dy: integer;           {Declares 3 variables as integers}
begin                              {Start of Main Programming code}
  DecodeDate(Date, Yr, Mth, Dy);  {Decodes the current Date}
  writeln(Date, ' ', Yr, ' ', Mth, ' ', Dy); {Prints each value}
end;                               {End of program}
```

DecodeTime

SYNTAX: **DecodeTime**(*Time*: TDateTime, var *Hour*, var *Minute*, var *Second*, var *MilliSec*: integer);

DESCRIPTION: **DecodeTime** is passed a *Time* value, and then returns the Hour, Minute, Second, and Milli-Second values corresponding to the specified *Time*. Pass **Now** as the *Time* parameter to break down the current time into its many parts.

PARAMETERS:

Time: This value is passed to the **DecodeTime** statement. A *TDateTime* variable type is expected. This time value is then broken down into the Hour, Minute, Second, and MilliSec values.
Hour: This variable receives the value of the hour (in 24 hour format, example 3:00 pm= 15)
Minute: This variable receives the value of the minute.
Second: This variable receives the value of the seconds.
MilliSec: This variable receives the value of the 1000th of a second.

EXAMPLE: This example prints the current time, and then decodes it into its individual parts.

```
var                                {Start of Variable Declarations}
  Hr, Min, Sec, Mil: integer;      {Declares 4 variables as integers}
begin                              {Start of Main Programming code}
  writeln(Time);                  {Prints the current time}
  DecodeTime(Now, Hr, Min, Sec, Mil); {Decodes the current time}
  writeln(Hr, ' ', Min, ' ', Sec, ' ', Mil); {Prints the individual time values}
end;                               {End of program}
```

Delete

SYNTAX: **Delete**(var *Text*: string, *Start*: integer, *Count*: integer);

DESCRIPTION: The **Delete** function is used to delete characters from a given *Text* string. The *Text* is passed to the function. The returned *Text* value does not include the deleted characters starting at the *Start* character, and ending at the *Start* plus *Count* character.

PARAMETERS:

Text: The *Text* parameter is the Text string that will be deleted from, and then returned.

Start: Specifies the starting character position in *Text* to start deleting.
Count: Specifies how many characters to delete, from the starting character.

EXAMPLE: The following example deletes 5 characters from the supplied text, starting at the 10th character position. The characters 'very' are deleted from the text.

```
var                                {Start of Variable Declarations}
  Text: string;                    {Text is declared as a string}
begin                              {Start of Main Programming code}
  Text:= 'This is a very big Move.'; {Text is assigned a value}
  writeln(Text);                   {Print original value of Text}
  Delete(Text, 10, 5);             {5 characters are deleted from Text}
  writeln(Text);                   {Print new value of Text}
end;                                {End of program}
```

DeleteBar

SYNTAX: **DeleteBar**(*Index*: integer): boolean;

DESCRIPTION: The **DeleteBar** function is used to delete a bar from a chart. Use **ChartRefresh**(True) to cause a chart to redraw, and reflect the change, after a bar has been deleted.

PARAMETERS:

Index: *Index* is the chart bar position (between 1 and the number of bars on the chart). If *Index* is less than or equal to zero, the function will use *Index* as an offset from the last bar on the chart (BarEnd). If *Index* is out of range, the function will return a *False* value, otherwise it will return *True*.

EXAMPLE: The following example opens an IBM daily chart and then deletes the 100th bar of the chart and the 5th bar from the end of the chart.

```
begin                              {Start of Main Programming code}
  Chart('IBM.D');                  {Opens a daily IBM chart}
  DeleteBar(100);                   {Deletes the 100th bar of the chart data}
  DeleteBar(-5);                    {Deletes the 5th bar from the end of the chart}
  ChartRefresh(True);               {Redraws the chart, without the deleted bars}
end;                                {End of program}
```

DeleteData

SYNTAX: **DeleteData**([*Feed*: integer]): boolean;

DESCRIPTION: The **DeleteData** function is used to delete a symbol from the Ensign quote pages. You must use the **Find** command before using the **DeleteData** function. The **Find** command will find the quote record and set a pointer to the quote record. Once the record is found, then it can be deleted. The **DeleteData** function will return a *True* value if the delete operation is successful, otherwise it will return a *False* value. When a quote record is deleted, the next record in the market group will become the active record, and the pointer will point to that record.

PARAMETERS:

Feed: *Feed* is one of these predefined constants. The default is the value assigned to the FEED global variable.

eFXCM	eIB	eSignal	eIQFeed	eNinja	eOpenECry
eTraderBytes	eTransAct	eGlobal	eDBFX	eATCBrokers	eCustom

EXAMPLE: The following example finds the IBM symbol in the Stock Market Group, and then deletes the quote record.

```
begin
  if Find(eSignal,'IBM') then DeleteData(eSignal);
end;
```

DimArray

SYNTAX: **DimArray**(UpperLimit: integer);

DESCRIPTION: The **DimArray** statement is used to initialize the *vArray* global array, and to set the upper boundary for the array. *vArray* is a single dimension variant array with a lower boundary of zero. A variant array can hold values of any variable type. **DimArray** redimensions *vArray* and sets the *UpperLimit* boundary. *vArray* must be dimensioned before it is used. New elements added to the array by redimensioning will be initialized to a value of zero. Previous elements will not be initialized to zero. See the documentation on *vArray* for details on array usage.

PARAMETER:

UpperLimit: *UpperLimit* specifies the maximum capacity of the array.
vArray will have a range of 0 to *UpperLimit*.

EXAMPLE: The following example dimensions *vArray* will an *UpperLimit* of 10. The array is filled with random numbers and then printed.

```
var                                     {Start of Variable Declarations}
  n: integer;                           {Declares n as an integer}
begin                                   {Start of Main Programming code}
  DimArray(10);                          {Dimensions vArray}
  for n := 0 to 10 do SetArray(n, Random(100)); {Assigns random numbers}
  for n := 0 to 10 do writeln(vArray(n));    {Prints the elements}
end;                                     {End of program}
```

Div

SYNTAX: (*Number1* **Div** *Number2*):integer

DESCRIPTION: The **Div** statement is a math operation used to perform Integer Division. *Number1* is divided by *Number2*. The result is truncated and does not include a remainder. The answer is always an integer value (no decimals).

EXAMPLE:

```
var
  A,B: integer;
begin
  A := 80.75 Div 19.88 ; {Assigns A the result of 80.75 divided by 19.88}
  B := 5 Div 2 ;        {Assigns B the result of 5 divided by 2}
  writeln(A, ' ',B);    {Prints the values of A and B (A=4 and B=2)}
end;
```


dlgColor dlgColor2

SYNTAX: **dlgColor**(color: integer): integer;
 dlgColor2.Execute(frmMain.Handle): boolean;

DESCRIPTION: The **dlgColor** command is used to open Ensign's hexagon Color dialog window. Colors can be designed and selected from the color window. After selecting a color and closing the color window, a color value is returned. The default color selected on the dialog window is the color parameter passed to the function.

The **dlgColor2** is the Microsoft Window color dialog component. Use its Execute method to display the form, and its Color property to return the color selected.

EXAMPLE: The following example opens the Color dialog window with Red as the default. After selecting a color and closing the color window, the color value returned is assigned to a variable named *NewColor*. An IBM daily chart is then opened, and the chart bars are changed to the new color. Accessing the **dlgColor2** dialog is also demonstrated.

```
uses
  graphics, dialogs, forms, classes, controls;    {include these libraries}

var
  i,NewColor: integer;                            {Start of Variable Declarations}
                                                {Declares two variables as integers}

begin
  NewColor := dlgColor(clRed);                    {Start of Main Programming code}
  Chart('IBM.D');                                {Assign the color selected to NewColor}
  for i:= 1 to BarEnd do SetBar(eColor,i,NewColor); {Opens an IBM daily chart}
  ChartRefresh(True);                             {Change bar colors}
  if dlgColor2.Execute(frmMain.handle) then       {Redraw chart with new colors}
    btnLayout.Font.Color := dlgColor2.Color;     {open the standard color dialog form}
    btnLayout.Font.Color := dlgColor2.Color;     {read the dialog's selection}
end;                                              {End of program}
```

dlgFont

SYNTAX: **dlgFont.Execute**: boolean;

DESCRIPTION: The **dlgFont.Execute** command is used to open a Microsoft Windows Font selection window. This window is used to select a Font, Font Color, Font Size, and Font Style. After making Font selections and closing the dialog box, a font value will be returned in the 'Font' property. The new font can then be used to change the font of the Form in the program. Use the **dlgFont.Font** property to assign the font selections to a form. The function returns **True** if the **OK** button is clicked. The function returns **False** if the **Cancel** button is clicked.

EXAMPLE: The following example opens an IBM daily chart, then opens the Windows Font selection window. Font selections can be made from the font window. After closing the Font selection window, the selected font is applied to the chart (the active form).

```
var
  Form1: TForm;                                  {Start of Variable declarations}
                                                {Form1 is declared as a TForm variable}

begin
  Chart('IBM.D');                                {Start of Main programming code}
  Form1 := Screen.ActiveForm;                    {Open an IBM daily chart}
  dlgFont.Execute ;                              {Assign pointer from the Chart to Form1}
  Form1.Font := dlgFont.Font ;                  {Open the Font selection window}
                                                {Assign Font selections to the Chart}
```

end;

dlgOpen dlgSave

SYNTAX: **dlgOpen**: boolean;
 dlgSave: boolean;

DESCRIPTION: The **dlgOpen.Execute** command displays a Microsoft Windows dialog box for selecting and opening files. This box can be used if your ESPL program requires you to select and open a file from the hard disk. When the dialog box is displayed, browse for and select a file and then click the **Open** button. The selected file name is stored in the *FileName* property. Both functions return True if the **OK** button is clicked. The functions returns False if the **Cancel** button is clicked.

The **dlgSave.Execute** command displays a similar dialog box, which is used to Save files. The sub-directory and filename can be selected for the save destination.

PROPERTIES: Properties can be read and set by appending the Property name after the **dlgOpen** statement.

DefaultExt: Specifies a file extension that is appended automatically to the selected filename, unless the file already has an extension.
FileName: Contains the name of the file that is selected from the Open dialog box.
Filter: Determines the File mask available in the Dialog. Only files of the specified type will be displayed in the Dialog.

Example: `dlgOpen.Filter := 'Text Files | *.TXT'`; will only list .TXT files.
Separate the description and file type with a '|' character.

InitialDir: A string that specifies the Initial Directory to Open a File from. Example: `dlgOpen.InitialDir := 'C:\ENSGN'`;
Make sure that you clear the *FileName* variable before calling the .Execute command, otherwise the *FileName* will override the *InitialDir*.

Title: Specifies a Title Caption in the Open dialog box. Example: `dlgOpen.Title := 'Select a File'`;

EXAMPLE: The following example displays an Open File dialog box. The dialog box allows you to browse any sub-directory for a particular file. The file is then loaded into a String list and then printed to the output window. A Save File dialog box is then opened, and the String List is saved to the selected file destination.

```
begin
    Output(eClear);
    dlgOpen.Title := 'Ensign Open File';
    dlgOpen.FileName := '';
    dlgOpen.InitialDir := 'C:\ENSGN';
    dlgOpen.Execute;
    writeln(dlgOpen.FileName);
    writeln();
    sList.LoadFromFile(dlgOpen.FileName);
    writeln(sList.Text);

    dlgSave.Title := 'Ensign Save File';
    dlgOpen.FileName := '';
    dlgSave.InitialDir := 'C:\';
    dlgSave.Execute;
    writeln(dlgSave.FileName);
    sList.SaveToFile(dlgSave.FileName);
end;
```

{Start of Main programming code}
{Clear the output window}
{Specifies a Title for the Dialog Box}
{Clear the FileName}
{Specifies the Directory to start in}
{Displays the Open File Dialog box}
{Prints FileName in the output window}
{Print blank line in the output window}
{Load Text file into a list}
{Print the file to the output window}

{Specifies a Title Save File Box}
{Clear the FileName}
{Specifies the Save default directory}
{Opens the Save File Dialog box}
{Prints FileName in the output window}
{Saves List to the selected destination}
{End of program}

dlgPrint dlgPrinterSetup

SYNTAX: **dlgPrint.Execute**: boolean;
 dlgPrinterSetup.Execute;

DESCRIPTION: The **dlgPrint.Execute** command displays a Microsoft Windows dialog box for Printer selection and number of copies. The function returns **True** if the **OK** button is clicked, or **False** if the **Cancel** button is clicked. The **dlgPrinterSetup.Execute** command displays a Printer control box. Both of these dialog boxes can be used if your ESPL program requires some Printer control.

PROPERTIES: The *Copies* property can be set by appending it to the **dlgPrint** statement.

Copies: Specifies the number of Print copies to make. See the example.

EXAMPLE: The following example opens a Print dialog box, then returns the number of copies. The program then opens a PrinterSetup dialog box.

```
begin
  dlgPrint.Execute;
  writeln(dlgPrint.Copies);
  dlgPrinterSetup.Execute;
end;
```

Download

SYNTAX: **Download**(*ChartSymbol*: string, [*FileName*: string, *Quantity*: integer, *BidAsk*: boolean]);

DESCRIPTION: The **Download** statement is used to download chart data directly into an ASCII text file rather than into an Ensign chart file. The *ChartSymbol* specifies the chart data to download (example: 'IBM.D'). The *FileName* specifies the path and name of the ASCII text file where the data will be saved. The *Quantity* parameter specifies the amount of chart data to download. The *BidAsk* parameter allows you to optionally include the Bid and Ask prices when downloading Tick data. The ASCII chart data file can be loaded into spread sheets or other programs for analysis.

If you are an eSignal data-feed user, the downloaded data is saved in a comma delimited file containing the following price fields.

Daily Charts:	Date, Open, High, Low, Last, Volume
Intra-day Charts:	Date, TimeStamp, Open, High, Low, Last, Volume
Tick Charts:	Date, TimeStamp, Tick, TickVolume
Tick Charts with Bid/Ask:	Date, TimeStamp, Bid Price, Bid Size, Ask Price, Ask Size

The Date format in the file is:	mm/dd/yy
The Intra-day TimeStamp format is:	HH:MM
The Tick chart TimeStamp format is:	HH:MM:SS

If you use the Ensign Internet version then downloaded data is saved as follows, and is delimited by | characters.

Tick Charts: DATE | TRADE | TRADE_SIZE | VOLUME |

2006-08-22 17:28:00 | 1301.75 | 1 | 2054 |
2006-08-22 17:28:00 | 1301.75 | 1 | 2053 |

Daily Chart: CLOSE | DATE | HIGH | LOW | OPEN | OPENINTEREST | VOLUME |

1302.00|2006-08-22 00:00:00|1305.75|1297.25|1301.50|0|771246|

Intraday Charts: DATE | OPEN | HIGH | LOW | CLOSE | BID | ASK | VOLUME | INTERVAL_VOLUME |
2006-08-21 00:00:00|1304.25|1304.50|1304.25|1304.50|1304.25|1304.50|8241|112|

PARAMETERS

ChartSymbol: The *ChartSymbol* parameter specifies the chart to download (example: 'IBM.D'). If the chart time frame is missing then .0 tick data will be the default.

FileName: The *FileName* parameter specifies the path and file where the chart data will save. If the *FileName* is omitted, the default *FileName* will be the C:\ENSIGN folder, and the *ChartSymbol* with a file extension of .TXT (example: C:\ENSIGN\IBM.TXT).

Quantity: The *Quantity* parameter specifies the amount of chart data to download. The *Quantity* parameter can have a value of 0 to 6. These numbers will correspond to the chart Refresh pop-up menu selections for different Time Frames. For example, a Daily chart would download the following amounts of data.

0= 1 day
1= 2 days
2= 1 week
3= 1 month
4= 6 months
5= 2 years
6= Maximum

BidAskNote: This is only available for eSignal data-feed users. False (the default) to only include the actual Trade ticks. Set the value to to include Bid and Ask prices in the download of Tick by Tick chart data. True Set this parameter to :

EXAMPLE: The following program uses the **Download** command to download data for an IBM 5-minute chart. The **Finished** command is used to detect the completion of the download. The ASCII chart data is saved to the C:\ENSIGN\IBM.TXT file, and then printed in the output window.

```
begin
  Output (eClear);
  Download('IBM.5', sPath + 'IBM.TXT', 3, False);
  if Finished(60) then Output (eLoad, 'IBM.TXT');
end;
```

DownloadData

SYNTAX: **DownloadData**(*List*: integer, [*Flag*: boolean]);

DESCRIPTION: The **DownloadData** command is used to 'Download Charts' using the Internet Services window. The Internet Services window displays a list of symbols that can be downloaded, with various Time Frames. NOTE: The Internet Services window must be open before executing the **DownloadData** command. Use the **btnInternet.click**

command to open the Internet Services window beforehand. Set the *Flag* parameter equal to `True` to download Intra-day charts.

PARAMETERS:

List: Specifies the Chart symbol *Tab* list for which to download chart data. Enter a number from 1 to 9.

Flag: Enter `True` to download Intra-day charts. Enter `False` (or leave blank) to download Daily, Weekly, Monthly entries from the list.

EXAMPLE: The following program opens the Internet Services window, and then downloads chart data for *Tab* list 1. Click ESPL button 5 to run the program. Make sure that the computer has an active Internet connection. NOTE: The symbol list must be entered manually before using this command.

```
begin
  if ESPL=5 then begin
    btnInternet.click;      // Open the Internet Services window
    DownloadData(1,True);   // Download Intra-day Charts from Tab List 1
  end;
end;
```

DrawPhase

SYNTAX: `DrawPhase: boolean;`

DESCRIPTION: The *DrawPhase* global variable can be used to determine if an ESPL Draw Tool is currently selected (and being applied or adjusted on the chart). The value of *DrawPhase* will be `False` if the tool is in an active selected state. The value will be `True` if the tool is no longer in an active selected state.

EXAMPLE: The following program draws a triangle connecting three marked points on the chart, using a User-Defined draw tool (ESPL=500). The three points are labeled with their prices, and a red center line divides the middle. The *DrawPhase* variable is used to prevent the tool from performing the **TextOut** commands until the three points have been selected and the tool is no longer in an active selected state.

```
uses
  Graphics;

procedure DrawTriangle;           {DrawTriangle procedure declared}
begin
  SetPen(clBlue,1,eSolid);
  if DrawPhase then begin        {Print TextOut if DrawPhase is True}
    TextOut(Pt1X+10,Pt1Y,FormatPrice(YtoPrice(Pt1Y)));
    TextOut(Pt2X+10,Pt2Y,FormatPrice(YtoPrice(Pt2Y)));
    TextOut(Pt3X+10,Pt3Y,FormatPrice(YtoPrice(Pt3Y)));
  end;
  MoveToLineTo(Pt1X, Pt1Y, Pt2X,Pt2Y);  {Connect the three points}
  MoveToLineTo(Pt2X, Pt2Y, Pt3X,Pt3Y);
  MoveToLineTo(Pt3X, Pt3Y, Pt1X,Pt1Y);
  SetPen(clRed,1,eSolid);
  MoveToLineTo(Pt1X,Pt1Y,(Pt2X+Pt3X)/2,(Pt2Y+Pt3Y)/2);  {Draw Center Line}
end;

begin                             {Start of Main Programming code}
  if ESPL=500 then DrawTriangle;   {Call the DrawTriangle procedure}
end;                               {End of program}
```

Drawing

SYNTAX: `Drawing: boolean;`

DESCRIPTION: The *Drawing* global variable can be used to determine if a Draw Tool is currently selected or being drawn manually. The value of *Drawing* will be `True` once a draw tool is create or selected, and `False` when the tool is finished and the construction boxes have been removed.

Email

SYNTAX: **Email**(*Subject, SenderAddress, User ID, Password, FileAttachment, RecipientAddress*
 [, *RecipientAddress2,...etc.*] : string);

DESCRIPTION: Use the **Email** command to send e-mails with the ESPL language. This allows some users to e-mail their pager with a message. The main content of the e-mail should be loaded into the *sList* global string list before using the **Email** command. The following program loads the *sList* string list with a message and then sends an e-mail.

WARNING: If the e-mail addresses are not entered properly (with actual addresses), the Ensign program can potentially abort when the e-mail attempts to send.

PARAMETERS:

Subject: Specifies the text that will be included in an E-mail's subject line.
SenderAddress: Specifies the Sender's E-mail address.
User ID: Specifies the E-mail account name for authentication. If blank, then no authentication will be attempted.
Password: Specifies the E-mail account password. If blank, then no authentication will be attempted.
FileAttachment: Specify an optional file attachment. The file will be attached to the e-mail.
RecipientAddress: Specifies the Recipient E-mail address that will receive the e-mail. Multiple recipient addresses can be entered (one after another).

EXAMPLE: The following program clears the *sList* global string list, then adds some text into *sList* (this is used as the e-mail contents). The e-mail is then sent with a subject line of 'Hello'. No file attachment is specified.

```
begin
  sList.Clear;
  sList.Add('This is the contents of the e-mail. Have a nice day. ');
  Email('Hello', 'Me@MyAddress.com', 'Me@MyAddress.com',
        'mypassword', '', 'You@YourAddress.com' );
end;
```

EmailForm

SYNTAX: **EmailForm**(*Box1, Box2, Box3, Box4:* boolean, [*OtherAddress, Subject, FileAttachment:* string, *User ID,*
*Password:*string, *TabIndex:* integer]);

DESCRIPTION: Use the **EmailForm** command to e-mail Charts and other screen Images utilizing the Internet Services E-mail window. The **EmailForm** command can be used to send e-mails and file attachments to clients and other traders. The Message Text of the e-mail should be loaded into the *sList* global string list before using the **EmailForm** command. The Internet Services E-mail form should be set-up manually by clicking on the **Internet Services** button and selecting the 'E-mail' tab before using this ESPL command. Example: To e-mail a chart, make sure that the desired chart has the 'Focus'

and then use the **mnuEmailImage.Click;** command to activate the e-mail feature. Then use the **EmailForm** command to send the e-mail.

PARAMETERS:

Box1: Enter True or False. True places a check mark in the 'Ensign Support' box. False unchecks the box.
Box2: Enter True or False. True places a check mark in the 'Ensign Software' box. False unchecks the box.
Box3: Enter True or False. True places a check mark in the 'Other' box. False unchecks the box.
Box4: Enter True or False. True places a check mark in the 'List' box. False unchecks the box.
OtherAddress: This address is used when Box3 is checked. The address shows on the form by the Other checkbox.
Subject: Specifies an optional change to the text that will be included in an E-mail's subject line.
FileAttachment: Specifies an optional change to the attachment file name. The file will be attached to the e-mail.
User ID: Specifies an optional change to the E-mail account name for authentication.
Password: Specifies an optional change to the E-mail account password for authentication.
TabIndex: Specifies the E-mail TAB on the Email Form to select. Example, enter 1 to select the 1st TAB on the list.
When the TabIndex is not provided then the form opens using the last TAB selected.

NOTE: The *Subject* and *FileAttachment* entries are automatically entered for you in the Internet Services E-mail window. The active window will be attached as a file attachment. However, you can optionally change them if necessary. Enter an empty string '' as the parameter entry if you don't want to change the default *OtherAddress*, *Subject*, *FileAttachment*, *User ID*, or *Password* values that will load with the Internet Services E-mail window. When *Use ID* and *Password* are blank, then no authentication will be attempted.

EXAMPLE: The following program clears the *sList* global string list, then adds some text into *sList* (this is used as the e-mail Message Text). An IBM Daily chart is opened and then e-mailed to 'Ensign Software'. The program assumes that the Internet Services E-mail window has been previously set-up with the proper e-mail settings.

```

begin
  sList.Clear;
  sList.Add('Here is an image of my IBM Daily Chart. ');
  Chart('IBM.D');           {this form has focus, and will be the image taken}
  ImageToFile('Ensign.png'); {save image to the Setup | Images harddisk path}
  EmailForm(False, True, False, False, '', 'Subject: My Chart',
    'C:\Ensign10\Images\Ensign.png', 'myUserID', 'myPassword');
  btnInternet.Click;       {close the Internet form opened by EmailForm}
end;

```

EmailFormTab

SYNTAX: **EmailFormTab**(*TabIndex: integer*);

DESCRIPTION: Use the **EmailFormTab** command to set the Tab on the Email Form. Example, enter 1 to select the 1st Tab on the Email form.

PARAMETERS:

TabIndex: Enter a number for the Tab to select on the Email Form.

EXAMPLE: The following program selects Tab 8.

```

begin
  EmailFormTab(8);
end;

```

EncodeDate

SYNTAX: **EncodeDate**(*Year, Month, Day: integer*): TDateTime;

DESCRIPTION: The **EncodeDate** function returns a TDateTime variable type from the values specified in the Year, Month, and Day parameters. The *Year* must be between 1 and 9999. The *Month* must be between 1 and 12. Valid *Day* values are 1 through 28, 29, 30, or 31, depending on the calendar month. Possible *Day* values for February are 1 through 28, or 1 through 29, depending on whether the specified *Year* is a leap year. If the provided values are not within the appropriate range, an error will occur. The result equals 1 plus the number of days between 12/30/1899 and the given date.

PARAMETERS:

Year: This variable specifies the Year.
Month: This variable specifies the Month.
Day: This variable specifies the Day of the Month.

EXAMPLE: The following example generates a TDateTime for the provided Year, Month, and Day. The date is then printed in the output window.

```

var                               {Start of Variable declarations}
  NewDate: TDateTime;             {NewDate is declared as a TDateTime}
begin                               {Start of Main Programming code}
  NewDate := EncodeDate(2002, 12, 25); {Encodes the Date of 12/25/2002 }
  writeln(NewDate);               {Print the date}
end;                               {End of program}

```


EncodeTime

SYNTAX: **EncodeTime**(*Hour, Minute, Second, MilliSec*: integer): TDateTime;

DESCRIPTION: **EncodeTime** creates a TDateTime variable from the provided *Hour, Minute, Second, and MilliSec* parameters. If the specified parameters are not within the appropriate range an error will occur.

The value returned by **EncodeTime** is a number between 0 and 1, and indicates the fractional part of a day represented by the specified time. The value 0 equals midnight, 0.5 equals noon, 0.75 equals 06:00 PM, etc.

PARAMETERS:

Hour: This variable specifies the value of the hour (in 24 hour format, example 03:00 PM = 15).
Minute: This variable specifies the minute (0 through 59).
Second: This variable specifies the second. (0 through 59).
MilliSec: This variable specifies the millisecond (1000th of a second).

EXAMPLE: This example encodes a time and then prints the time.

```
var                                     {Start of Variable Declarations}
  xTime: TDateTime;                     {Declares xTime as a TDateTime variable}
begin                                   {Start of Main Programming code}
  xTime := EncodeTime(15, 30, 0, 0);    {Encodes the time of 03:30:00 PM}
  writeln(xTime);                       {Prints the Time}
end;                                    {End of program}
```

Encrypt Decrypt Hash

SYNTAX: **Encrypt**(StringToEncrypt, Key: string): string;
Decrypt(StringToDecrypt, Key: string): string;
Hash(StringToHash: string): string;

DESCRIPTION:

Encrypt Pass a string to encrypt, and an encryption key. Function returns an encrypted string.
Decrypt Pass a string to decrypt, and an encryption key. Function returns the decrypted string.
Hash Pass a string to hash. Function returns a 32 character unique hash string.

EXAMPLE: The example encrypts and decrypts a string, and prints a hash string. These routines can be used to implement security for parameter files.

```
var
  s: string;
begin
  s := 'This is an example string';      {some string to encrypt}
  s := Encrypt( s, 'MySecretKey' );      {encrypt it with a key}
  writeln( s );                           {encrypted string looks like gibberish}
  s := Decrypt( s, 'MySecretKey' );      {decrypt string using same key}
  writeln( s );                           {string is back to original text}
```

```
writeln( Hash( s ));           {generate a hash code for a string}
end;
```

Exp

Ln

Log2

Log10

SYNTAX: **Exp**(x : real): real;
 Ln(x : real): real;
 Log2(x : real): real;
 Log10(x : real): real;

DESCRIPTION:

Exp calculates the Exponential of (x). The return value is **e** raised to the power of (x), where **e** is the base of the natural logarithms. The value of **e** is 2.7182818.

Ln calculates the Natural Logarithm of (x). Example: **Ln**(10) equals 2.30.

Log2 calculates the Log (base 2) of (x). Example: **Log2** (8) equals 3.

Log10 calculates the Log (base 10) of (x). Example: **Log10** (100) equals 2.

EXAMPLE: The following example calculates and prints the **Exp**, **Ln**, **Log2**, and **Log10** for various numbers.

```
var                                            {Start of Variable declarations}
  r1,r2,r3,r4: real;                         {Variables declared as Real}
begin                                         {Start of Main Programming code}
  OutPut (eClear);                         {Clear the output window}
  r1 := Exp(1);                             {calculate the Exp of 1 }
  r2 := Ln(10);                            {calculate the Ln of 10}
  r3 := Log2(8);                            {calculate the Log2 of 8 }
  r4 := Log10(100);                         {calculate the Log10 of 100}
  writeln(r1, ' ', r2);                     {print r1 and r2 }
  writeln(r3, ' ', r4);                     {print r3 and r4 }
end;                                         {End of program}
```

ExtractFileDrive

ExtractFileExt

ExtractFileName

ExtractFilePath

SYNTAX: **ExtractFileDrive**(const *FileName*: string): string;
 ExtractFileExt(const *FileName*: string): string;
 ExtractFileName(const *FileName*: string): string;
 ExtractFilePath(const *FileName*: string): string;

DESCRIPTION:

ExtractFileDrive returns the Drive portion from *FileName*.

ExtractFileExt returns the Extension portion from *FileName*.

ExtractFileName returns the FileName portion from FileName.
ExtractFilePath returns the Path portion from FileName.

EXAMPLE: The following example illustrates how to parse a filename and path into its separate parts. The **Application** command is used to determine the path and filename for the Ensign program. Normally **Application.ExeName** will return a value of C:\ENSIGN\ENSIGN.EXE.

```

var                                {Start of Variable declarations}
  drive, ext, name, path: string;   {Variables declared as Strings}
begin                               {Start of Main Programming code}
  drive:= ExtractFileDrive (Application.ExeName); {Extract Drive}
  ext  := ExtractFileExt (Application.ExeName);   {Extract Extension}
  name := ExtractFileName (Application.ExeName); {Extract FileName}
  path := ExtractFilePath (Application.ExeName);  {Extract Path}
  writeln (Application.ExeName);                 {prints C:\ENSIGN\ENSIGN.EXE }
  writeln (drive);                               {prints C: }
  writeln (ext);                                 {prints .EXE }
  writeln (name);                                {prints ENSIGN.EXE }
  writeln (path);                                {prints C:\ENSIGN\ }
end;                                           {End of program}

```

Filter

SYNTAX: **Filter**(*Field*: integer, *Min*, *Max*: real [[, *FieldN*: integer, *MinN*, *MaxN*: real],...): boolean;

DESCRIPTION: The **Filter** function is used to scan and filter quote prices based on the selected criteria. The prices are tested to see if they fall within a *Min* and *Max* value. For the currently decoded database record, **Filter** returns True when all the *Field* values are >= the *Min* value and <= the *Max* value. Multiple fields can be tested by repeating the parameters. If any test is False, then **Filter** returns a False value.

PARAMETERS:

Field: The *Field* parameter should be one of the following quote field specifiers.

eAsk	
eAskSize	
eAveVol	IQFeed only
eBeta	
eBid	
eBidSize	
eEstEPS	
eEPSPercent	
eExpiration	
eDailyHigh	if (High>0) and (Last>=High) and (High>Low) then Result:=0 else Result:=1;
eDailyLow	if (Low>0) and (Last<=Low) and (High>Low) then Result:=0 else Result:=1;
eDividend	
eDividendPercent	if Last=0 then Result:=0 else Result:=100*Dividend/Last;
eDown	

eDownNetOpen	
eDownPercentOpen	
eEarnings	
eEPS	
eEPSPercent	if Last=0 then Result:=0 else Result:=100*Earnest/Last;
eHigh	
eInstitution	IQFeed only
eInterest	
eLast	
eLow	
eNet	
eNetPercent	if (Last=0) or (Net=0) then Result:=0 else Result:=100*Net/(Last-Net);
eOpen	
eOpenNet	if (Last=0) or (YesterdayClose=0) then Result:=0 else Result:=Open-YesterdayClose;
ePercent	if (Last=0) or (High=Low) then Result:=0 else Result:=100*(Last-Low)/(High-Low);
ePERatio	
eSharesOut	
eTickTime	
eTickVolume	
eTotal	
eUnchanged	
eUp	
eUpNetOpen	if (Last=0) or (Open=0) or (Last<Open) then Result:=0 else Result:=Last - Open;
eUpPercentOpen	if (Last=0) or (Open=0) or (Last<Open) then Result:=0 else Result:=100*(Last-Open)/Open;
eVolume	
eYearlyHigh	if (H52>0) and (Last>=H52) then Result:=0 else Result:=1;
eYearlyHighDate	if (L52>0) and (Last<=L52) then Result:=0 else Result:=1;
eYearlyLow	
eYearlyLowDate	
eYesterday	
eYield	

Min: Enter a minimum value.

Max: Enter a maximum value.

NOTE: For eDailyHigh, eDailyLow, eYearlyHigh, and eYearlyLow, enter the *Min* and *Max* parameters as zeros. This will generate a True value when the Close is on the Daily or Yearly High or Low. For eExpiration the *Min* and *Max* values must be of a TDateTime type. It will find symbols that have expiration dates between the *Min* and *Max* dates (see example2 below).

EXAMPLE 1: The following example scans all the symbols in the Nasdaq market group. Any symbol that has a Last price between \$15 and \$16 dollars, and a Volume between 50,000 and 9,999,999 shares will be reported in the output window.

```
begin                                     {Start of Main Programming code}
  if Find(eSignal) then                   {Find the eSignal feed group}
    repeat                                 {Define a Repeat loop}
      if Filter(eLast,1500,1600,eVolume,50000,99999999) then
        writeln(Align(GetData(eSymbol),8,eLeft),Round(GetData(eLast,true)));
      until not Find(eNext);              {Repeat Loop until done}
    writeln('Done');                       {Print 'Done' in output window}
end;                                       {End of program}
```

EXAMPLE 2: The following example reports all Stock Options that start with the letters N - Q which have expired in the past 60 days.

```
begin                                     {Start of Main Programming code}
  if Find(eSignal) then                   {Find eSignal feed group}
    repeat                                 {Define a Repeat loop}
      if Filter(eExpiration, Now-60, Now) then writeln(GetData(eSymbol));
    until not Find(eNext);              {Repeat loop until done}
end;                                       {End of program}
```

Find FindMarket

SYNTAX: **Find**(*Feed*: integer [, *Symbol*: string]): boolean;
 Find(*Symbol*: string, [*Feed*: integer]): boolean;
 Find(eNext): boolean;
 Find(ePrior): boolean;
 Find(eChart): boolean;
 FindFeed(*Symbol*: string): integer;
 FindMarket(*Symbol*: string): integer;

DESCRIPTION:

Find: The **Find** function is used to locate a symbol in the quote pages. If the symbol is found, then the function returns a **True** value. The function returns a **False** value if the symbol is not found. The **Feed** is specified so that Ensign will know which market to find the symbol in. If a symbol is not specified, then **Find(Feed)** will return the first symbol in the market group. This is useful before performing a loop through all the symbols in the market group. After finding a symbol, use the **GetData** function to retrieve and process values from the found symbol.

The format of **Find(Symbol)** can be used to find the symbol and the optional **Feed** can be specified by the **FEED** variable. If **FEED = 0** or the 2nd parameter is a zero, then the feed will be looked up.

Find(eNext): Use this command to Find the next symbol record in the market group. The symbol's data is decoded for use by the **GetData** function. **Find(eNext)** is often used in a loop to process or test data for all symbols in a market group. A **False** value will be returned when the last symbol in the market group is reached.

Find(ePrior): Use this command to back-up and Find the previous symbol record in the feed group. A **False** value will be returned when the first symbol in the market group is reached.

Find(eChart): This command will Find the symbol record for the current active chart. This allows access to the current quote prices while performing a user-defined study on a chart. See the **TextOut** command for an example.

FindFeed: This command is used to find a symbol. The function returns a Zero value if the symbol is not found. The function returns an integer value which identifies the feed group. All feed groups are searched until the symbol is found.

FindMarket: Use the command to return the Market Group for the symbol, such as: eFuture.

NOTE: All uses of the **Find** function decode the symbol's data for use by the **GetData** function. For example, to retrieve the Bid price for a stock you would first **Find** the symbol, and then use **GetData(eBid)** to retrieve the price information.

PARAMETERS:

Feed: *Feed* is one of these predefined constants. The default is the value assigned to the FEED global variable.

eFXCM	eIB	eSignal	eIQFeed	eNinja	eOpenECry
eTraderBytes	eTransAct	eGlobal	eDBFX	eATCBrokers	eCustom

EXAMPLE: The following example clears the output window, and then locates the first symbol in the Nasdaq stock market group. A loop is then used to print all Symbols whose volume is greater than 5 million shares. The example, then Finds the IBM stock symbol in the eStock market group and prints its Bid price. The **FindMarket** function is then used to retrieve and print the Market Group for the OEX index symbol.

```
begin
  Output (eClear);
  if Find(eSignal) then
  repeat
    if GetData(eVolume) > 5000000 then writeln(GetData(eSymbol));
  until not Find(eNext);
  if Find(eSignal, 'IBM') then writeln('IBM Bid= ', GetData(eBid));
  writeln('OEX Market= ', FindMarket('OEX'));
end;
```

FindClose

FindFirst

FindNext

SYNTAX: **FindClose;**
FindFirst(Path: string): string;
FindNext: string;

DESCRIPTION:

FindFirst: This command is used to find a File on the computer's hard disk. The specified *Path* (which includes the filename) is searched. Wildcard characters can be used in the *Path* filename. The function will return the filename for the first file it finds that matches the *Path* filename. A null string value is returned if no file is found.

FindNext: This command returns the name of the next file that matches the *Path* filename, otherwise, it returns a null string. The **FindFirst** command must be used before using **FindNext**. After using the **FindFirst** command, you can then use **FindNext** multiple times to find subsequent files matching the *Path* filename criteria.

FindClose: This command must be used after using the **FindFirst** command. The **FindFirst** command allocates computer memory which is used by **FindNext**, and must be released by calling **FindClose**. Failure to call **FindClose** results in a memory leak (the computer memory that is used will not be released for other uses).

PARAMETERS:

Path: The *Path* parameter is used to specify the Directory and FileName to search. For example, 'C:\ENSIGN\HIST*.*' specifies all files in the C:\ENSIGN\HIST directory.

Wildcard Characters:

* An asterisk after a string will match any number of occurrences of that string followed by any characters. For example, to find all files that start with the letter 'B' you could use 'C:\ENSIGN\HIST\B*.*'

? A question mark matches any single character in that character position. For example, IB? would match IBM

EXAMPLE: The following example is used to print all the 5-minute intraday chart filenames that start with the letter 'S'. Intraday chart files are located in the \ENSIGN\TICK sub-directory. The **FindFirst** command is used to find the first file. A **While** loop is then used to loop through and **FindNext** the rest of the matching files. **FindClose** is used to release the computer memory when the program is done.

```
var
  Symbol: string;
begin
  Symbol := FindFirst('C:\ENSIGN\TICK\S*.5');
  while length(Symbol)>0 do
  begin
    writeln(Symbol);
    Symbol:=FindNext;
  end;
  FindClose;
end;
```

FindStudy FindStudyName

SYNTAX: **FindStudy**(*Study*: integer [, *Instance*: integer]): integer;
FindStudyName(*Study*: integer [, *Name*: string]): integer;

DESCRIPTION: The **FindStudy** command is used to find a study on a chart. The study values can then be retrieved by using the **GetStudy** command. The **FindStudy** command returns the study's object number.

FindStudyName searches the chart object list for the actual short name listed on the list.

Example: FindStudyName(eSto, 'STO 9,5'); will find a Stochastics study listed as 'STO 9,5' in the list.

Example: For DYO studies, include 'DYO:' before the short name in the list. FindStudyName(eDYO, 'DYO:Test One'); would find a DYO study listed as 'Test One' in the objects list.

Example: FindStudyName(eESPL, 'ESPL 100'); would find an ESPL study which passes 100 as the ESPL value. Do not use FindStudy(eESPL, 100); because the 2nd parameter is the instance, and not the ESPL value.

If the study is found, these functions will return the object number. A Zero value is returned if the study is not found. The study object number can then be used by the **GetStudy**, **SetStudy**, and **Remove** functions. Use the **FindWindow** or **Chart** command before using **FindStudy**, so that **FindStudy** will know which chart to find the study on.

PARAMETERS:

Study: The *Study* parameter should be one of the following predefined study specifiers:

eACC	eADX	eArn	eAsh	eASI	eATR
eAve	eBal	eDon	eDvg	eERG	eESPL
eBol	eCHI	eCCI	eCyc	eDYO	eHlo
eHull	eKel	eMOM	eMRg	eOsc	eOvr
ePAF	ePVI	ePVP	ePDA	ePar	ePAT
eReg	eROC	eRSI	eSMI	eSto	eTex
eTnd	eTrl	eTrx	eUlt	eUni	eVlt
eWlm	e3PB				

NOTE: See **AddStudy** for details.

eGrid - is used to get the handle for the Study Grid line object

eAlan	eAndrews	eArrow	eCircle	eCycle	eFibCycle
eFibonacci	eFibRuler	eGann	eGannCycle	eGannSquare	eLabel
eLevel	eLine	eLinear	eNote	eParallel	ePyrapoint
eRetrace	eScale	eSpeed	eSupport	eESPLTool	eWave3
eWave5					

NOTE: The *Study* parameter for a chart overlay is eOvr. Overlays can be accessed with the **Formation**, **Bar**, **ChartBar**, **Summation**, **Average**, **ExpAverage**, **Highest**, **Lowest**, **StdDev**, **Last**, **Open**, **High**, **Low**, **Volume** and **OpenInt** functions.

Instance: The *Instance* parameter is used if a chart contains multiple copies of the same study. For example, the RSI study could be applied three times on the same chart with different RSI settings. The *Instance* parameter is used to select the first (1), second (2), or third (3), etc. occurrence of the study object. A value of zero will also default to the first instance of the named study. The default is the 1st occurrence.

When the object is a Grid, the *Instance* parameter is the sub-window for the Grid object. Example, **FindStudy(eGrid,3)** will return the handle for the Grid object in sub-window #3. **GetStudy(handle,950)** can be used to return the sub-window number for a study for use in finding the sub-window's grid.

EXAMPLE: The following example assumes that an IBM daily chart is already open. It also assumes that the Relative Strength Index (RSI) study is displayed on the chart. The example finds the IBM chart window, locates the RSI study, and then prints the last ten RSI values.

```

var                                     {Start of Variable declarations}
  StudyHandle,i: integer;               {Variables declared as Integers}
  Value: real;                          {Value declared as a Real}
begin                                   {Start of Main programming code}
  FindWindow(eChart, 'IBM.D');          {Find the IBM chart}
  StudyHandle := FindStudy(eRSI);       {Find the RSI study}
  if StudyHandle > 0 then               {if found then continue}
  for i := BarEnd-9 to BarEnd do        {Loop through last 10 bars}
  begin                                  {Start of Loop code}
    Value := GetStudy(StudyHandle,1,i); {Retrieve the RSI value}
    writeln(Value);                    {Print the RSI value}
  end;                                  {End of Loop code}
end;

```



```

end;                                {End of program}

var                                  {Start of Variable declarations}
  StudyHandle,SubWin: integer;      {Variables declared as Integers}
begin                                {Start of Main programming code}
  FindWindow(eChart);              {Find a chart}
  StudyHandle := FindStudy(eRSI);   {Find the RSI study}
  SubWin := GetStudy(StudyHandle,950); {Which sub-window the RSI uses}
  StudyHandle := FindStudy(eGrid,SubWin); {Handle for the sub-window grid object}
end;                                {End of program}

```

FindWindow

SYNTAX: **FindWindow**(*Type*: integer [, *Name*: variant, *Page*: string]): integer;
FindWindow(eChart [, *ChartName*: string, *Instance*: integer]): integer;

DESCRIPTION: The **FindWindow** command is used to find an open window on the screen. The function returns the Window number for use by other functions which require a Window number. **FindWindow** returns a Zero value if the window is not found. The global Window variable is set to the window number.

PARAMETERS:

Type: The *Type* parameter should be one of the following predefined window constants:

eAccount	- Find the main Trading Account window
eChart	- Find a Chart window
eNews	- Find a News window
eOptimizer	- Find an Optimizer window
eOrderEntry	- Find an Order Entry window
eQuote	- Find a Quote window
eScanner	- Find a Chart Scanner window
eScript	- Find the ESPL Script Editor window
eSpreadsheet	- Find the Spreadsheet window
eText	- Find a TextBox window
eTrade	- Find a specific Trading Account window
eStack	- Find a Stack window

Name: For eQuote the *Name* is either the Market Group to find, or 'eCustom' plus a custom *Page* name.
For eText the *Name* is the Window Caption for the TextBox window.
For eSpreadsheet the *Name* is the text to match in the form's drop-down combo box.
When *Name* is omitted, the first window found for the specified *Type* will be reported.
Name is ignored when the *Type* is eAccount, eNews, eScript, and eTrade.

Page: *Page* is the name of a custom quote page. *Page* is only included when the *Type* is eQuote. *Page* should match the text in the Quote Page drop-down combo box next to the Custom button. For example, to locate a custom quote page window named 'Dow30' you would use the following command:
FindWindow(eQuote, eCustom, 'Dow30').

ChartName: When the *Type* is eChart, *ChartName* identifies a specific chart to find (and its Time Frame).
Example: FindWindow(eChart,'IBM.D');

Instance: If there are multiple charts opened of the same symbol, then an *Instance* parameter may be entered. For example, an *Instance* value of 3 would attempt to find the 3rd 'IBM.D' chart window that is open.
Example: FindWindow(eChart,'IBM.D', 3);

EXAMPLE: The following example assumes that an IBM daily chart and a Quote page (displaying the Nasdaq market group) are already displayed on the screen. The program finds the IBM window and Flashes the window caption twice. The program then finds the Nasdaq quote window and Flashes the window caption twice.

```
begin
  if FindWindow(eChart,'IBM.D') then begin
    Flash; Pause(1); Flash; Pause(1);
  end;
  if FindWindow(eQuote,eNasdaq) then begin
    Flash; Pause(1); Flash;
  end;
end;
```

Finished

SYNTAX: **Finished**(Seconds: integer): boolean;

DESCRIPTION: The **Finished** command is used to determine if the chart download is finished. The function pauses the ESPL program until the chart has been downloaded. The **Finished** function returns a **True** value when the chart has been downloaded. The ESPL program can then proceed to the next programming tasks, using complete chart data.

The function has a parameter specifying the number of *Seconds* to wait before proceeding (in case the chart fails to download within the specified time). The **Finished** function will return a **False** value if the time elapses before the chart has been downloaded.

eSignal Version: The **Finished** function can also be used after using the **Manager**(eOnLine) command. This causes the ESPL program to pause until the Turbo Data Manager program accepts and activates the Ensign symbols list. The **Finished** function will return a **True** value when the Data Manager has completed the OnLine initialization tasks. A **False** value will be returned if the specified *Seconds* elapse before the Data Manager is finished with the OnLine command.

The **Finished** function can also be used with the **HTTP** command. It will cause the program to wait until the HTTP request has been received.

PARAMETERS:

Seconds: The *Seconds* parameter specifies the number of seconds to wait before proceeding to the next ESPL command. The ESPL program will pause on this command until either the task completes or the time elapses.

EXAMPLE: The following example requests an IBM daily chart. The chart data will be downloaded from the *eSignal* internet chart servers. The **Finished** command will wait up to 60 seconds before proceeding. The program will print a message indicating if the chart was downloaded successfully, or not. The Data Manager is then given the OnLine command. The program waits up to 60 seconds for the Data Manager to complete the OnLine tasks.

```
begin
  Chart('IBM.D');                               {Open the IBM daily chart}
  if Finished(60) = True then                    {Wait for the chart download}
    writeln('IBM Chart downloaded. ');
  else
    writeln('IBM Chart failed to download. ');
  Feed := eSignal;
  Manager(eOnLine);                              {Put the Data Manager OnLine}
  if Finished(60) = True then                    {Wait until Initialized}
```

```

    writeln('Data Manager is On-Line.')
else
    writeln('Data Manager is not responding.');
```

end;

Also see Global Variable: **RefreshBusy: boolean;**

This variable will be True if there are charts being refreshed. It will be false if the chart refresh is finished for all charts that are open. Test the value of the this variable if you want to wait for the refresh to complete before continuing with further items.

Flash

SYNTAX: **Flash;**

DESCRIPTION: The **Flash** command is used to Flash the caption bar for a specified window. The caption bar will change from the Active state to an Inactive state each time the Flash command is used. Multiple calls to the Flash command will imitate a Flashing window. This can be useful to attract attention to the window if an alert condition has been reached. The window that Flashes is specified by the *Window* variable as set by the **FindWindow** or **Chart** functions.

EXAMPLE: The following program is run by clicking ESPL button 1. An IBM daily chart is opened and a **Timer** is started that causes the chart window to blink every second. The *ESPL* value is set to 10 by the Timer. This allows the program to know who called the program, and call the `BlinkWindow` procedure.

```

procedure BlinkWindow;
begin
    FindWindow(eChart, 'IBM.D');           {Find the IBM chart}
    Flash;                                 {Flash the chart caption}
end;

{***** Main Program *****)
begin
    if ESPL=1 then begin
        Timer(eStart,1,10);               {Start the 1 second Timer}
        Chart('IBM.D');                   {Open an IBM daily chart}
    end;
    if ESPL=10 then BlinkWindow;          {Call BlinkWindow every second}
end;

```

FloatToStr StrToFloat

SYNTAX: **FloatToStr**(*Value*: real): string;
StrToFloat(*Text*: string): real;

DESCRIPTION: **FloatToStr** converts the floating-point *Value* to its string representation. The conversion uses general number format with 15 significant digits. **StrToFloat** converts a text number into a floating-point number. The string must consist of an optional sign (+ or -), a string of digits with an optional decimal point, and an optional 'E' or 'e' followed by a signed integer. Any leading or trailing blanks in the string are ignored. Thousand separators and currency symbols are not allowed in the string. If the string doesn't contain a valid value, an error will occur.

EXAMPLE: The following example converts a floating-point number into a string. The example then converts a string text number into a floating-point value. The values are then printed.

```
var                                     {Start of Variable declarations}
  Value1, Value2: real;                 {Variables declared as Real}
  Text1, Text2: string;                {Variables declared as Strings}
begin                                   {Start of Main programming code}
  Value1 := 523.76;                    {Assigns a number to Value1}
  Text1 := '450.87';                   {Assigns some text to Text1}
  Text2 := FloatToStr(Value1);         {Converts Value1 to a String}
  Value2 := StrToFloat(Text1);         {Converts Text1 to a Real number}
  writeln(Text2, ' ', Value2);         {Prints the values}
end;                                    {End of program}
```

FloodFill

SYNTAX: **FloodFill**(*X, Y*: integer, *Color*: integer, *FillStyle*: integer);

DESCRIPTION: **FloodFill** is used to paint an enclosed area of the chart with the current brush pattern and color as specified by the **SetBrush** command.

PARAMETERS: **FloodFill** begins painting at the X,Y coordinate position and continues in all directions until a boundary is encountered. The *FillStyle* parameter determines the method in which the area is painted. If *FillStyle* is 1, the area fills until a border of the color specified by the *Color* parameter is encountered. If *FillStyle* is 0, the area fills as long as the color specified by the *Color* parameter is encountered (FillStyle=0 should be used to paint an area that has a multicolored border. A global ESPL variable named *Window* is used to specify which chart to paint. The *Window* variable can be manually assigned a window pointer value (if you have been keeping track of the window handles), or you can use the **FindWindow** or **Chart** functions to set the *Window* variable.

EXAMPLE: The following example opens an IBM daily chart, draws a rectangle with a blue border, then fills the rectangle with white vertical lines.

```
uses
  Graphics;
begin                                   {Start of Main programming code}
  Chart('IBM.D');                       {Open an IBM daily chart}
  SetPen(clBlue, 2, eSolid);             {Pen color=Blue, 2 pixel width, Style=Solid}
  Rectangle(50,10,250,110);             {Draw a blank rectangle with a blue border}
  SetBrush(clWhite, eVertical);         {Brush fill White, fill=Vertical lines}
  FloodFill(51,11,clBlack,0);          {Fills rectangle with white vertical lines}
end;                                    {End of program}
```

For

SYNTAX: **For** *Count* := *Start* **to** | **downto** *End* **do** command;

or

```
For Count := Start to | downto End do
begin
  {multiple command statements}
end;
```

DESCRIPTION: A **For** loop is used to execute a command (or block of commands) several times. The **For** loop's *Start* and *End* variables specify how many times to loop through the commands. Loops are used to scan through lists, to count items, and to increment values.

PARAMETERS:

Count: *Count* is a numeric variable used to count the loops as they occur.
Start: *Start* is the initial value assigned to *Count*.
End: *End* specifies the ending loop value (or where it will count to).
to: The '**to**' keyword specifies that the counting will increment +1 with each loop. The loop counts higher.
downto: The '**downto**' keyword specifies that the counting will decrement -1 with each loop. The loop counts down.

EXAMPLE: The 1st example loop counts from 1 to 100 and prints the value of the counter. The 2nd loop counts from 200 downto 100 and performs a series of sample calculations using multiple commands.

```
var                                {Start of Variable declarations}
  x,y,z: integer;                 {x,y, and z are declared as Integers}
begin                              {Start of Main programming code}
  for x :=1 to 100 do writeln(x);  {Count to 100, print the counter}
  for x :=200 downto 100 do begin  {Count from 200 downto 100}
    y:= Sqr(x);                   {y equals the Square of x }
    writeln(x, ' Squared = ',y);   {Prints x and y }
    z:= y - x;                     {z equals x Squared minus x}
    writeln(y, ' - ',x, ' = ',z);  {Prints y and z }
  end;                             {End of Loop command block}
end;                               {End of program}
```

ForceDirectories

SYNTAX: ForceDirectories(*DirectoryPath*: string);

DESCRIPTION: The ForceDirectories function creates all a new directory as specified in *DirectoryPath*, which must be a fully-qualified path name. If the directories given do not yet exist, ForceDirectories attempts to create them.

PARAMETER:

DirectoryPath: The *DirectoryPath* is a fully-qualified path name.

EXAMPLE: The following example uses the ForceDirectories function to create a series of folders. If none of the directories exist, they will be created, starting with the parent, 'mike, then onto the next and so forth.

```
begin
  ForceDirectories('c:\Ensign10\Backup\MyData');
end;
```

Format

SYNTAX: Format(*FormatString*: string, [*Value1*: real [, *Value2*, ... *Value7*:real]]): string;

DESCRIPTION: The **Format** function formats one or more numbers into a string representation. Formatting is controlled by the *FormatString*. Up to seven values may be formatted at the same time.

PARAMETERS:

Value1 through Value7: The numeric values of *Value1* can be any real number.

FormatString: Format Specifiers are used to format the *Value* parameter. Each *FormatString* contains text characters and Format Specifiers. Text characters are copied verbatim to the resulting string.

Format Specifiers have the following form: % [-] [width] [.prec] *Type*

The Format Specifier always begins with a %. The following characters may optionally follow the %, in this order:

- A left *Justification* indicator, [-]
- A *Width* specifier, [width]
- A decimal point *Precision* specifier, [.prec]
- A numeric conversion *Type* character

The default *Justification* is right-justified, resulting in added blanks in front of the value. If the format specifier contains a left-justification indicator (a '-' dash character preceding the width specifier), the result is left-justified by adding blanks after the value.

The *Width* specifier sets the minimum field width for a conversion. If the resulting string is shorter than the minimum field width, it is padded with blanks to increase the field width.

The *Precision* specifier indicates how many decimal places to include in the resulting string.

The *Type* character may be one of the following characters (e, f, g, n, or m):

- e: Scientific. Value is converted to a string of the form "-d.ddd...E+ddd". The resulting string starts with a minus sign if the number is negative, and one digit always precedes the decimal point. The total number of digits in the resulting string (including the one before the decimal point) is given by the precision specifier in the format string -- a default precision of 15 is assumed if no precision specifier is present. The "E" exponent character in the resulting string is always followed by a plus or minus sign and at least three digits.
- f: Fixed. Value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative. The number of digits after the decimal point is given by the precision specifier in the format string -- a default of 2 decimal digits is assumed if no precision specifier is present.
- g: General. Value is converted to the shortest possible decimal string using fixed or scientific format. The number of significant digits in the resulting string is given by the precision specifier in the format string -- a default precision of 15 is assumed if no precision specifier is present. Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision, and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format.
- n: Number. Value is converted to a string of the form "-d,ddd,ddd.ddd...". The "n" format corresponds to the "f" format, except that the resulting string contains thousand separators.
- m: Money. Value is converted to a string that represents a currency amount. The conversion is controlled by the CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator, and CurrencyDecimals global variables, all of which are initialized from the Currency Format in the International section of the Windows Control Panel. If the format string contains a precision specifier, it overrides the value given by the CurrencyDecimals global variable.

EXAMPLE: The first line below formats 6.6667 into a string equal to 6.67. The second line formats the numbers 8 and 6.6668 into the string '8 Units, Price= \$6.67'. NOTE: Each % starts a new Format Specifier (to be used for the next number in the parameter list).

```

begin                                     {Start of Main programming code}
  writeln(Format('%4.2f', [6.6667]));      {formats 6.6667 and prints 6.67}
  writeln(Format('%1.0f Units, P= %6.2m', [8.0,6.6668])); {8 Units, P= $6.67}
end;                                       {End of program}

```

FormatDateTime

SYNTAX: **FormatDateTime**(*Format*: string; *DateTime*: TDateTime): string;

DESCRIPTION: The **FormatDateTime** function is used to format the *DateTime* value using the specified *Format*. The following format specifiers can be used:

c	Displays the date using the format given by the ShortDateFormat global variable, followed by the time using the format given by the LongTimeFormat global variable. The time is not displayed if the fractional part of the DateTime value is zero.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat) using the strings given by the ShortDayNames global variable.
dddd	Displays the day as a full name (Sunday-Saturday) using the strings given by the LongDayNames global variable.
dddddd	Displays the date using the format given by the ShortDateFormat global variable.
dddddd	Displays the date using the format given by the LongDateFormat global variable.
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec) using the strings given by the ShortMonthNames global variable.
mmmm	Displays the month as a full name (January-December) using the strings given by the LongMonthNames global variable.
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
t	Displays the time using the format given by the ShortTimeFormat global variable.
tt	Displays the time using the format given by the LongTimeFormat global variable.
am/pm	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case.
a/p	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
ampm	Uses the 12-hour clock for the preceding h or hh specifier, and displays the contents of the TimeAMString global variable for any hour before noon, and the contents of the TimePMString global variable for any hour after noon.
/	Displays the date separator character given by the DateSeparator global variable.
:	Displays the time separator character given by the TimeSeparator global variable.
'xx'	Characters enclosed in single quotes are displayed as-is, and do not affect formatting.

EXAMPLE: The following example show 4 different ways to format the Date and Time values using the **Now** command (assuming the date is 12-25-2000 at noon).

```

begin                                     {Start of Main Programming code}
  writeln(FormatDateTime('c', Now));      {Prints 12/25/2000 12:00:00 PM }

```

```

writeln(FormatDateTime('d',Now));           {Prints 25 }
writeln(FormatDateTime('ddd',Now));         {Prints Mon }
writeln(FormatDateTime('yyyy',Now));        {Prints 2000 }
end;                                         {End of program}

```

Formation

SYNTAX: **Formation**(*Type*: integer, *Index*: integer [, *Count*: integer, *Dataset*: integer]): integer;

DESCRIPTION: The **Formation** function determines if a particular chart bar formation exists at the indicated bar location. The function can optionally find formations in chart overlay data by specifying the *Dataset* to use. The overlay's object number can be used as the *Dataset* parameter, or the values 1, 2, 3 ... may be used, indicating the 1st, 2nd, or 3rd ... overlay. The overlay object number can be obtained with the **FindStudy** function.

PARAMETERS:

Type: The *Type* parameter is used to identify the chart formation to search for. The parameter may be one of the following predefined constants:

eCandlestick	eGap	eGapOpen	eHook	eIsland	eKeyReversal
eLedge	eOutside	ePivot	ePivotClose	ePivotRange	eRangeSize
eTrend	eTurningPoint	eVolumeSize			

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart. If *Index* is less than or equal to zero, the function will use *index* as an offset from the last bar on the chart. If *Index* is out of range, the function will return zero. Both the host and the overlay use the same indexing.

Count: *Count* is only used by ePivotClose and ePivotRange. *Count* can be omitted for all others.

DataSet: The *DataSet* parameter is an optional object number for an overlay data set. The chart's bars are used by default.

Chart Formation Descriptions

Candlestick: Candlestick returns 1 for a bullish candlestick bar, 2 for a bearish candlestick bar, and 0 otherwise.
A bullish candlestick bar has a close above its open.
A bearish candlestick bar has a close below its open.

Gap: Gap returns 1 for bullish gaps, 2 for bearish gaps, and 0 otherwise.
A bullish gap has a low that is higher than the high of the previous bar.
A bearish gap has a high that is lower than the low of the previous bar.

Gap Open: Gap Open returns 1 for bullish gaps, 2 for bearish gaps, and 0 otherwise.
A bullish gap has an open that is higher than the high of the previous bar.
A bearish gap has an open that is lower than the low of the previous bar.

Hook: Hook returns 1 for a bullish hook bar, 2 for a bearish hook bar, and 0 otherwise.
A bullish hook bar is preceded by two negative nets, and followed by a positive net.
A bearish hook bar is preceded by two positive nets, and followed by a negative net.

Island: Island returns 1 for bullish Islands, 2 for bearish Islands, and 0 otherwise.
A bullish Island has a high that is lower than the low of the bar on either side.
A bearish Island has a low that is higher than the high of the bar on either side.

KeyReversal: KeyReversal returns 1 for a bullish key reversal bar, 2 for a bearish key reversal bar, and 0 otherwise.
 A bullish key reversal bar meets these conditions:
 1. The bar's close is in the upper 20% of its range.
 2. The bar's close is above 60% of the previous bar's range.
 3. The previous bar's close is in the lower 40% of its range.
 4. The previous bar's low was lower than the low of the previous bar.
 A bearish key reversal bar is the inverse formation of the bullish key reversal bar.

Ledge: Ledge returns 1 when the bar's close is equal to the previous bar's close, and 0 otherwise.

Outside/Inside: Outside/Inside returns 1 for outside range bars, 2 for inside ranges bars, and 0 otherwise.
 Outside range bars have a higher high and a lower low than the previous bar.
 Inside range bars have a lower high and a higher low than the previous bar.

Pivot: Pivot returns 1 for a bullish pivot bar, 2 for a bearish pivot bar, and 0 otherwise.
 A bullish pivot bar has a close that is lower than the close of the bar on either side.
 A bearish pivot bar has a close that is higher than the close of the bar on either side.

PivotClose: PivotClose returns 1 for a bullish pivot bar, 2 for a bearish pivot bar, 4 for maybe bullish, and 5 for maybe bearish. States 4 and 5 should be reevaluated after more bars have been received.
 Count is the number of bars to check on both sides of the pivot bar. Default is 2.
 A bullish pivot bar has a close that is lower than the close(s) of the bar(s) on either side.
 A bearish pivot bar has a close that is higher than the close(s) of the bar(s) on either side.
 Additional checks are made for equal closes so the pivot bar is the first in the group.
 Example: `Formation(ePivotClose,index,2);`

PivotRange: PivotRange returns 1 for a Low pivot bar, 2 for a High pivot bar, 3 for both Low and High, 4 for maybe Low pivot, and 5 for maybe High pivot. States 4 and 5 should be reevaluated after more bars have been received.
 Count is the number of bars to check on both sides of the pivot bar. Default is 2.
 A Low pivot bar has a low that is lower than the low(s) of the bar(s) on either side.
 A High pivot bar has a high that is higher than the high(s) of the bar(s) on either side.
 Additional checks are made for equal highs or lows so the pivot bar is the first in the group.
 Example: `Formation(ePivotRange,index,4);`

RangeSize: RangeSize returns 1 for large range bars, 2 for small range bars, and 0 otherwise.
 A large range bar has a range more than 1.618 times the average range.
 A small range bar has a range less than 0.382 times the average range.
 The average range is the simple average of the ranges of the previous 5 bars.

Trend: Trend returns 1 for bullish bars, 2 for bearish bars, and 0 otherwise.
 A bullish bar has a higher high and a higher low than the previous bar.
 A bearish bar has in a lower low and a lower high than the previous bar.
 If neither of the above conditions were met, then a bar is bullish if its net is positive and its close is above its open, or a bar is bearish if its net is negative and its close is below its open. If none of the above conditions are met, then a bar is bullish if it has a higher high than the previous bar, and its close is above its midpoint, or a bar is bearish if it has a lower low then the previous bar, and its close is below its midpoint.

TurningPoint: TurningPoint returns 1 for a bullish turning point bar, 2 for a bearish turning point bar, and 0 otherwise.
 A bullish turning point bar meets these conditions:
 1. Its low is lower than or equal to the low of the previous bar.
 2. Its low is lower than the low of the succeeding bar.
 A bearish turning point bar is the inverse formation of the bullish turning point bar.
 1. Its high is higher than or equal to the high of the previous bar.
 2. Its high is higher than the high of the succeeding bar.

If a turning bar meets both the bearish and the bullish conditions simultaneously, then the trend of the prior two bars, and the net is used to choose between bullish and bearish.

VolumeSize: **VolumeSize** returns 1 for large volume bars, 2 for small volume bars, and 0 otherwise. A large volume bar has a range more than 1.618 times the average volume. A small volume bar has a range less than 0.382 times the average volume. The average volume is the simple average of the volumes of the previous 5 bars.

EXAMPLE: The following example opens an IBM daily chart. A **For** loop is used to examine each bar and determine if an Outside Range or Inside Range condition exists. The bars are colored based on the **Formation** results.

```
var                                {Start of Variable declarations}
  i:integer;                       {i is declared as an Integer }
begin                               {Start of Main programming code}
  Chart('IBM.D');                 {Open an IBM daily chart}
  for i:= BarBegin to BarEnd do SetBar(eColor, i , Formation(eOutside, i ));
  ChartRefresh();                 {Refresh the chart to display colors}
end;                               {End of program}
```

Frac Round Trunc

SYNTAX: **Frac**(*Number*: real): real;
Round(*Number*: real): integer;
Trunc(*Number*: real): integer;

DESCRIPTION: The **Round** function rounds a *Number* to the nearest whole number. **Round**(10.5) returns 10. **Round**(10.51) returns 11. The **Trunc** function truncates a *Number* (rounds down to the nearest whole number). **Trunc**(10.51) returns 10. The **Frac** function returns the fractional part of a *Number*. **Frac**(10.51) returns 0.51.

EXAMPLE: The following example prints the results of **Frac**, **Round**, and **Trunc** with the value 10.51.

```
begin
  writeln(Frac(10.51));           {Prints  0.51}
  writeln(Round(10.51));         {Prints  11}
  writeln(Trunc(10.51));         {Prints  10}
end;
```

FTPdownload FTPupload

SYNTAX:

FTPdownload(*HostName*, *HostDirectory*, *HostFileName*, *LocalFileName*, *UserName*, *Password*: string);
FTPupload(*HostName*, *HostDirectory*, *HostFileName*, *LocalFileName*, *UserName*, *Password*: string);

DESCRIPTION: **FTPdownload** and **FTPupload** are used to transfer files to and from an Internet Server Host computer. This allows you to download and upload files from an Internet site. You can actually update your web site using the ESPL language. The functions return a **True** value if the download or upload is successful, otherwise a **False** value. **NOTE:** Your computer must have an active Internet connection in order for the functions to upload or download files.

PARAMETERS:

HostName: Enter the IP address of the Host Internet server computer. Example: '214.113.64.3'
HostDirectory: Enter the Directory Path on the Host Internet server where the HostFile is located.
HostFileName: Enter the FileName on the Host Internet server computer that will be uploaded or downloaded.
LocalFileName: Enter the Path and FileName for the file on your computer that will be uploaded or downloaded to.
UserName: Enter the required 'UserName' to log on to the Host Internet server computer.
Password: Enter the required 'Password' to log on to the Host Internet server computer.

EXAMPLE: The following program demonstrates how to upload and download a file from an Internet Server computer. An HTML file named TEST.HTML is uploaded when ESPL button 1 is clicked. The file is downloaded when ESPL button 2 is clicked.

```
begin
  if ESPL=1 then
    FTUPupload('214.113.64.3','/www/ftp/','Test.html','C:\Test.html','MyName',
      'MyPassword');
  if ESPL=2 then
    FTPdownload('214.113.64.3','/www/ftp/','Test.html','C:\Test.html',
      'MyName','MyPassword');
end;
```

Function

SYNTAX: **Function** *FunctionName*(*ParameterList*);

DESCRIPTION: Use the **Function** keyword to create a function subroutine in the ESPL program. Functions are called from within the ESPL program. When the programming code in the function has completed, program execution resumes on the next line following the calling statement. A predefined variable named 'Result' is used to return a value to the calling line.

PARAMETERS:

FunctionName: *FunctionName* is the name of the function, and must be a unique name.

ParameterList: The ParameterList for a function definition is not optional. At least one parameter must be used, even if it amounts to being an dummy parameter. The ParameterList is a list of parameters that will be passed to the function, using the following syntax:

ParameterName: VariableType [*ParameterName:* VariableType [...]]

ParameterName is the name the parameter will be called in the function, and *VariableType* is the variable type of the parameter. Note that multiple parameters are separated by a semi-colon in the ParameterList for the function definition, but they are separated by a comma in the function call.

If you precede a parameter with the keyword **var**, then any changes made to the parameter will be reflected in the actual parameter, rather than just passing the value of the parameter.

EXAMPLE: The following example defines a simple **Function** named 'AddNumbers'. Two random numbers (between 1 and 1000) are passed to the function. The values are added together. The resulting integer sum is returned to the calling line. The program is run by clicking button 1 (*ESPL=1*) from within the ESPL Script Editor window. The programming code in the function is executed when called by the **AddNumbers**(Random(1000),Random(1000)) function call statement. NOTE: Several functions can be created in the same ESPL program. However, make sure that a Function is

always higher up in the programming code, than the programming line that calls it. For example, a Function that starts on line 30 can't be called by a reference on line 10. The Function must be higher in the code than any call to the function.

```
Function AddNumbers (Number1, Number2:integer);
var
  Sum: integer;
begin
  Sum := Number1 + Number2 ;
  write (Number1, ' + ', Number2, ' = ');
  Result := Sum;
end;

{**** Main Program****}
begin
  if ESPL = 1 then writeln (AddNumbers (Random (1000), Random (1000)));
end;
```

Get

SYNTAX: **Get**(*Symbol*: string[, *Field*: integer, *Feed*: integer]): real;

DESCRIPTION: **Get** returns the Field value for the specified symbol. The Feed for the vendor data feed may be provided or set previously in the Feed global variable. If the symbol is not found, then **Get** will return zero. This function is more efficient than using the two functions of **Find** and **GetData**. The default Field is eClose.

PARAMETERS:

Symbol: Specifies the symbol to search for, and return the Field value.
Field: The value from the quote record to return. See the Field selections under GetData.
Feed: The vendor data feed for the symbol. Ie eSignal, eIB, eFXCM, eDBFX, eTransAct, eTraderBytes.

EXAMPLE: The following example gets the Last price for 3 symbols, and then updates a Custom symbol.

```
var
  price: real;
begin
  Feed := eSignal;
  price:= Get ('MSFT') + Get ('DELL') + Get ('APPL');
  Put ('MYSTOCK', price, 2);
end;
```

GetBar

SYNTAX: **GetBar**(*Index*: integer, **var** *Date*: integer, **var** *Open*: real, **var** *High*: real, **var** *Low*: real, **var** *Close*: real, **var** *Volume*: integer, **var** *OpenInterest*: integer): boolean;

DESCRIPTION: **GetBar** is used to read a chart bar's values. Example, the date, open, high, low, close, volume, and open-interest can be read. For Intra-day charts (like a 1-minute chart), the open-interest field will contain the bar time-stamp (example: 1030 1031 1032 1033 1034 1035 etc.). The open-interest field for stock charts generally contains a zero value. **GetBar** is a useful way to read all the values for a bar, using just one statement.

PARAMETERS:

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart. If *Index* is less than or equal to zero, the function will use index as an offset from the last bar on the chart. If *Index* is out of range, the function will return false, otherwise true.

The *Date*, *Open*, *High*, *Low*, *Close*, *Volume*, and *OpenInterest* variables will contain the bar values after calling the function.

EXAMPLE: The following example opens an IBM daily chart, then reads and prints the values for the last 10 bars on the chart.

```
var                                {Start of Variable declarations}
  Open,High,Low,Close: real;        {Declares variables as Real}
  i,Date,Volume,OpenI: integer;    {Declares variables as Integers}
begin                               {Start of Main Programming code}
  Chart('IBM.D');                  {Opens an IBM daily chart}
  for i:= BarEnd-10 to BarEnd do begin {FOR loop, counts last 10 bars}
    GetBar(i,Date,Open,High,Low,Close,Volume,OpenI); {Read bar values}
    writeln(Date,' ',Open,' ',High,' ',Low,' ',Close,' ',Volume); {Print}
  end;                               {End of loop code}
end;                                {End of program}
```

GetCell

SetCell

SelectedCell

RowColor

SYNTAX: **GetCell**(*Column*, *Row*: integer): string;
SetCell(*Column*, *Row*: integer, *Text*: string [, *Color*,*Marker*: integer]): boolean;
SelectedCell(var *Column*, var *Row*: integer, var *Text*: string, var *Value*: real, var *Feed*: integer, var *Symbol*: string, var *Title*: string): boolean;
RowColor(*Row*,*Color*: integer);

DESCRIPTION:

GetCell **GetCell** can read any cell from a Quote, Account, Spread Sheet or Trade table. If the command fails, a null string is returned.

SetCell: **SetCell** is the reverse of the **GetCell** function. Any string can be written to any cell. Using **SetCell**, a quote table can be filled with custom titles, values and markers. The cell will be colored using the *Color* parameter if it is provided. **SetCell** returns False if an error occurs, True otherwise.

SelectedCell: **SelectedCell** returns the *Column*, *Row* and *Text* for the cell the user clicked on. The *Text* from the cell is returned as a value in the *Value* parameter (useful if it is a number). The *Feed*, *Symbol*, and column *Title* for the cell are also returned. **SelectedCell** returns a False a value if an error occurs, True otherwise.

RowColor: **RowColor** is used to set the color of a row of cells. This statement is supported for the Portfolio, Spreadsheet, Optimizer and Scanner grids.

PARAMETERS:

Column: *Column* specifies the column. Column zero is the left-most column.
Row: *Row* specifies the row. Row zero is the top title row.

Text: *Text* specifies the text to save into a cell (when using the **SetCell** function).
Text contains the retrieved cell text (when using the **SelectedCell** function).
Color: Specifies a Color. The cell will be colored with this color when using **SetCell**.
The row will be colored with this color when using **RowColor**.
Marker: Specifies a Marker. If the *Text* string is empty, the marker will be centered in the cell. Otherwise the marker will be shown on the left side of the text. To set a Marker without changing the cell color, pass -1 as the *Color* parameter. See [Appendix: Markers](#)

EXAMPLE #1: The following example opens a Custom Quote page. The top left cell is set to 'Custom' and is colored red. The Symbols in the left column are printed (until a blank row is encountered).

```
var
  i: integer;
  Symbol: string;
begin
  Quote (eCustom);
  SetCell(0,0, 'Custom', clRed);
  i:=1;
  while GetCell(1,i) <> '' do begin
    Symbol:= GetCell(1,i);
    writeln('Row ',i, ' =', Symbol);
    inc(i);
  end;
end;
```

EXAMPLE #2: The following example is used to set an Alert for a selected symbol. Click the mouse on a quote page row, and then Click the Run button. The **SelectedCell** function is used to retrieve values from the selected cell. An Alert is set at the symbol's High price.

```
var                                     {Start of Variable declarations}
  row,col,feed: integer;                {Variables declared as Integers}
  value: real;                           {Variable declared as a Real}
  text,symbol,title: string;            {Variables declared as Strings}
begin                                    {Start of Main Programming code}
  FindWindow(eQuote);                   {Locates the open Quote page}
  SelectedCell(col,row,text,value,feed,symbol,title); {Read Cell data}
  Alert(symbol,market,value,true);      {Set high alert}
end;                                     {End of program}
```

GetData

SYNTAX: **GetData**(*Field*: integer [, *Format*: boolean]): variant;

DESCRIPTION: Returns the specified *Field* price value for the last retrieved quote record. Use the **Find** and **FindMarket** functions to locate a symbol in the quote pages. After finding a symbol, use the **GetData** function to retrieve and process values for the symbol.

PARAMETERS:

Field: Field can be one of the following predefined constants:

eAsk	eAskSize	eBeta	eBid	eBidSize	eClose
eDividend	eDown	eEarnings	eEPS	eEstEPS	eExchange
eExpiration	eFormT	eFlag	eHigh	eInterest	eIssuer
eLast	eLow	eMarket	eMarketID	eMarketName	eName

eNet	eOpen	eScaleFactor	ePERatio	eSettled	eSharesOut
eStrike	eSymbol	eTickTime	eTickVolume	eTotal	eUnchanged
eUp	eVolume	eYearlyHigh	eYearlyHighDate		eYearlyLow
eYearlyLowDate	eYesterday	eYield			

For DTN data-feed users, the following fields are also available: eAveVolume and eInstitution

eFlag returns a byte (bit 1 will equal 1 if a chart is open, bit 2 will equal 1 if a price alert is set).

To test the value of bit 1 use the following code: `if (GetData(eFlag) and 1)=1 then`

To test the value of bit 2 use the following code: `if (GetData(eFlag) and 2)=1 then`

eMarketName and eSymbol return strings.

eMarketID returns a market group character for use in creating custom quote page files. Append a symbol to the eMarketID and save to a file. Example: `FileSymbolEntry:= GetVariable(eMarketID) + GetVariable(eName);`

Format: The *Format* parameter is used to specify a format for the returned value. The value of *Format* should be either True or False. The default value of *Format* is False.

For Open, High, Low, Last, Net, Bid and Ask the default price format is a Display value (example: 10516 for bonds). Set *Format* to True to return a Calculation value (105.50 instead of 10516 for bonds).

If *Format* is True, eTickTime is calculated in number of seconds since midnight.

If *Format* is False, eTickTime returns a TickTime string showing hours, minutes and seconds.

If *Format* is True, eExpiration, eYearlyHighDate and eYearlyLowDate return the number of days since 01-01-1970. If *Format* is False, then return a Date in the format of 'mm-dd-yy'.

EXAMPLE: The following example finds the IBM symbol in the quote pages, retrieves the symbol name and values for the Bid and Ask prices, and then prints them.

```
begin                                     {Start of Main Programming code}
  if Find(eStock,'IBM') then             {Find and retrieve the IBM record}
  begin
    writeln(GetData(eSymbol),' Bid Price= ', GetData(eBid)); {Print Bid price}
    writeln(GetData(eSymbol),' Ask Price= ', GetData(eAsk)); {Print Ask price}
  end;
end;                                     {End of program}
```

GetLevels

SYNTAX: **GetLevels**(DayFlag: byte; var Open, var High, var Low, var Close: real): boolean;

DESCRIPTION: The **GetLevels** function is used to retrieve the Open, High, Low, Close prices for any of the past 3 days. The function can also retrieve the Bar Index positions for the Open, High, Low, Close prices for any of the past 3 days. The prices are the same as the Daily Price Levels draw tool. However, the Daily Price Levels tool does not need to be applied on the chart to retrieve the prices or index positions. The function returns a True value if it succeeds, otherwise a False value is returned.

PARAMETERS:

DayFlag: The *DayFlag* value should be a value between -3 and 3 as shown below.

-3	Returns the Bar Indexes for the O,H,L,C for 2 days ago
-2	Returns the Bar Indexes for the O,H,L,C for yesterday
-1	Returns the Bar Indexes for the O,H,L,C for today

- 1 Returns the O,H,L,C price values for today
- 2 Returns the O,H,L,C price values for yesterday
- 3 Returns the O,H,L,C price values for 2 days ago

Open, High

Low, Close: These Variable parameters need to be declared before using this function. The return values will be placed in these Variables. See example below.

EXAMPLE: The following program opens an IBM 5-minute chart, and then retrieves the Prices and Bar Index Positions for yesterday's Open, High, Low, Close values.

```
var
  oPrice,hPrice,lPrice,cPrice:real;
  oIndex,hIndex,lIndex,cIndex:integer;
begin
  Chart('IBM.5');
  GetLevels(2, oPrice, hPrice, lPrice, cPrice);
  GetLevels(-2, oIndex, hIndex, lIndex, cIndex);
  writeln('Yesterdays High was ', hPrice);
  writeln('The bar index for yesterdays Open is ', oIndex);
end;
```

GetStudy SetStudy

SYNTAX: **GetStudy**(*Study*: integer, *Select*: integer [, *Index*: integer]): variant;
SetStudy(*Study*: integer, *Select*: integer, *Value*: real [, *Index*: integer]): boolean;

DESCRIPTION:

GetStudy: **GetStudy** is used to retrieve chart Study and Draw Tool values, parameters, and settings. The *Index* parameter is not necessary when retrieving study parameters (since they are not associated with a particular bar).

SetStudy: **SetStudy** is the reverse of **GetStudy**. **SetStudy** is used to Set the Study or Draw Tool parameters and settings, but cannot set the crossing flags.

PARAMETERS:

Study: *Study* is the study Object ID number (handle) returned by the **FindStudy**, **AddStudy**, or **AddLine** functions. If the value of *Study* is set to Zero, then the **GetStudy** and **SetStudy** commands will default to the calling chart study (without having to specifically identify the Object ID number). Use the **FindStudy** command to find a Study or Draw Tool line on a chart, and then use the returned *Handle* as the *Study* parameter when using the **GetStudy** or **SetStudy** commands.

Select: *Select* specifies which study or draw tool value to Get or Set. *Select* is a number from 0 through 955 and numbers for each Study and Draw Tool are documented on the following pages. These constants may be used as well for the *Select* parameter.

- eParm1: Get or Set the 1st parameter value from the Properties window.
- eParm2: Get or Set the 2nd parameter value from the Properties window.
- eParm3: Get or Set the 3rd parameter value from the Properties window.
- eOffset: Get or Set the Offset Up/Down value from the Properties window.
- Eshift: Get or Set the Shift Left/Right value from the Properties window.

Example: `SetStudy(handle, eParm1, value);`

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart.

Value: *Value* is used to set a study or line parameter value.

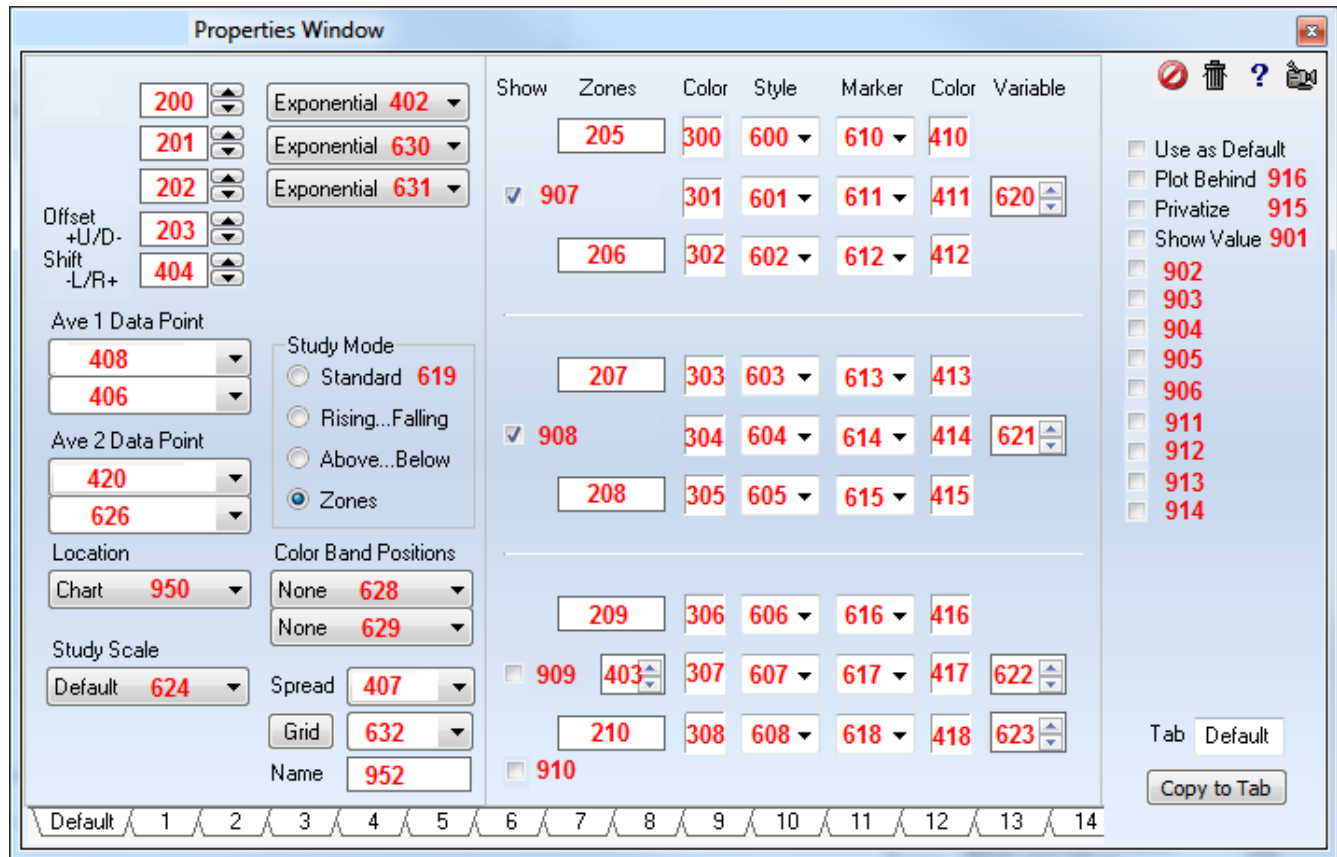
Select	Most Studies	Description
0	1 st line value	Study value, same as Select = 1
1, 2, 3	1 st , 2 nd , 3 rd line value	Study value
4, 5, 6	1 st , 2 nd , 3 rd line slope	Study value – prior study value
7	1 st line >= 2 nd line	Flag: 1 st line greater than or equal to 2 nd
8	1 st line <= 2 nd line	Flag: 1 st line less than or equal to 2 nd line
9	1 st line X> 2 nd line	Flag: 1 st line cross above 2 nd line
10	1 st line X< 2 nd line	Flag: 1 st line cross below 2 nd line
11	1 st line X<> 2 nd line	Flag: 1 st line crosses 2 nd line
12	1 st line and 2 nd line rising	Flag: both line slopes are positive
13	1 st line and 2 nd line falling	Flag: both line slopes are negative
14, 15, 16	1 st , 2 nd , 3 rd line rising	Flag: study value => prior study value
17, 18, 19	1 st , 2 nd , 3 rd line falling	Flag: study value <= prior study value
20, 21, 22	1 st , 2 nd , 3 rd turns up	Flag: slope goes positive
23, 24, 25	1 st , 2 nd , 3 rd turns down	Flag: slope goes negative
26, 27, 28	1 st , 2 nd , 3 rd changes direction	Flag: line turns up or turns down
29, 30, 31	1 st , 2 nd , 3 rd near #3 +/- #4	Flag: study >= #3 - #4 and #2 <= #3 + #4
32, 33, 34	1 st , 2 nd , 3 rd between #3 & #4	Flag: study >= #3 and study <= #4
35, 36, 37	1 st , 2 nd , 3 rd between #3 & (#3+#4)	Flag: study >= #3 and study <= #3 + #4
38, 39, 40	1 st , 2 nd , 3 rd as percent of scale	100* (Study – Scale low) / Scale Range
41,42,43,44,45	1 st , 2 nd , 3 rd , 4 th , 5 th parameters	
46	Object ID	Each object is assigned a unique number
47, 48, 49	1 st , 2 nd , 3 rd line color	The color the study line segment was drawn with
Select	Stop Studies	Description
1, 2, 3	Stop, High stop, Low stop value	Study value
4	Stop spread	Close – Study Value
5	Stop slope	Study value – prior study value
6	Stop as percent of scale	100* (Study – Scale low) / Scale Range
7, 8, 9	Stop hit, High and Low stop hit	Flag: true when stop is touched
10	Position long	Flag: Low stop is active
11	Position short	Flag: High stop is active
12, 13, 14	1 st , 2 nd , 3 rd parameters	Parameters 1, 2, Offset U/D

Select	DYO Study	Description
1, .. 12	Row A – L value	Study value
13 .. 24	Row A – L <> 0	Flag: study value not equal to zero
25 .. 36	Row A – L = 0	Flag: study value equal to zero, ie False
37 .. 48	Row A – L rising	Flag: study value >= prior study value
49 .. 60	Row A – L falling	Flag: study value <= prior study value
61 .. 72	Row A – L turns up	Flag: slope goes positive
73 .. 84	Row A – L turns down	Flag: slope goes negative
85 .. 96	Row A – L goes true	Flag: study flag goes from False to True
97 .. 108	Row A – L goes false	Flag: study flag goes from True to False
109 .. 120	Row A – L changes state	Flag: study flag changes state
121 .. 132	Row A – L bars since true	Index – prior index where flag was True
133 .. 144	Row A – L bars since false	Index – prior index where flag was False
Array Index	All Studies	Description
200 .. 250	Properties	Single precision values for edit fields
300 .. 315	Colors	Line Colors
400 .. 424	Integers	Integer values for combo box selections
501 .. 512	Word	Index for DYO Variables
600 .. 800	Byte	Byte values for combo box, and strings
901 .. 932	Flags	Check box selections
915	Privatize Flag	Set to True prevents display of property form
950	wIDX	Location window selection, 0 .. 9
951	Object ID	Each object is assigned a unique number
952	sShortName	Study short name string
953	Scale High	Study High Scale
954	Scale Low	Study Low Scale
955	Scale Range	Study Range = High Scale – Low Scale
956	Tab	Which tab is selected, 0 through 14.
960	NoSave flag (SetStudy only)	Use SetStudy(study,960,True) to block saving
Select	Commodity Channel Index	Description
0 .. 49	See documentation for Most Studies	
50	Woodie CCI Trend Up	Boolean flag
51	Woodie CCI Trend Down	Boolean flag
52	Woodie CCI Pre-Trend	Boolean flag
53	Woodie CCI Trigger Bar	Boolean flag

Select	Pesavento Patterns	Description
0,1	Last swing price	Could be a swing high or a swing low price
2	Swing High price	
3	Swing Low price	
4	Direction flag	True= Up, False = Down
5 .. 12	1 st through 8 th swing prices	
13 .. 20	1 st through 8 th swing indexes	
21	Swing bar value	0 – nothing, 1 – High swing, 2 – Low swing bar
22	Flag for a swing high bar	
23	Flag for a swing low bar	
24	Trend line value	
25	Trend line slope	
26	Trend line rising flag	
27	Trend line falling flag	
28	Trend line changed direction flag	
29	Object ID	Each object is assigned a unique number
Select	Most Draw Tools	Description
0,1	Line Value	
2	Line Slope	
3	Line Rising	
4	Line Falling	
5	Close >= Line	
6	Close <= Line	
7	Close >= Line and Low < Line	
8	Close <= Line and High > Line	
9	Close X> Line	Bar's last crosses above line
10	Close X< Line	Bar's last crosses below line
11	Close X<> Line	Bar's last crosses the line in either direction
12	Close X<> Line and bar on line	Bar's last crosses line and High > line, Low < line
13	High X> Line	
14	High X< Line	
15	Low X> Line	
16	Low X< Line	
17	High X> Line or Low X< Line	
18	Point A price	
19	Point B price	
20	Point C price	

21	Point A index	
22	Point B index	
23	Point C index	
24	Bar count after point A	
25	Bar count after point B	
26	Bar count after point C	
27	Upper standard deviation channel	
28	Lower standard deviation channel	
29	wID	Object ID
30	Note text	For lines, right side label text
31	Left Label text	For lines, left side label text
	Select	Pyrapoint
201	Degrees	
	Select	Cycles
204	Period	
	Select	Draw Line
206	Slope	
	Select	Andrews Pitchfork
209	Fork Variations	

PROPERTY FORM for MOST STUDIES:



Example: Handle:= FindStudy(eSto); {Find the Stochastics study. *Handle* identifies the study}
 SetStudy(Handle, 200, 14); {Set the identified Stochastics study 'Bars' parameter to 14}
 SetStudy(Handle, eParm2, 9); {Set the identified Stochastics study '%K' parameter to 9}
 SetStudy(Handle, eParm3, 3); {Set the identified Stochastics study '%D' parameter to 3}

EXAMPLE: The following program opens an IBM daily chart and applies a Relative Strength Index study from *Tab 4*. The **SetStudy** command is used to change the Line Color and the Line Style. The **GetStudy** command is used to get and print the RSI study values for the last 10 bars on the chart.

```
uses Graphics;

begin
    Chart('IBM.D');                                            {Start of Main Programming code}
    Handle:= AddStudy(eRSI,4);                                {Open an IBM daily chart}
    SetStudy(Handle,301,clAqua);                            {Add Tab 4 RSI study to the chart}
    SetStudy(Handle,601,7);                                 {Set Line Color to Aqua}
    ChartRefresh(True,GetStudy(Handle,950),Handle);        {Set Line Style to Dotted}
    ChartRefresh(True,GetStudy(Handle,950),Handle);        {Redraw the Changes}
    for I := 1 to 10 do begin                                 {Loop through the last 10 bars}
        Value := GetStudy(Handle,1,BarEnd-i+1);            {Get the RSI value for bar}
        writeln(i,' ',Value);                                {Print the value}
    end;
end;                                                            {End of program}
```

EXAMPLE: The following extreme example shows changing every field. This is not necessary as shown in the previous example. This example is shown for the sake of documentation.

```

uses Graphics;

var Handle: integer;

procedure ChangeProperties;
begin
  FindWindow(eChart);
  Handle := FindStudy(eAve);
  if Handle=0 then Handle := AddStudy(eAve,0);

  SetStudy(Handle,200,15);      // Set 1st parameter to 15 (can also use eParm1 )
  SetStudy(Handle,201,5);      // Set 2nd parameter to 5 (can also use eParm2 )
  SetStudy(Handle,202,3);      // Set 3rd parameter to 3 (can also use eParm3)
  SetStudy(Handle,203,0);      // Set Offset to 0 (can also use eOffset)
  SetStudy(Handle,404,5);      // Set Shift to 5 bars (can also use eShift)

  SetStudy(Handle,402,0);      // Set 1st Average Calculation to Simple
  SetStudy(Handle,630,1);      // Set 2nd Average Calculation to Exponential
  SetStudy(Handle,631,2);      // Set 3rd Average Calculation to Weighted
  SetStudy(Handle,408,1);      // Set 1st data point to High
  SetStudy(Handle,406,0);      // Set 1st StudyOnStudy data point to 1st item
  SetStudy(Handle,420,2);      // Set 2nd data point to LOW
  SetStudy(Handle,626,1);      // Set 2nd StudyOnStudy data point to 2nd item

  SetStudy(Handle,619,1);      // Set Study Mode to Rising...Falling
  SetStudy(Handle,950,1);      // Set plot Location to Sub Window 1
  SetStudy(Handle,624,3);      // Set Study Scale to Data Set
  SetStudy(Handle,628,2);      // Set 1st Color Band Position to Above High 1
  SetStudy(Handle,629,3);      // Set 2nd Color Band Position to Below Low 1
  SetStudy(Handle,407,2);      // Set Spread plot alignment to Right
  SetStudy(Handle,632,6);      // Set Grid Lines to Tab 6
  SetStudy(Handle,952,'Hello'); // Set Name of study to 'Hello'
  SetStudy(Handle,907,True);    // Set Show to True for 1st study line
  SetStudy(Handle,908,True);    // Set Show to True for 2nd study line
  SetStudy(Handle,909,False);   // Set Show to False for Spread or 3rd study line
  SetStudy(Handle,910,False);   // Set Show to False for Donchian Channel 3rd line

  SetStudy(Handle,205,80);      // Set 1st line Upper Zone value to 80
  SetStudy(Handle,206,20);      // Set 1st line Lower Zone value to 20
  SetStudy(Handle,207,90);      // Set 2nd line Upper Zone value to 90
  SetStudy(Handle,208,10);      // Set 2nd line Lower Zone value to 10
  SetStudy(Handle,209,75);      // Set Spread Upper Zone value to 75
  SetStudy(Handle,210,25);      // Set Spread Lower Zone value to 25
  SetStudy(Handle,403,3);      // Set Spread amplitude multiplier to 3

  SetStudy(Handle,300,clRed);    // Set 1st Upper Color to Red
  SetStudy(Handle,301,clBlue);  // Set 1st Line Color to Blue
  SetStudy(Handle,302,clYellow); // Set 1st Lower Color to Yellow
  SetStudy(Handle,303,clLime);  // Set 2nd Upper Color to Lime Green
  SetStudy(Handle,304,clOrange); // Set 2nd Line Color to Orange
  SetStudy(Handle,305,clLtBlue); // Set 2nd Lower Color to Light Blue
  SetStudy(Handle,306,clPurple); // Set Spread Upper Color to Purple
  SetStudy(Handle,307,clBlack); // Set Spread Line Color to Black
  SetStudy(Handle,308,clLtRed); // Set Spread Lower Color to Light Red

```

```

SetStudy(Handle,600,0); // Set 1st Upper Style to None
SetStudy(Handle,601,1); // Set 1st Line Style to 1 Pixel thickness
SetStudy(Handle,602,2); // Set 1st Lower Style to 2 Pixels
SetStudy(Handle,603,3); // Set 2nd Upper Style to 3 Pixels
SetStudy(Handle,604,4); // Set 2nd Line Style to 4 Pixels
SetStudy(Handle,605,5); // Set 2nd Lower Style to 5 Pixels
SetStudy(Handle,606,6); // Set Spread Upper Style to Dashed line
SetStudy(Handle,607,7); // Set Spread Style to Dotted line
SetStudy(Handle,608,10); // Set Spread Lower Style to Fuzzy line

SetStudy(Handle,610,0); // Set 1st Upper Marker to None
SetStudy(Handle,611,11); // Set 1st Line Marker to Thick Up Arrow
SetStudy(Handle,612,12); // Set 1st Lower Marker to Thick Down Arrow
SetStudy(Handle,613,13); // Set 2nd Upper Marker to Thick Left Arrow
SetStudy(Handle,614,14); // Set 2nd Line Marker to Thick Right Arrow
SetStudy(Handle,615,21); // Set 2nd Lower Marker to Big Circle
SetStudy(Handle,616,35); // Set Spread Upper Marker to Hollow Square
SetStudy(Handle,617,41); // Set Spread Marker to a Thick X
SetStudy(Handle,618,56); // Set Spread Lower Marker to Thumbs Up

SetStudy(Handle,410,clRed); // Set 1st Upper Marker Color to Red
SetStudy(Handle,411,clBlue); // Set 1st Line Marker Color to Blue
SetStudy(Handle,412,clYellow); // Set 1st Lower Marker Color to Yellow
SetStudy(Handle,413,clLime); // Set 2nd Upper Marker Color to Lime
SetStudy(Handle,414,clOrange); // Set 2nd Line Marker Color to Orange
SetStudy(Handle,415,clLtBlue); // Set 2nd Lower Marker Color to Light Blue
SetStudy(Handle,416,clLime); // Set Spread Upper Marker Color to Lime Green
SetStudy(Handle,417,clLime); // Set Spread Line Marker Color to Lime Green
SetStudy(Handle,418,clLime); // Set Spread Lower Marker Color to Lime Green

SetStudy(Handle,620,100); // Set 1st Line Global Variable to variable 100
SetStudy(Handle,621,101); // Set 2nd Line Global Variable to variable 101
SetStudy(Handle,622,102); // Set 3rd Line Global Variable to variable 102
SetStudy(Handle,623,103); // Set Spread Global Variable to variable 103

SetStudy(Handle,901,True); // Set to True (checked) for Show Value check box
SetStudy(Handle,902,True); // Set to True (checked) for this check box
SetStudy(Handle,903,True); // Set to True (checked) for this check box
SetStudy(Handle,904,True); // Set to True (checked) for this check box
SetStudy(Handle,905,False); // Set to False (unchecked) for this check box
SetStudy(Handle,906,False); // Set to False (unchecked) for this check box
SetStudy(Handle,911,False); // Set to False (unchecked) for this check box
SetStudy(Handle,912,False); // Set to False (unchecked) for this check box
SetStudy(Handle,913,False); // Set to False (unchecked) for this check box
SetStudy(Handle,914,False); // Set to False (unchecked) for this check box
SetStudy(Handle,915,True ); // Set to True prevents display of property from
SetStudy(Handle,916,True); // Set to True (checked) for Plot Behind check box
SetStudy(Handle,960,True); // Set to True to prevent object from saving

ChartRefresh(True);
end;

begin
  if ESPL=1 then ChangeProperties;
end;

```

PROPERTY FORM for MOST DRAW TOOLS:

Show	Level	Color	Style	Left	Right	Variable
<input type="checkbox"/> 400	210	300	410	600	611	222
<input type="checkbox"/> =2	211	301	411	601	612	223
<input type="checkbox"/> =4	212	302	412	602	613	224
<input type="checkbox"/> =8	213	303	413	603	614	225
<input type="checkbox"/> =16	214	304	414	604	615	226
<input type="checkbox"/> =32	215	305	415	605	616	227
<input type="checkbox"/> =64	216	306	416	606	617	228
<input type="checkbox"/> =128	217	307	417	607	618	229
<input type="checkbox"/> =256	218	308	418	608	619	230
<input type="checkbox"/> =512	219	309	419	609	620	231
<input type="checkbox"/> =1024	220	310	420	610	621	232

Alert Message 910 Silent 409 Font 405 Panel 407
 Auto Remove 911 Beep
 Sound once 913 Voice
 E-mail 914 WAV 952

Use as Default
 Draw Behind 916
 Privatize 915
 901
 902
 903
 904
 905
 906
 907
 908
 909
 205 Price A
 206 Price B
 207 Price
 Width Multiplier
 208
 Tab Default

Default 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Example: Handle:= FindStudy(eFibonacci); {Find the Fibonacci Draw Tool on the chart}
 SetStudy(Handle, 301, clRed); {Set the Line Color to RED for the 2nd row}

EXAMPLE: The following program opens an IBM daily chart and adds an ESPL Color Band study from *Tab 9*. Click ESPL button 1 to run the program. The *ESPL* variable value is set to 61. The 'UpdateColorBand' procedure marks the direction of the Last price using the **SetStudy** command. The Color Band will draw the Bullish Color and Marker for Up bars, and the Bearish Color and Marker for down bars. NOTE: Entering a 0 value for the *Study* handle (in the **SetStudy** parameter) will automatically refer to the calling Color Band study.

```

procedure UpdateColorBand;
var i:integer;
begin
  for i:= BarBegin to BarEnd do
    if Last(i)>=Last(i-1) then SetStudy(0,0,1,i) else SetStudy(0,0,2,i);
end;

{*****Main Program Code*****}
begin
  if ESPL=1 then begin

```

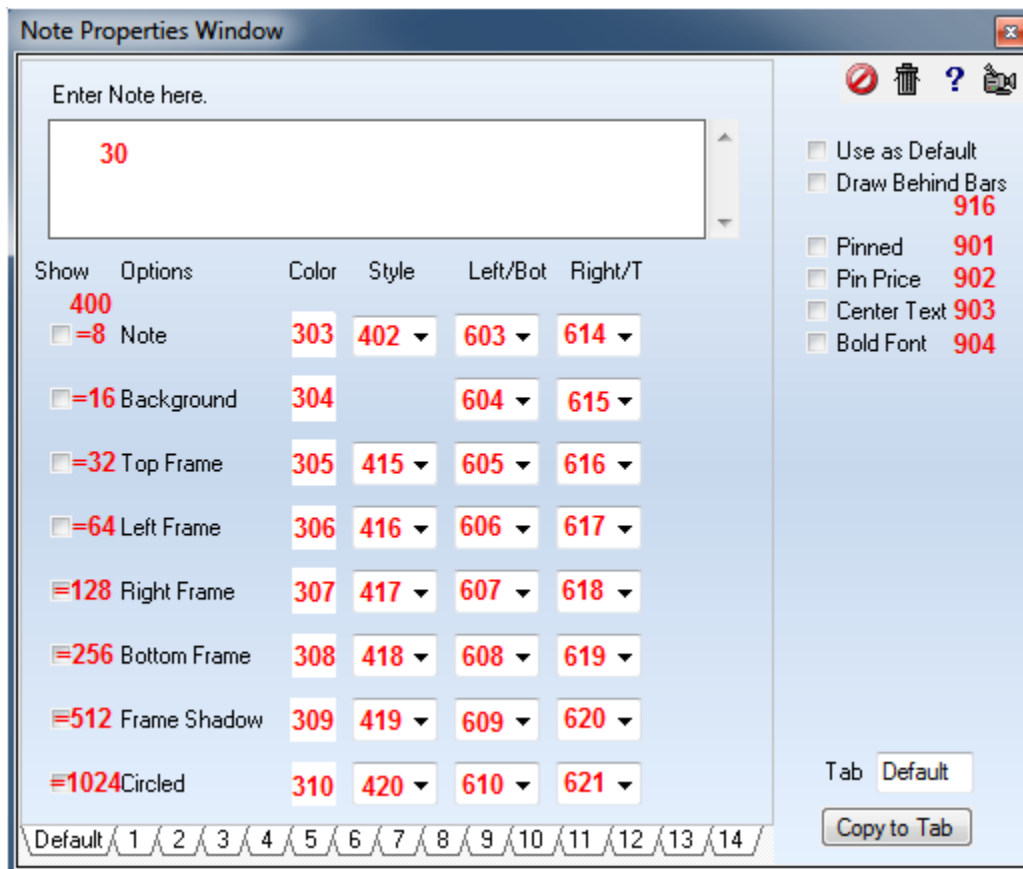


```

Chart('IBM.D'); AddStudy(eBnd,9,0,0,0,0,61,0,7,0,False);
end;
if ESPL=61 then UpdateColorBand; {runs the UpdateColorBand procedure}
end;

```

PROPERTY FORM FOR NOTES



GetToken SetToken

SYNTAX: **GetToken**(Position: integer, Text: string [, Delimiter: string]): string;
SetToken(Position: integer, var Text: string, Token: variant, Delimiter: string);

DESCRIPTION: Use **GetToken** to retrieve the text located between the Nth and N-1 *Position* delimiter. This could be used to extract a price from a group of prices in a string (where each price is separated with a comma). Use **SetToken** to insert text into a string at the Nth delimiter *Position*.

PARAMETERS:

Position: *Position* specifies the delimiter position to search for, starting at position 1.

Text: This is a string containing text characters that are separated by delimiter characters (like commas or spaces).

Delimiter: *Delimiter* is the character (like a comma) that separates different sections of the *Text*. The default delimiter is a Space character. This parameter is optional for **GetToken**, but required for **SetToken**.

NOTE: When using **SetToken**, if *Token* is not a string, it will be converted to a string. Integer numbers will be converted using **IntToStr(Token)**. Real numeric types will be converted to a string with two decimal places. Boolean types will be converted to the text 'True' or 'False'.

EXAMPLE: The following example uses **GetToken** to extract prices from a string. The values in the string represent the Date, Open, High, Low, Last, and Volume. **SetToken** is then used to replace the Volume text with a different value. The original text, the extracted text, and the altered text are printed.

```
var                                     {Start of Variable declarations}
  Text, D, O, H, L, C, V: string;       {Variables declared as Strings}
begin                                   {Start of Main Programming code}
  Text:='12-31-02,100,200,50,150,1000'; {Text prices assigned to Variable}
  D:= GetToken(1, Text, ',');           {Extract Date}
  O:= GetToken(2, Text, ',');           {Extract Open}
  H:= GetToken(3, Text, ',');           {Extract High}
  L:= GetToken(4, Text, ',');           {Extract Low}
  C:= GetToken(5, Text, ',');           {Extract Close}
  V:= GetToken(6, Text, ',');           {Extract Volume}
  writeln(Text);                        {Print original text}
  writeln(D, ' ', O, ' ', H, ' ', L, ' ', C, ' ', V); {Print extracted text}
  SetToken(6, Text, '5000', ',');       {Change Volume to 5000}
  writeln(Text);                        {Print altered text value}
end;                                    {End of program}
```

GetUser SetUser or Plot

SYNTAX: **GetUser**(*Variable* [, *Index*]): real;
SetUser(*Variable*, *Value* [, *Index*, *Color*, *Style*, *Marker*, *MColor*]): boolean;
SetUser(*eWindow*: constant, *Location* [, *Scalelow*, *Scalehigh*: real,
50%label, 25%label, 75%label: string]): boolean;

DESCRIPTION: The **GetUser** command is used to retrieve ESPL study values and parameters from a chart. The **SetUser** command is used to Plot lines on a chart and to set ESPL study parameters. The **SetUser**(*eWindow*....) command specifies the *Location* where the ESPL study will plot, sets the scale range, and specifies some optional grid line labels. NOTE: The **Plot** function is identical to **SetUser**. Either command will perform the exact same functions.

When an ESPL study is run on a chart, several ESPL variables and arrays are made available for saving and retrieving values. The **GetUser** and **SetUser** commands are used to retrieve and save values into these variables and arrays. Each instance of an ESPL study will have its own copy of these variables and arrays. The variables and arrays provide a location to store and retrieve values that can be used and plotted.

PARAMETERS:

Value: *Value* is the value that will be stored into the referenced variable or array.
Index: *Index* is the bar position between 1 and the number of bars on the chart.
Color: *Color* specifies the Color of the ESPL line at the specified *Index*. Enter a color (ex. clBlue).
Style: *Style* specifies the line Style at the specified *Index*. Enter a number from 0-9 based on drop-down list.

Marker: *Marker* specifies the *Marker* object at the specified *Index*. Enter a number from 0-177 based on drop-down list.

MColor: *MColor* specifies the *Marker Color* at the specified *Index*. Enter a color (ex. `clRed`).
Example: `Plot(1, Low(Index), Index, clRed, 7, 40, clWhite);`
This is a powerful example. The plotted Line segment at each bar *Index* position can have its own color, style, and marker. The above example code stores the LOW price of the indicated bar into ESPL array 1. The color of the ESPL line is set to Red, with a dotted line style. The Marker is selected as a 'Star', and the Marker Color is set to White.

Location: *Location* specifies which study sub-window to plot the ESPL study lines in.
eESPL = Plot in the main Chart window and supply a user-defined range for the study.
0 or eChart = Plot in the main Chart window and use the Chart high and low price range.
1 = Plot in study sub-window 1 2 = Plot in study sub-window 2
3 = Plot in study sub-window 3 4 = Plot in study sub-window 4
5 = Plot in study sub-window 5 6 = Plot in study sub-window 6
7 = Plot in study sub-window 7 8 = Plot in study sub-window 8
9 = Plot in the Volume sub-window

ScaleLow: An optionally supplied low scale range for the plot.

ScaleHigh: An optionally supplied high scale range for the plot.

The following default scale ranges are used for each Panel:

0 default scale is the chart high and low

1-4 and eESPL default scale is 0 through 100

5-8 default scale is -200 through 200

25%Label

50%Label

75%Label: The 3 grid labels are optional strings that will label the study panel's scale. The *50%Label* is typically used for the name of the study.

Variable: *Variable* is one of the following numbers or constants, and indicates which ESPL study array, or study variable is being set, plotted, or retrieved.

ESPL Study Arrays

- 1: Get or Set the values in the 1st ESPL study array dimensioned from 1 to BarEnd (last bar on chart)
- 2: Get or Set the values in the 2nd ESPL study array dimensioned from 1 to BarEnd
- 3: Get or Set the values in the 3rd ESPL study array dimensioned from 1 to BarEnd
- 4: Get or Set the values in the SPREAD study array dimensioned from 1 to BarEnd
- 5: Get or Set the values in the 5th ESPL study array dimensioned from 1 to BarEnd
- 6: Get or Set the values in the MISCELLANEOUS ESPL study array dimensioned from 1 to 50

Example: `Plot(1, Low(Index), Index, clRed, 7, 22, clWhite);`

An Array is a list of numbers. A number can be saved into or retrieved from any bar *Index* position in the Array. The arrays can be used to store calculated study values for any bar on the chart. Study Lines are automatically drawn on the chart for any values stored in the 1st, 2nd, 3rd, and SPREAD arrays. Use **SetUser** or **Plot** to save values into the study arrays. The 5th ESPL array can be used to store and display values, but the values cannot be plotted on the chart. The MISCELLANEOUS array can be used to store up to 50 miscellaneous program values or variables.

The SPREAD array can auto-calculate the difference between the 1st and 2nd ESPL study arrays, and plot as a histogram. Or, the SPREAD array can be used to plot any price value, color, and marker (like arrays 1-3). This gives you a potential fourth custom Line that can be plotted on the chart. The default mode for the SPREAD array is *eSpread*. This will cause

the SPREAD array to auto-enter the difference between Array 1 and Array 2. The value can be plotted as a histogram. If you want to use the SPREAD array as a fourth custom line array, then the mode must be changed to *eNormal*.

eScaleMode: Set the 4th Array Spread Mode to be either *eSpread* or *eNormal*
 Example: SetUser(eScaleMode, eNormal); {Enter any Value}
 Example: SetUser(eScaleMode, eSpread); {Auto-enter Spread value}

eParm1: Get or Set the 1st parameter value from the Properties window
 eParm2: Get or Set the 2nd parameter value from the Properties window
 eParm3: Get or Set the 3rd parameter value from the Properties window
 Example: GetUser(eParm1);

eESPL: Get or Set the *ESPL* variable value (formerly the *ESPL* value)
 eShift: Get or Set the Shift Left/Right value from the Properties window
 eCenter: Get or Set the Spread Alignment (0=Left 1=Center 2=Right)
 eName: Get or Set the Name of the ESPL study

Example: SetUser(eName, 'MyStudy');

ePercent: Get or Set the 'Plot Percent' checkbox (True=checked, False=unchecked)

eRepaint: Get or Set the 'Repaint' checkbox (True=checked, False=unchecked)

NOTE: This is necessary for programs that draw lines, shapes, or text with the **MoveToLineTo, LineTo, Arc, Chord, Ellipse, Pie, Rectangle, and TextOut** commands. These items need to be repainted any time the chart changes.

eClose: Get or Set the 'Close Only' checkbox (True=checked, False=unchecked)

Example: SetUser(eClose, True);

ePlot1: Get or Set the Show 1st Line checkbox (True=checked, False=unchecked)

ePlot2: Get or Set the Show 2nd Line checkbox (True=checked, False=unchecked)

ePlot3: Get or Set the Show 3rd Line checkbox (True=checked, False=unchecked)

ePlot4: Get or Set the Show Spread Line checkbox (True=checked, False=unchecked)

NOTE: These checkboxes specify whether to Plot the array lines on the chart or not.

eShow1: Get or Set the 'Show 1st Array' checkbox (True=checked, False=unchecked)

eShow2: Get or Set the 'Show 2nd Array' checkbox (True=checked, False=unchecked)

eShow3: Get or Set the 'Show 3rd Array' checkbox (True=checked, False=unchecked)

eShow4: Get or Set the 'Show Spread' checkbox (True=checked, False=unchecked)

eShow5: Get or Set the 'Show 5th Array' checkbox (True=checked, False=unchecked)

eShow6: Get or Set the 'Calc. Sequence' box (True=checked, False=unchecked)

eShow7: True = 'Show 6th Array' 1st and 2nd values, False=Hide values

NOTE: The Show checkboxes specify whether to display the array values in the Study window on the left edge of the chart. The 5th array cannot be plotted, but the values can be displayed in the Study window. The 1st and 2nd values of the 6th array can be shown using eShow7.

eAll: Get or Set all the ePlot and eShow boxes at the same time. (True=checked, False=unchecked)

Example: SetUser(eAll, True); {Sets all ePlot and eShow items to True}

eScaleFactor: Get or Set the Display price format for values stored in the 1st, 2nd, 3rd, SPREAD, and 5th ESPL arrays. The price format affects the Display of the array values in the Study window. The default Scalefactor is the chart's scale. Example, **eScaleFactor** could be used to display decimal values in the Study window of a Bond chart (which normally converts array values into 32nds).

- 1 Display array values in 8ths, with no decimal places
- 3 Display array values in 32nds, with no decimal places
- 0 Display array values with no decimal places (555 displays as 555)
- 1 Move decimal 1 place to the left (555 displays as 55.5)
- 2 Move decimal 2 places to the left (555 displays as 5.55)

3	Move decimal 3 places to the left	(555 displays as .555)
4	Move decimal 4 places to the left	(555 displays as .0555)
5	Move decimal 5 places to the left	(555 displays as .00555)
10	Leave decimal. Display no fractional portion	(example: 45)
11	Leave decimal. Display 10ths position	(example: 45.5)
12	Leave decimal. Display 100ths position	(example: 45.56)
13	Leave decimal. Display 1000ths position	(example: 45.567)
14	Leave decimal. Display 10,000ths	(example: 45.5678)
15	Leave decimal. Display 100,000ths	(example: 45.56789)

Example: `SetUser(eScaleFactor, 2);`

eString1: Get or Set a String text variable (with a maximum of 19 characters)
eString2: Get or Set a String text variable (with a maximum of 19 characters)
eString3: Get or Set a String text variable (with a maximum of 19 characters)

Example: `SetUser(eString1, 'Buy IBM');`

NOTE: These 3 string variables can be used for miscellaneous string storage. The eString3 string shares the same variable space as the eName value. Do not use eString3 if you use the eName value to change the name of the study.

More than one chart can access ESPL programs at the same time. For example, two charts can run the same ESPL study. Each occurrence of the ESPL study will have its own set of ESPL variables. Use the **BarBegin**, **BarBeginLeft**, or **BarEnd** global variables in a **For** loop to assign values to ESPL study arrays. Use the MISCELLANEOUS ESPL array to store values that must be remembered between execution passes. The Main programming code should test the *ESPL* variable to determine which Procedure to call. The sub-routine procedure (see Connect Prices below) for the ESPL study should store calculation values for all bars, starting from either BarBegin or BarBeginLeft, and ending with BarEnd. Since an ESPL study is called from a specific chart, the *Window* variable is set to zero. This allows the ESPL program to operate on the chart that contains the ESPL study object.

EXAMPLE: The following ESPL program demonstrates the power of an ESPL custom study. Run the study on a chart by clicking ESPL Studies button 100 in the ESPL Run panel. The study will draw 4 lines on the chart, connecting the High, Last, Open, and Low prices. The study will update as new bars are completed on the chart. Storing price values into the ESPL study arrays causes lines to draw on the chart. The lines are connected, using the prices from each array value. In this example, the SPREAD array is used to Plot the 'Open' price lines.

```

procedure ConnectPrices;           {Declares the 'ConnectPrices' subroutine }
begin                               {Start of the sub-routine code}
  SetUser(eWindow,eChart);         {Plot lines directly on the chart}
  SetUser(eClose,True);           {Update lines only when bar completes}
  SetUser(eName,'MyLines');       {Name the study 'MyLines'}
  SetUser(eScaleMode, eNormal);   {Change Spread Scale mode to eNormal}
  SetUser(ePlot1,True);           {Plot the 4 arrays}
  SetUser(ePlot2,True);
  SetUser(ePlot3,True);
  SetUser(ePlot4,True);
  SetUser(eShow1,True);           {Show the values for the 4 arrays}
  SetUser(eShow2,True);
  SetUser(eShow3,True);
  SetUser(eShow4,True);

  for i := BarBegin to BarEnd do begin {Loop through all the bars}
    Plot(1,High(i),i,clBlue,1,40,clWhite); {Plot blue line with White Stars}
    if Last(i)>=Last(i-1) then          {Plot the Last line based on NET}
      Plot(2,Last(i),i,clLime,3,0)      {Plot Last with thick Green line}
    else
      Plot(2,Last(i),i,clRed,3,0);      {Plot Last with a thick Red line}
    Plot(3,Low(i),i,clAqua,7,0);        {Plot Low with a dotted Aqua line}
  end

```

```

    Plot(4,Open(i),i,clWhite,1,0);           {Plot the Open with a White line}
end;                                         {End of Loop code}
end;                                         {End of the sub-routine code}

{*****Main Program*****}
begin                                       {Start of Main Programming code}
    if ESPL=100 then ConnectPrices;        {Call the 'ConnectPrices' procedure}
end;                                         {End of program}

```

NOTE: The program code above specified the **Plot** settings to be used for each bar. Default colors, line styles, and markers will be used if the **Plot** command does not override them. For example, the following commands will use the default settings from the ESPL Properties window, instead of specifying unique colors and settings for each bar.

```

Plot(1,High(i),i);   {Plot High with default 1st Line settings}
Plot(2,Last(i),i);  {Plot Last with default 2nd Line settings}
Plot(3,Low(i),i);   {Plot Low with default 3rd Line settings}
Plot(4,Open(i),i);  {Plot Open with default Spread Line settings}

```

GetVariable

SetVariable

SYNTAX: **GetVariable**(*Field*: integer): variant;
 SetVariable(*Field*: integer, *Value*: variant);

DESCRIPTION: **GetVariable** retrieves various chart variables. **SetVariable** can be used to change some of the chart window variable values. **GetVariable** can also be used to retrieve Trading Account information from the active trading account window.

PARAMETERS:

Field:Field is one of the following predefined constants:

eBarCount	number of bars in the chart's data set. Could be used in a FOR loop.
eBarRight	index of the last bar displayed, which may be less than BarCount.
eBarLeft	index of the first bar displayed, which usually is greater than 1.
eBarEnd	index of the last bar displayed, which may be less than BarCount.
	NOTE: The global BarLeft and BarEnd variables are automatically set for the active chart. GetVariable could be used to read these values from a chart that is not the active chart.
eMaxBar	maximum number of bars allowed for the chart (array max size)
eSaveBar	get the Save/Resize value for the chart
eBarTime	time stamp for the bar currently being built, (ex. 945 for 5-minute bar)
eBarMinutes	number of minutes in the intra day bar. This number is negative for constant tick bars, -10 for ten ticks.
eStart	Returns True with 1st tick which starts a new bar, False otherwise.
eBarSpace	dot column spacing of the bars on a chart.
eScaleHigh	the highest value of the chart scale.
eScaleLow	the lowest value of the chart scale.
eScaleMode	retrieves the ScaleMode. The result will be a 0, 1, 2, or 3.
eScaleInterval	retrieves the price interval between grid lines.
eScaleMidPoint	the (highest+lowest) / 2 value of the chart scale.
eScaleFactor	controls decimal placement and price conversion. -3 is for 32nds.
eChartHigh	the highest bar price for the bars to be displayed. Normally ScaleHigh>=ChartHigh
eChartLow	the lowest bar price for the bars to be displayed. Normally ScaleLow<=ChartLow

eSlope	Returns the 'Pts/Bar' Scale value from the charts Property window
eGrid	Returns the Minimum Grid size value from the Symbol Properties
eTick	Returns the 'Tick' value from the Symbol Properties
eMinTick	Returns the minimum Tick size.
eMargin	the number of pixels in the right hand margin of the chart.
eMarket	market group code.
eMarketOpen	day session market open time, such as 720.
eMarketClose	day session market close time, such as 1400.
eMarketOpen2	evening session market open time, such as 1430.
eMarketClose2	evening session market close time, such as 630 .
eMarketTime	time of the last tick in hhmm format, such as 1400.
eMarketMinutes	time of the last tick in minutes since midnight, i.e. hour*60 + min.
eDaySession	Returns True if the DaySession properties box is checked, otherwise False.
eLayer	Returns the layer number the chart is on, {1..9}
eFirstObject	object number for the first object on the chart.
eLastObject	object number for the last object on the chart.
eObject	Returns the position in the charts Objects list (starting at position 0). If studies are removed from the chart, the position in the list can change.
eTemplate	Returns the Name of the current Template applied to the Chart.
eWinTrades	number of winning trades.
eLossTrades	number of losing trades.
eTrades	total number of trades.
eWinProfit	profit from winning trades.
eLossProfit	losses from losing trades.
eProfit	net profit from all trades.
ePosition	current position: 0 = out. Negative = short size. Positive = long size.
eLeverage	returns a leverage value (dollars per point) = '\$ / Tick' value divided by 'Tick' value
eCommission	commission subtracted for a round trip trade.
eBuyStop	the Buy Stop price that will initiate a long trade
eBuyLimit	the Buy Limit price that will initiate a long trade
eSellStop	the Sell Stop price that will initiate a short trade
eSellLimit	the Sell Limit price that will initiate a short trade
eSymbol	returns the chart name.
eSymbolGroup	returns the Integer Color Value of the Symbol Group Color Box
eTimeGroup	returns the Integer Color Value of the Time Group Color Box
eName	returns the file name for the chart.
eChart	returns the Chart Form number. This can be used to uniquely identify a chart, when more than one chart of the same symbol is open.
eForm	returns the TForm handle for the Chart. Can be assigned to a TForm variable to access form properties.
ePath	returns the path where the chart file is stored.
eLocked	returns the boolean Locked status.
eColor	returns the current ColorBar state
eColorChart	returns long integer value of Background color
eColorBars	returns long integer value of Chart Bar color
eColorBarsUp	returns long integer value of Bullish Bar color
eColorBarsDn	returns long integer value of Bearish Bar color
eColorBars3rd	returns long integer value of the 3 rd Bar color
eColorBars4th	returns long integer value of the 4 th Bar color
eColorCross	returns long integer value of Big Cross cursor color
eColorVolume	returns long integer value of Volume bar color
eColorOpenInt	returns long integer value of Open Interest color
eColorGrid	returns long integer value of Chart Grids color
eColorFont	returns long integer value of Chart Font color
eFontHeight	returns the FontHeight, in pixels, for the current Chart Font
eFontName	returns the name of the Chart Font
eFontSize	returns the size of the Font

eFontStyle returns True if font is Bold, False otherwise.
 eStudySize1..eStudySize9 returns the size of the indicated Study sub-window 1 through 9 (a % of the Chart height).

Trading Account Predefined constants:

eName returns the Account name
 eAccount returns the Account Number
 ePhone returns the Account Telephone number
 eProfit returns the Trading Account Balance
 eMarket returns the Market Value of the account
 eTrades returns the Total number of Trades
 eAve returns the Average value of each trade
 eWinTrades returns the number of Winning trades
 eLossTrades returns the number of Losing trades
 eGrid returns the Row number of the last entry in the Trade account. This value can be used in programming loops that need to know the ending point in the account.

SetVariable Parameters

eBarRight selects the index for the last bar to be displayed and redraws the chart.
 This is a useful way to scroll the chart to a new position.

eBarCount the number of bars in the chart's data set.

eBarSpace the dot column spacing of the bars on a chart.

eSaveBar Set the Save/Resize bar parameter for the chart. Does not cause the chart to reinitialize.
 A resize event may just happen sooner if the value is lowered.

eMaxBar Set the Maximum Bars parameter for the chart. A change will cause the chart to reinitialize.

eScaleFactor controls decimal placement and price conversion. -3 is for 32nds.

eMargin the number of pixels in the right hand margin of the chart.

eMarketOpen day session market open time, such as 720.

eMarketClose day session market close time, such as 1400.

eMarketOpen2 evening session market open time, such as 1430.

eMarketClose2 evening session market close time, such as 630 .

eDaySession set to True to place a check mark in the DaySession properties box.
 set to False to uncheck the DaySession properties box.
 Example: `SetVariable(eDaySession, True);`

eLeverage change leverage used to convert points into dollars.

eCommission commission subtracted for a round trip trade.

eBuyStop the Buy Stop price that will initiate a long trade

eBuyLimit the Buy Limit price that will initiate a long trade

eSellStop the Sell Stop price that will initiate a short trade

eSellLimit the Sell Limit price that will initiate a short trade

eSymbol the chart name.

eSymbolGroup Sets the Color Value of the Symbol Group Color Box
 Example: `SetVariable(eSymbolGroup, clRed);` sets the Group color to red.

eTimeGroup Sets the Color Value of the Time Group Color Box
 Example: `SetVariable(eTimeGroup, clBlue);` sets the Group color to blue.

eName the file name for the chart.

ePath the path where the chart file is stored.

eLocked the Locked status.

eScaleHigh sets the Price for the Scale high for the chart

eScaleLow sets the Price for the Scale low for the chart

eScaleMode sets the ScaleMode. The parameter should be a 0, 1, 2, or 3:
 0=Automatic price scaling
 1=Data Set (use the highest high and lowest low of the complete set of chart bars)
 2=Use Range
 3=Use Interval

eScaleInterval sets the price interval between grid lines.

eSlope sets the Scale mode to 'Square Chart' and specifies a 'Pts/Bar' value (`SetVariable(eSlope,1)`)

eColorChart sets chart Background color
eColorBars sets Chart Bar color
eColorBarsUp sets Bullish Bar color
eColorBarsDn sets Bearish Bar color
eColorBars3rd sets 3rd Bar color
eColorBars4th sets 4th Bar color
eColorCross sets Big Cross cursor color
eColorVolume sets Volume bar color
eColorOpenInt sets Open Interest color
eColorGrid sets Chart Grid Lines color
eColorFont sets Chart Font color
eFontName selects the Chart Font by name.
eFontSize set the size of the Font
eFontStyle if 2nd parameter is True, font will be Bold.
eStudySize sets the percentage of the chart the study panels use.
eStudySize1...eStudySize9 sets the size of Study sub-windows 1 through 9 (based on a % of the Chart height).

Example: `SetVariable(eStudySize2,10);`

will set Study sub-window 2 equal to 10 percent of the entire Chart's height.

EXAMPLE: The following example opens two daily charts. The program finds the IBM chart, retrieves some bar count values, and then randomly Colors the bars using a **FOR** loop. The program then finds the MSFT chart and sets the chart background color to Red.

```

var                                     {Start of Variable declarations}
  i, Color, Start, Stop: integer;      {Variables declared as Integers}
begin                                  {Start of Main Programming code}
  Chart('IBM.D');                     {Open an IBM daily chart}
  Chart('MSFT.D');                    {Open a MSFT daily chart}
  FindWindow(eChart,'IBM.D');          {Find the IBM chart}
  Start:= GetVariable(eBarLeft);       {Get bar index for left edge of chart}
  Stop := GetVariable(eBarEnd);        {Get bar index for the last bar}
  for i:=Start to Stop do begin        {Loop through the bars}
    Color:= Random(16777215);          {Generate a Random color value}
    SetBar(eColor,i,Color);            {Color the bar}
  end;                                  {End of Loop code}
  ChartRefresh();                     {Repaint chart to show color changes}
  FindWindow(eChart,'MSFT.D');        {Find the MSFT chart}
  SetVariable(eColorChart,clRed);     {Set Chart background color to Red}
end;
```

High
Last
Low
Open
OpenInt
Volume

SYNTAX: **High**(Index: integer [, Dataset: integer]): real;
Last(Index: integer [, Dataset: integer]): real;
Low(Index: integer [, Dataset: integer]): real;
Open(Index: integer [, Dataset: integer]): real;
OpenInt(Index: integer [, Dataset: integer]): integer;

Volume(*Index*: integer [, *DataSet*: integer]): integer;

DESCRIPTION: These functions are used to retrieve the specified value for the *Index* referenced chart bar. These functions can also retrieve price values from chart overlay data by passing the overlay's object number as the last parameter, or by passing a number 1, 2, 3 ... for the 1st, 2nd, 3rd ... overlay. The overlay object number is obtained with the **FindStudy** function. The functions use the *Window* variable as set by the **FindWindow** or **Chart** function to know which chart to retrieve the data from. If the script is called by a User-Defined study, the *Window* variable will be set to zero, which defaults to the chart that contains the User-Defined study object.

PARAMETERS:

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart. If *Index* is less than or equal to zero, the function will use index as an offset from the last bar on the chart. If *Index* is out of range, the function will return zero. Both the host and the overlay chart data use the same indexing.

DataSet: *DataSet* is an optional object number for an overlay data set.

EXAMPLE: The following example illustrates how to read bar values from a chart, and a chart overlay. An IBM daily chart is opened, then a MSFT daily chart is overlaid on the IBM chart. The value of the Last price for each chart is then retrieved from the last bar of each chart.

```
var                                {Start of Variable declarations}
  ObjectNumber: integer;           {Declares ObjectNumber as an Integer}
begin                               {Start of Main Programming code}
  Chart('IBM.D');                 {Opens an IBM daily chart}
  ObjectNumber := AddOverlay('MSFT.D'); {Overlays a MSFT chart}
  writeln(Last(BarEnd));           {Get the Last bar price from IBM}
  writeln(Last(BarEnd, ObjectNumber)); {Get the Last bar price from MSFT}
end;                                {End of program}
```

GV Global Variables

SYNTAX: **SetGV**(*Index*: integer, *Value*: real);
GV(*Index*: integer): real;

DESCRIPTION: Ensign has a predefined Global Variable (GV) Array that can be used by studies and ESPL programs. The GV Array can hold up to 400 values (0 – 399). The **SetGV** command is used to place values into the array. The **GV** command is used to retrieve values from the array. There is no need to dimension or free the array.

Values assigned into array elements 0 through 199 are global to Ensign. They are shared. Values assigned into array elements 200 through 399 are owned and remembered by the specific chart that the ESPL program is run on. This allows you to run the same ESPL program on multiple charts and not have a conflict with values in the 200 to 399 element ranges.

EXAMPLE: The following sample program stores some values in the Global Array, and then prints them.

```
begin
  SetGV(1, 25);
  SetGV(2, 50);
  SetGV(3, 75);
  writeln(GV(1), ' ', GV(2), ' ', GV(3));
end;
```

Highest Lowest

SYNTAX: **Highest**(*Type, Index, Period, var BarIndex, Rank: integer, DataSet: integer*): real;
 Lowest(*Type, Index, Period, var BarIndex, Rank: integer, DataSet: integer*): real;

DESCRIPTION: The **Highest** function returns the highest *Type* value and its *BarIndex* within a given range of chart bars. The **Lowest** function returns the lowest *Type* value and its *BarIndex* within a given range of chart bars. The **Highest** and **Lowest** functions can also operate on chart Study values or chart Overlay bars by passing the object number for the study or overlay as the *DataSet* parameter. The Study and Overlay object numbers can be obtained with the **FindStudy** function.

PARAMETERS:

Type: *Type* is one of the following predefined constants:

eArray	eClose	eHigh	eLast	eLow	eMidPoint
eMid3	eMid4	eNet	eOpen	eOpenInterest	ePercent
eRange	eTrueHigh	eTrueLow	eTrueRange	eVolume	
1	2	3	4		

Refer to the **Bar** function for a complete description of these constants.

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart. Both the host and the overlay use the same indexing.

Period: *Period* specifies the number of bars to include the range. The range will include bars from (*Index* - *Period* + 1) through and including (*Index*).

BarIndex: *BarIndex* is a bar index value returned by the function. *BarIndex* indicates the exact bar where the highest or lowest value occurred. If multiple bars have equally high or low values, then *BarIndex* will report the first occurrence it finds (closest to index 1).

Rank: *Rank* is a value from 1 to 5, and causes the functions to report the 1st, 2nd, 3rd, 4th, or 5th highest or lowest value in the range of bars.

DataSet: *DataSet* is used to optionally access data from either a chart Study or a chart Overlay. The host chart's bar data set is used by entering the number 0 as the *DataSet* parameter. Chart Overlay DataSets can be referenced by entering 1, 2, or 3, or the Object number for the overlay. Study data is accessed by using the **FindStudy** command first, and then passing the Study object number as the *DataSet* parameter.

NOTE: The *DataSet* parameter is not optional.

EXAMPLE: The following example opens an IBM daily chart and applies the Relative Strength Index (RSI) study to the chart. The program then finds and prints the 2nd highest High bar value for the last 10 bars, and also the *BarIndex* at which the High occurred. The program then finds and prints lowest RSI study value for the last 20 bars.

```
var                                        {Start of Variable declarations}
  BarIndex, StudyObject: integer;        {Variables declared as Integers}
begin                                     {Start of Main Programming code}
  Chart ('IBM.D');                         {Open an IBM daily chart}
  StudyObject := AddStudy(eRSI);         {Apply RSI study to the chart}
  writeln('Chart High= ', Highest(eHigh, BarEnd, 10, BarIndex, 2, 0));    {Find High bar}
  writeln('Bar Index at High= ', BarIndex);                                {Print message}
  writeln('RSI Low= ', Lowest(1, BarEnd, 20, BarIndex, 1, StudyObject)); {Find Low RSI}
end;                                       {End of program}
```

Holiday

SYNTAX: **Holiday**(*Market*: integer, *Symbol*: string, *Date*: integer): boolean;

DESCRIPTION: The **Holiday** function is used to determine if a specified date is a holiday. This allows you to skip calculations on holidays. This can also be useful if an ESPL program is run by the Scheduler. The program can be aborted if it is a holiday. The Ensign Holiday screen will be used to determine if the date is a holiday or not. Select **Set-Up | Holiday Schedule** from the Ensign main menu to view or edit the holidays.

The **Holiday** function requires three parameters. The format for the *Date* is a long integer bar date. Example YYYYMMDD where the year is counting up from 100. Example: 102=2002. 1021225 = Dec 25, 2002. The function returns a *True* value if the date is a holiday for the specified *Market* group. The function returns a *False* value if the date is not a holiday.

PARAMETERS:

Market: *Market* is one of the following predefined constants:

eADSOOption	eCanadian	eCash	eCorporate	eCustom	eEHSOption
eFOption	eFund	eFuture	eGovernment	eIMSOOption	eIndex
eIndicator	eLFuture	eLiberty	eLStock	eMoney	eMunicipal
eNasdaq	eNQSOOption	eRUSOption	eSpread	eStatistic	eStock
eVZSOOption	eZion				

Symbol: Specifies the *Symbol*. For Stocks, the symbol can be an empty string. For Futures, the symbol will be used to determine if the exchange (for that symbol) is open.
Example: `Holiday(eNasdaq, ' ', DateToLong(Now)) ;`

Date: Use the `DateToLong` command to convert a `TDateTime` into a long date format.
Example: Enter a long integer bar date. 1030704 = July 4th, 2003

EXAMPLE: The following program will report if the current day is a holiday.

```
begin
  if Holiday(eFuture, 'SP H1', DateToLong(Now)) = True then
    writeln('Today is a Holiday')
  else
    writeln('Today is not a Holiday');
end;
```

HTTP

SYNTAX: **HTTP**(*URL*: string, *FileName*: string);

DESCRIPTION: The **HTTP** command is used to read and save the HTML source code from an Internet web page. An active Internet connection is required if accessing web pages from the Internet. The HTML code from the specified *URL* page will be saved to the specified *FileName*. The HTML code can then be loaded, edited, viewed, or used in other programs.

PARAMETERS:

URL: Specifies the web page address to download. The HTML programming code from the specified web page will be downloaded and saved to the *FileName*.

FileName: Specifies the *FileName* and path for the downloaded HTML source code. If no path is specified, then the file will be stored in the Ensign program folder (example: C:\ENSIGN\).

EXAMPLE: The following program uses the **HTTP** command to download and save HTML source code from a web page. The source code from the Ensign Software web site (menus frame) is downloaded and saved in a file named TEST.TXT. The HTML code is then loaded into the output window.

```
begin
  HTTP('http://www.ensignsoftware.com/menu.htm', 'TEST.TXT');
  Finished(5); {give it some time to receive the http data}
  Output(eLoad, 'TEST.TXT');
end;
```

IF..Then..Else

SYNTAX: **IF***ConditionalExpression* **THEN**
 {statement to do if conditional-expression is evaluated to TRUE}
 [**ELSE**
 {statement to do if conditional-expression is evaluated to FALSE}];

DESCRIPTION: The **IF THEN ELSE** statement is used to choose code execution based upon a True or False condition. The statement(s) following **THEN** will be executed if the *ConditionalExpression* is True. The statement(s) following **ELSE** will be executed if the *ConditionalExpression* is False. The **ELSE** statement is optional. If the **ELSE** statement is used, the statement just ahead of the **ELSE** keyword should not be terminated with a semicolon. A block of code can be executed if desired by encasing the code with **Begin** and **End** statements. Each statement in the **Begin...End** block should end with a semicolon.

PARAMETERS:

ConditionalExpression is a logical expression that can be evaluated to a Boolean value of True or False.

EXAMPLE: The following example is run by clicking the RUN button in the script editor. An Inputbox will open and ask you to type the day of the week. A response will be printed, depending on the answer. An **IF THEN ELSE** statement is used to determine the response.

```
var                                     {Start of Variable Declarations}
  Text: string;                         {Text is declared as a string}
begin                                   {Start of Main Programming code}
  Text := InputBox('Work Days','Enter the Day of the Week', ''); {Open Inputbox}
  if (Text='Saturday') or (Text='Sunday') then {If Then}
    writeln('Why are you working today ?') {Print if True}
  else {else}
    writeln(Text, ' is a work day.');
```

Import

SYNTAX: **Import**(*ControlFile*: string, [*Convert, Merge, Close*: boolean]);

DESCRIPTION: The **Import** command is used to open the Import ASCII Data form, populate the form's components, and optionally Convert, Merge and Close the form.

PARAMETERS

ControlFile: Pass the string for the control file combo box. This file sets the form's properties.
Convert: Pass a True to Convert the file. Equivalent to clicking the Convert button. Default is True.
Merge: Pass a True to Merge the file. Equivalent to clicking the Merge button. Default is True.
Close: Pass a True to Close the Import form after the file has been imported. Default is True.

EXAMPLE: The following example opens the Import form, sets the form's 12 parameters, converts, merges and leaves the Import form open when the import process is finished. The 'MyImport' control file would have been previously configured with parameters which successfully convert the Source file data into the Target file.

```
begin                                {Start of Main Programming code}
  Import('MyImport', true, true, false);
end;                                  {End of program}
```

ImageToFile

SYNTAX: **ImageToFile**(*FileName*: string, [*left, top, width, height, form, x, y*: integer]);

DESCRIPTION: The **ImageToFile** command is used to save any window to a .PNG graphics file. The .PNG files can be used to e-mail chart images or to display chart images on web pages. The command will save any window that has the focus. The .PNG file is saved to the file specified by the *FileName* parameter. The folder will be the path specified on the Setup | System | Images form.

Two images can be merged as well. The first image is from the active form, and its size is the rectangle(left,top,width,height) and the 2nd image is specified by *Form*. The overlay position is (x,y);

PARAMETERS

FileName: Specify the *FileName* for the .PNG file.
Left, top, width, height: An optional rectangle can be specified for the size of the image capture. The rectangle is relative to the form's left and top location.
Form: Specify a 2nd image the 1st image will overlay. The final image size is the size of the 2nd form.
x, y: This is the coordinate on the 2nd image where the 1st image is placed.

EXAMPLE: The following example opens an IBM daily chart, and then saves the window image to a .PNG graphics file.

```
begin                                {Start of Main Programming code}
  Chart('IBM.D');                    {Open an IBM daily chart}
  ImageToFile('IBM.PNG');             {Save the window to a .PNG file}
end;                                  {End of program}
```

EXAMPLE: The following example finds a spreadsheet, and then saves a section of its image on a chart. The final image is the composite of both.

```
begin                                {Start of Main Programming code}
  FindWindow(eChart, 'ES #F.30');     {Find a specific chart}
  SetMyFocus;                         {Give the chart focus}
  f := ActiveChild;                  {Set a variable to this chart form}
  FindWindow(eSpread, 'TEST');        {Find a specific spread sheet}
  SetMyFocus;                         {Make the spread sheet the active form}
  {A section of the Spreadsheet is overlaid on the chart at position (0,40)}
  ImageToFile('Spread.PNG', 131, 85, 256, 270, f, 0, 40);
end;                                  {End of program}
```

Index1

Index2

Index3

Index4

Index5

Index6

SYNTAX: **Index1** through **Index6** : integer;

DESCRIPTION: The above Index variables are automatically set when the mouse is clicked on any chart. The variables contain the bar Indexes for the 6 most recent mouse clicks. The most current mouse click is stored in *Index1*. The Index values are all shifted down to the next variable as new mouse clicks are made. The variables are global variables and do not need to be declared. User-Defined Studies and Draw Tools can use these variables to draw lines or make calculations.

EXAMPLE: The following program draws 5 lines on a chart. The lines connect the last 6 mouse clicks. First, use the mouse to click on 6 chart points. Example, mark the high and low swings of a 5 wave Elliott wave series. Then click ESPL button 0 in the Script editor to run the program and draw the lines on the chart. The Y coordinates are converted into 'Price' values for the **AddLine** statement. NOTE: Refer to the documentation for the **PtX1, PtY1** global variables.

```
begin
  FindWindow (eChart) ;
  AddLine (eLine, 0, Index1, YtoPrice (PtY1) , Index2, YtoPrice (PtY2)) ;
  AddLine (eLine, 0, Index2, YtoPrice (PtY2) , Index3, YtoPrice (PtY3)) ;
  AddLine (eLine, 0, Index3, YtoPrice (PtY3) , Index4, YtoPrice (PtY4)) ;
  AddLine (eLine, 0, Index4, YtoPrice (PtY4) , Index5, YtoPrice (PtY5)) ;
  AddLine (eLine, 0, Index5, YtoPrice (PtY5) , Index6, YtoPrice (PtY6)) ;
end;
```

IndexToX

XToIndex

SYNTAX: **IndexToX**(*Index*: integer): integer;
 XToIndex(*X-Coordinate*: integer): integer;

DESCRIPTION: The **IndexToX** function is used to convert a bar *Index* position on a chart, to its *X-Coordinate* horizontal pixel position in the chart window. The **XToIndex** function is used to convert an *X-Coordinate* pixel position to the nearest bar *Index* position on a chart. Both of these commands are useful for translating horizontal screen position values to and from pixel and index positions.

PARAMETERS:

Index: *Index* is the bar position on the chart.

X-Coordinate: The first column of screen pixels (dots) on the left edge of a chart has an *X-Coordinate* of zero. The count increases to the right horizontally.

EXAMPLE: The following example opens an IBM daily chart. The bar *Index* position for the left edge of the chart is printed using the **XToIndex** command. The *X-Coordinate* pixel for the last bar on the chart is printed using the **IndexToX** command.

```
begin                                     {Start of Main Programming code}
  Chart('IBM.D');                       {Open an IBM daily chart}
  writeln(XToIndex(0));                   {Print Index of bar on left edge of chart}
  writeln(IndexToX(BarEnd));             {Print horizontal position of last bar on chart}
end;                                     {End of program}
```

Initialize

SYNTAX: **Initialize**: boolean;

DESCRIPTION: The **Initialize** global variable is used with user defined studies. The variable is True on the 1st call of the study execution, and False for all subsequent executions. Use the flag to know when to initialize study properties.

InputBox InputQuery

SYNTAX: **InputBox**(*Caption, Prompt, Default*: string): string;
 InputQuery(*Caption, Prompt, var Default*: string): boolean;

DESCRIPTION: The **InputBox** function opens an input window. Text can be entered into the window's edit box. The text is returned to the program when the **OK** button is clicked. The *Default* string is returned to the program if the **Cancel** button is clicked. The window *Caption*, and *Prompt* can be specified.

The **InputQuery** function is nearly identical to the **InputBox** function. Text entered into the edit box is returned in the *Default* variable when the **OK** button is clicked. The function returns True if the **OK** button is clicked, and False if the **Cancel** button is clicked.

PARAMETER:

Caption: The *Caption* parameter specifies the window caption for the dialog box.
Prompt: The *Prompt* parameter is the text that prompts the user to enter input in the edit box.
Default: The *Default* parameter is the string that appears in the edit box when the dialog box first appears.

EXAMPLE: The following example uses an **InputQuery** function and a **Repeat...Until** loop to open daily charts for entered symbols. The loop repeats until the **Cancel** button is clicked (returning a False value in *Test*).

```
var                                     {Start of Variable declarations}
  Text: string;                         {Text is declared as a String}
  Test: boolean;                        {Test is declared as a Boolean}
begin                                   {Start of Main Programming code}
  Text := 'IBM';                         {IBM set as default text value}
  repeat                                 {Start of Repeat loop}
    Test:=InputQuery('Open Chart','Enter a Symbol',Text);   {Input symbol}
    if Test then Chart(Text + '.D');     {Open chart}
  until Test=False;                     {Repeat loop until Test=False}
end;                                    {End of program}
```


Insert

SYNTAX: **Insert**(*Text2*, *Text1*: string, *Position*: integer);

DESCRIPTION: The **Insert** command is used to insert *Text2* into *Text1* at the specified character *Position*.

EXAMPLE: This program inserts the word 'buy' into a sentence at the 22nd character. The new sentence is printed.

```
var                                {Start of Variable declarations}
  Text:string;                     {Text is declared as a String}
begin                              {Start of Main Programming code}
  Text:= 'I recommend that you IBM'; {Text is assigned a value}
  Insert('buy ', Text, 22);        {'buy' is inserted into Text}
  writeln(Text);                   {Prints Text}
end;                                {End of program}
```

InsertBar

SYNTAX: **InsertBar**(*Index*: integer [, *Last*, *High*, *Low*, *Open*: real , *Volume*, *Date*, *Time*: integer]): boolean;

DESCRIPTION: The **InsertBar** function is used to insert a bar on a chart. Use **ChartRefresh**(True) to cause the chart to redraw after the bar has been inserted. The bar prices, date, and time can be optionally specified. If they are not specified then the new bar will default to the Date and Last price of the bar just previous to the inserted bar. The **SetBar** command can also be used to adjust the bar data after being inserted.

PARAMETERS:

Index: *Index* is the chart bar position (between 1 and the number of bars on the chart). If *Index* is less than or equal to zero, the function will use *Index* as an offset from the last bar on the chart (BarEnd). If *Index* is out of range, the function will return a False value, otherwise it will return True.

Last, *High*, *Low*, *Open*: Specify prices for each bar price level.

Volume: Specify a volume amount for the new bar.

Date: Specify a date for the new bar.

Time: Specify an Intraday time for the new bar, or an Open Interest amount for a daily Futures chart.

EXAMPLE: The following example opens an IBM daily chart and then inserts a bar for every Calendar day (including weekends). Click ESPL button 1 to run the program.

```
procedure CalendarDays;
var
  i,j,Difference:integer;
  BarDate1, BarDate2 : TDateTime;
begin
  Chart('IBM.D');
  if Finished(10)=True then // Wait until chart is drawn
  begin
    for i:= BarEnd downto 1 do
    begin
      BarDate1:= LongToDate(Bar(eDate,i)); // Get Current Bar Date
      BarDate2:= LongToDate(Bar(eDate,i-1)); // Get Previous Bar Date
```

```

    Difference := BarDate1 - BarDate2;           // Difference between days
    if Difference > 1 then
    for j := Difference-1 downto 1 do
        InsertBar(i, Last(i-1), Last(i-1), Last(i-1), Last(i-1) ,1,
            DateToLong(BarDate2+j) );
    end;
    ChartRefresh(True);
end;
end;

begin
    if ESPL=1 then Calendardays;
end;

```

IntToHex

IntToStr

StrToInt

StrToPrice

SYNTAX: **IntToHex**(*Number*: integer, *Digits*: integer): string;
 IntToStr(*Number*: integer): string;
 StrToInt(*Text*: string): integer;
 StrToPrice(*Text*: string): real;

DESCRIPTION: The **IntToStr** function converts an integer number into a text string. The **IntToHex** function converts an integer number into a hexadecimal (base 16) string value. The **StrToInt** function converts a string value to an integer value. The **StrToPrice** function converts a string representation of a stock price to a Real number (for example, **StrToPrice**('9 5/8') would report 9.625 as the resulting Real number).

PARAMETERS:

Number: Specifies an integer *Number* to convert.
Digits: *Digits* indicates the minimum number of hexadecimal digits to return in the string.
Text: Specifies the *Text* value to convert to an integer.

EXAMPLE:

```

var
    High: integer;           {Start of Variable declarations}
    Text1, Text2: string;   {High is declared as an Integer}
begin
    High:=17;               {Variables declared as Strings}
    Text1:='The High value is ' + IntToStr(High); {Start of Main Programming code}
    Text2:='The Hex value of High is ' + IntToHex(High,6); {High assigned a value of 17}
    writeln(Text1);        {High converted to a string}
    writeln(Text2);        {Convert to hex}
    High:= StrToInt('20'); {Print Text1}
    writeln(StrToPrice('10 5/8')); {Print Text2}
end;                       {Converts '20' to an integer value}
                           {Prints 10.625 }
                           {End of program}

```

IsNumeric

SYNTAX: **IsNumeric**(Text: string, var Number: real): boolean;

DESCRIPTION: The **IsNumeric** function can be used to convert a *Text* string into a *Number* value. The function returns a **True** value if the string is successfully converted to a *Number*, and returns **False** if the conversion failed (and the value of *Number* will be zero). If the string contains characters that will not convert to numbers, then the conversion will fail.

EXAMPLE: The following example attempts to convert two strings into numbers. The first attempt is successful. The second attempt fails since *Text2* cannot be converted into a numeric value.

```
var                                     {Start of Variable declarations}
  Text1, Text2:string;                 {Variables declared as Strings}
  Number: real;                        {Number declared as a Real}
begin                                   {Start of Main Programming code}
  Text1:= '543.78';                    {Text1 assigned a value}
  Text2:= '2000 was an interesting year.'; {Text2 assigned a value}
  IsNumeric(Text1, Number);            {Text1 converted to a Real number}
  writeln(Number);                     {Number is printed}
  IsNumeric(Text2, Number);            {Text2 fails to convert}
  writeln(Number);                     {Printed value is a zero}
end;                                    {End of program}
```

IsSelected

SYNTAX: **IsSelected**: integer;

DESCRIPTION: The **IsSelected** function is used to determine if an ESPL draw tool on a chart is currently selected and active. The object number of the draw tool will be returned. The object number can be used by the **GetStudy**, **SetStudy**, and **Remove** functions. **IsSelected** returns a zero value if an ESPL draw tool is not selected. A draw tool can be selected by clicking the mouse on the draw tool Line. ESPL draw tools will be in a selected state while applying the tool to a chart. You may have programming tasks that you do not want to perform while applying the tool to a chart. Use the **IsSelected** command to determine when to perform your programming actions.

EXAMPLE:

```
procedure DrawMyTool;
begin
  if IsSelected = 0 then
  begin
    {draw your tool}
  end;
end;

begin
  if ESPL=11 then DrawMyTool;
end;
```

IT

DESCRIPTION: The *IT* variable returns text from a selection made from a **Choose** List box. The *IT* variable is also used with the **AlertEvent** function, and reports which numeric event triggered the call to the ESPL engine.

EXAMPLE: The following example uses the **Choose** function to set an **AlertEvent** for three symbols. The *IT* variable reports which symbol is selected from the Choose list. An AlertEvent is created for the selected symbol. The AlertEvent will be triggered each time the symbol trades. The *IT* variable will equal AlertEvent value when the symbol trades (example 61, 62, or 63). Click ESPL button 1 to display the Choose list. Click ESPL button 2 to clear all AlertEvents.

NOTE: A symbol must be placed in the Alert log before an AlertEvent can be set for the symbol. In this program a generic Alert is set for the symbols, before the AlertEvents are set.

```

var                               {Start of Variable declarations}
  Selection: integer;              {Selection is declared as an Integer}
begin                               {Start of Main Programming code}
  if ESPL=1 then begin              {ESPL Button 1 displays the choose list}
    Selection := Choose('Select a Symbol', 'IBM', 'MSFT', 'DELL'); {Display list}
    Output(eClear);                 {Clear output window}
    writeln(Selection, ' ', IT);     {Print Selection value and IT value}
    Alert(IT, FindMarket(IT), 0, 0); {Set a generic Alert for the symbol}
    AlertEvent(eTrade, 60+Selection, IT) ; {Set an AlertEvent for the symbol}
  end;                               {block end}
  if ESPL=2 then AlertEvent(eClear); {Button 2 clears all AlertEvents}
  if ESPL=61 then writeln(IT, ' traded'); {Print IT AlertEvent triggers}
  if ESPL=62 then writeln(IT, ' traded'); {Print IT AlertEvent triggers}
  if ESPL=63 then writeln(IT, ' traded'); {Print IT AlertEvent triggers}
end;
```

KeyDown

SYNTAX: **KeyDown**(Code: integer): boolean;

DESCRIPTION: The **KeyDown** function is used to determine if a specified keyboard key has been pressed, or if a mouse button is being held down. The function will return a True value if the specified keyboard key or mouse button is down when the function is called (or has been pressed since the last call to KeyDown).

PARAMETERS:

Code: *Code* is the ASCII code (65 .. 90) for the letter keys 'A' through 'Z'. Other keys, including the mouse buttons, can be specified using the following table:

1	Left Mouse button	91	LeftWindows Button
2	Right Mouse button	92	RightWindows Button
112	F1	48 & 96	0
113	F2	49 & 97	1
114	F3	50 & 98	2
115	F4	51 & 99	3
116	F5	52 & 100	4
117	F6	53 & 101	5
118	F7	54 & 102	6
119	F8	55 & 103	7
120	F9	56 & 104	8
121	F10	57 & 105	9
122	F11		
123	F12		
8	Backspace	106	*

9	Tab	107	+
13	Enter	109	-
16	Shift	111	/
17	Ctrl	186	;
18	Alt	187	=
19	Break	188	,
20	Caps Lock	189	_
32	Space Bar	190	.
33	Page Up	191	?
34	Page Down	192	`
35	End	219	[
36	Home	220	\
37	Left Arrow	221]
38	Up Arrow		
39	Right Arrow		
40	Down Arrow	144	Num Lock
44	SysRq	145	Scroll Lock
46	Insert		
47	Delete		
96	Ins		
110	Del		

NOTE: The ESC key cannot be tested. Pressing the ESC key will abort an ESPL program. The above codes do not distinguish between shifted and un-shifted keys. The numbers 0-9 have two codes. The first code represents the number keys on the top row of the keyboard. The second code represents the number keys on the 10-keypad.

EXAMPLE: The following program uses a **Repeat...Until** loop to increment and print a counter. The loop continues until one of the SHIFT keys is pressed. The **KeyDown** function is used to determine when the SHIFT key is pressed.

```
begin                                {Start of Main Programming code}
  j:=0;                               {J is assigned the value of zero}
  repeat                               {Repeat loop}
    writeln(j);                       {Print the value of J}
    Pause(1);                         {Pause for 1 second}
    inc(j);                            {Increment J, J=J+1}
  until (KeyDown(16));                {Loop until SHIFT key has been pressed}
  writeln('A Shift Key was pressed'); {Print statement}
end;                                  {End of program}
```

LeftStr

RightStr

ReverseString

SYNTAX: **LeftStr**(Text : string, Count : integer) : string;
RightStr(Text : string, Count : integer) : string;
ReverseString(Text : string) : string;

DESCRIPTION:

LeftStr: Returns the number of characters specified by *Count* (from the Left edge of the string).

RightStr: Returns the number of characters specified by *Count* (from the Right edge of the string).

ReverseString: Returns the *Text* string in reverse order.

EXAMPLE: The following simple program demonstrates some string handling.

```
begin
  writeln( LeftStr('This is a string', 7));  {Prints 'This is' }
  writeln( RightStr('This is a string', 7)); {Prints ' string' }
  writeln( ReverseString('abcdef');        {Prints 'fedcba' }
end;
```

Length

SYNTAX: **Length**(Text : string) : integer;

DESCRIPTION: The **Length** function is used to determine the length of a string. The function will return the number of characters contained in *Text*.

EXAMPLE: The following simple program prints the **Length** of a sentence.

```
begin
  writeln(Length('This is a test.'));  {Prints 15 (the length of the text)}
end;
```

LineTo MoveTo MoveToLineTo

SYNTAX: **LineTo**(X, Y: integer);
 MoveTo(X, Y: integer);
 MoveToLineTo(X1, Y1, X2, Y2: integer);

DESCRIPTION:

LineTo: draws a line on a chart from the current X,Y pen position to the new point specified by X and Y. The new pen position is set to X and Y. The line is drawn using the current pen color.

MoveTo: moves the current pen position to the point specified by X and Y, but no line is drawn. Use the **MoveTo** command to set the pen position prior to using the **LineTo** command.

MoveToLineTo draws a line on a chart between point X1,Y1 and point X2,Y2. **MoveToLineTo** is equivalent to using the two functions **MoveTo** and **LineTo**. The pen position is moved to point X2,Y2. The line is drawn using the current pen color.

PARAMETERS:

X and Y: X and Y specify vertical and horizontal pixel coordinates in the chart window. The top left corner of the chart window will have an X,Y coordinate value of 0,0. The X coordinate specifies horizontal pixels across the screen, starting from the left edge. The Y coordinate specifies vertical pixels down the screen.

EXAMPLE: The following example opens an IBM daily chart. The pen color is varied between Red, White, and Blue. The **MoveTo**, **LineTo**, and **MoveToLineTo** statements are used to draw three lines in different locations on the chart.

NOTE: The lines that are drawn using these commands are not remembered by the chart since they are not line Objects. Use the **AddLine** command to draw permanent chart lines.

```

uses
  Graphics;
begin
  Chart('IBM.D');           {Start of Main Programming code}
  SetPen(clRed);           {Open an IBM daily chart}
  MoveTo(0,10);           {Set pen color to Red}
  LineTo(200,100);        {Move pen position, without drawing}
  SetPen(clWhite,1,eDot);  {Draw a line}
  MoveToLineTo(50,10,250,100); {Set pen to white and dotted}
  SetPen(clBlue,1,eSolid); {Draw a line}
  MoveToLineTo(100,10,300,100); {Set pen to blue and solid}
end;                       {Draw a line}
                           {End of program}

```

Layout

LayoutName

LayoutOpen

SYNTAX: **Layout**(*Name*: string): boolean;
 LayoutName : string;
 LayoutOpen : boolean;

DESCRIPTION: The **Layout** command is used to open an Ensign layout. The **Layout** command allows you to display layouts using the ESPL language. Layouts are created and saved by using the Layout button. For more information, consult the help documentation for Layouts.

LayoutName is a global variable that contains the text name of the most recently opened Layout.

Example: writeln(LayoutName);

LayoutOpen is a global variable that is True when a layout has been opened.

Example: If LayoutOpen then begin end;

PARAMETERS:

Name: *Name* should be the name of the Layout that will be opened. NOTE: Layout files have a file extension of .TXT and are saved in the \Ensign10\Layouts\Layout1..8 folders.

EXAMPLE: The following example uses a **Timer** command to open five different Layouts (based on the time). The program assumes that the Layouts have already been created. Each Layout will remain open for two minutes. Click ESPL button 1 to start the timer. Click ESPL button 2 to stop the timer. Every two minutes a different Layout will open.

NOTE: The *ESPL* variable equals 10 when the **Timer** calls the ESPL program.

```

begin
  if ESPL=1 then Timer(eStart,60,1,10); {Start of Main Programming code}
  if ESPL=2 then Timer(eStop);         {ESPL button 1 starts the Timer}
  if ESPL=10 then begin                {ESPL button 2 stops the Timer }
    s := TimeStr;                       {Run this code every 60 seconds}
    case s[5] of                         {TimeStr returns hh:mm format}
      '0': Layout('BANKING');           {5th character is a minute digit}
      '2': Layout('DOW30');
      '4': Layout('TECH');
      '6': Layout('RETAIL');
      '8': Layout('BIOTECH');
    end;
  end;                                  {end case}

```

```

end;                                {end block}
end;                                {End of program}

```

LowerCase UpperCase UpCase

SYNTAX: **LowerCase**(*Text*: string): string;
 UpperCase(*Text*: string): string;
 UpCase(*Text*: string): string;

DESCRIPTION: The **LowerCase** function converts all letters contained in *Text* into lower case letters. The **UpperCase** function converts all letters contained in *Text* into upper case letters. The **UpCase** function returns only the first letter of *Text* as an upper case letter.

EXAMPLE: The following example prints and converts *Text* to **UpperCase**, then to **LowerCase**, and finally, returns the 1st letter using the **UpCase** command.

```

begin                                {Start of Main Programming code}
  Text:='The NYSE is in New York.';   {Text is assigned a value}
  writeln(Text);                      {Prints Text}
  writeln(UpperCase(Text));           {Print and Uppercase Text}
  writeln(LowerCase(Text));          {Print and Lowercase Text}
  writeln(UpCase(Text));             {Print and Upcase Text}
end;                                  {End of program}

```

Manager

SYNTAX: **Manager**(eSave, *FileName*: string [,*Feed*: integer]): boolean;
 Manager(eLoad, *FileName*: string [,*Feed*: integer]): boolean;
 Manager(eAppend, *FileName*: string [,*Feed*: integer]): boolean;
 Manager(eSymbol, *Symbol*: string [,*Feed*: integer]): boolean;
 Manager(eClear [,*Feed*: integer]): boolean;
 Manager(ePrint [,*Feed*: integer]): boolean;
 Manager(eSet [,*Feed*: integer]): boolean;
 Manager(eOnLine [,*Feed*: integer]): boolean;
 Manager(eOffLine [,*Feed*: integer]): boolean;
 Manager(eRequest [,*Feed*: integer]): boolean;
 Manager(eReset, *ListNumber*: integer [,*Feed*: integer]): boolean;

DESCRIPTION: The **Manager** function allows you to manipulate the Manager symbol list. The **Manager** function will return a **True** value if successful, otherwise it will return a **False** value. The *Feed* parameter selects which vendor feed to work with, and when omitted the default is the feed set in the FEED global variable.

PARAMETERS:

Feed: *Feed* is one of these predefined constants. The default is the value assigned to the FEED global variable.

eFXCM	eIB	eSignal	eIQFeed	eNinja	eOpenECry
eTraderBytes	eTransAct	eGlobal	eDBFX	eATCBrokers	eCustom

Manager(eSave, *FileName*) Save the current symbol list to the specified *FileName*.

Manager (eLoad, <i>FileName</i>)	Load the specified <i>FileName</i> as the current symbol list.
Manager (eAppend, <i>FileName</i>)	Append the specified file contents to the end of the current symbol list.
Manager (eSymbol, <i>Symbol</i>)	Add the specified symbol to the bottom of the current symbol list.
Manager (eClear)	Erase the current symbol list.
Manager (ePrint)	Print the symbol list to the output window.
Manager (eSet)	Copy all text from the output window into the current symbol list.
Manager (eOnLine)	Upload the current symbol list, and request a current update for each symbol.
Manager (eOffLine)	All symbols will stop updating. Unloads all symbols from the Manager list.
Manager (eRequest)	Request an update for all symbols in the current symbol list.
Manager (eReset, <i>ListNumber</i>)	Unload the current symbol list and replace it with the specified symbol list. The Manager window contains numbered Tabs which allow you to change symbol lists. This function is equivalent to clicking a numbered Tab in the Manager window. Specify the Tab number as the <i>ListNumber</i> parameter (a value from 1 to 6).

EXAMPLE: The following example illustrates several **Manager** commands. The current symbol list is taken off-line. The symbol list is cleared. Two symbols are added to the symbol list. The new list is placed back on-line. Click ESPL button 1 to run the program.

```
begin                                {Start of Main Programming code}
  if ESPL=1 then begin                {if ESPL button 1 is clicked then ESPL=1}
    Feed := eSignal;                  {Set the default feed to eSignal}
    Manager(eOffLine);                 {Take the current symbol list off-line}
    Manager(eClear);                   {Clear the current symbol list}
    Manager(eSymbol, 'IBM');           {Add IBM to the list}
    Manager(eSymbol, 'MSFT');          {Add MSFT to the list}
    Manager(eOnLine);                  {Place the new list back on-line}
  end;                                  {end of block}
end;                                    {End of program}
```

Max Min

SYNTAX: **Max**(*Value1*: variant [, *Value2*: variant *Value100*: variant]): variant;
 Min(*Value1*: variant [, *Value2*: variant *Value100*: variant]): variant;

DESCRIPTION: The **Max** function looks at the list of supplied *Values* and returns the item with the highest value. The **Min** function looks at the list of supplied *Values* and returns the item with the lowest value. The list of *Values* may contain up to 100 items of Numeric, String or Boolean type. The items in the list must all be of the same type. For example, strings cannot be mixed with numbers in the list.

PARAMETERS:

Value: The *Value* parameters may be numbers, Strings, or Boolean (True or False) values. Strings are compared by ASCII sequence (alphabetical order).

The **Min** Boolean value is False. The **Max** Boolean value is True.

EXAMPLE: The following program illustrates the **Max** and **Min** functions on Numbers, Strings, and Boolean values.

```
begin                                     {Start of Main Programming code}
  writeln(Max(5,3,9,-1));                 {Prints 9}
  writeln(Min(5,3,9,-1));                 {Prints -1}
  writeln(Max('M','A','K'));             {Prints M}
  writeln(Min('M','A','K'));             {Prints A}
  writeln(Max(True,False));               {Prints True}
  writeln(Min(True,False));               {Prints False}
end;                                       {End of program}
```

Menu Commands

DESCRIPTION: Many Ensign menus can be accessed with the ESPL programming language. Use the *Click* command to run the menu item (as if the menu had been clicked with the mouse). Each menu has an assigned name. The names are shown below. For example, use `mnuCloseWindow.Click` to close the window which has focus. Use `mnuCloseAll.Click` to close all open windows. The caption for each menu item can be changed by using the *Caption* property.

COMMANDS:

Click: *Click* is used to simulate a mouse click on a menu (as if the user had clicked the menu item).
Caption: *Caption* is used to read or set a menu's text caption.

Account	Bars	Charts	Close
mnuTradeReport	mnuBars	mnuProperties	mnuCloseAll
	mnuCandles	mnuSettings	mnuCloseLayer
	mnuSolid	mnuDataPanel	mnuCloseWindow
Alert	mnuZebra	mnuKeyboard	
mnuLog	mnuRockets	mnuToolHotKeys	
mnuList	mnuLine	mnuStudyHotKeys	
mnuAlertEmail		mnuColorBarHotKeys	
		mnuPopupMenu	
		mnuConvertEWTemplates	
Database	Docs	Feeds	Help
mnuDatabaseManager	mnuUsersGuide	mnuDataFeeds	mnuWebSiteHelp
mnuMergeChartFiles	mnuStudies	mnuActivity	mnuAdditionalHelp
mnuDelete1Date	mnuTemplates	mnuPackage	mnuRecentArticles
mnuImportASCIIData	mnuDYOdoc		
mnuRolloverSymbols	mnuDrawTools		
mnuZeroVariables	mnuSpreadDoc		

mnuEraseNeuralNets	mnuESPLdoc		
	mnuKnowledgebase		
Image	Internet	Package	Reports
mnuEnsignServer	mnuDownload	mnuBuild	mnuMemory
mnuFile	mnuData	mnuServer	mnuHardware
mnuEmailImage	mnuEmail	mnuExtract	mnuDYOSTudies
mnuFullScreenToFile	mnuNewUser	mnuPackageSettings	
mnuClipboard			
Main	Printer	Symbol	System
mnuLayout	mnuPrintNormal	mnuIBGuide	mnuSystem
mnuChart	mnuPrintStretched	mnuESignalGuide	mnuSecurity
mnuStack	mnuPrintOriented	mnuIQFeedGuide	mnuScheduler
mnuQuote		mnuTraderBytesGuide	mnuImages
mnuOption	Stack	mnuBarChartGuide	mnuESPL
mnuNews	mnuStack1	mnuDTNGuide	mnuMaintenance
mnuAccount	mnuMDI	mnuYahooGuide	mnuButtons
mnuAlert			
mnuOrderEntry			
mnuSpreadSheet	Theme	Upgrade	Video
mnuChatRooms	mnuLuna	mnuEnsignProgram	mnuIntroduction
mnuDatabaseManager	mnuObsidian	mnuUsersGuideManual	mnuVideoLibrary
mnuESPLEditor	mnuSilver	mnuStudiesManual	mnuWebinarArchive
mnuSnapQuote	mnuBlue	mnuTemplatesManual	mnuYouTubeVideos
mnuSaveLayout	mnuOlive	mnuDYOManual	mnuDownloadedVideo
mnuPrint	mnuGray	mnuDrawToolsManual	
mnuPrinterSetup	mnuClassic	mnuSpreadsheetManual	
mnuResizeRibbon	mnuXP	mnuESPLManual	
mnuExit	mnuWindows7	mnuKnowledgeBaseManual	

Merge

SYNTAX: **Merge**(*SourceFile*, *TargetFile*[, *StartDate*, *EndDate*]: string);

DESCRIPTION: The **Merge** command is used to merge two Chart data files together. The chart data from the *SourceFile* will be merge with the chart data from the *TargetFile*. An optional *StartDate* and *EndDate* can be specified. Chart data that is contained within the dates will be merged. The merge will be performed without any price or volume

adjustments. The merge process can be used to replace erroneous target bars with source bars, fill-in gaps of missing data, create continuation charts, and to extend chart data to include more data.

PARAMETERS:

- SourceFile*: The *SourceFile* specifies the chart file to merge. If a file path is not specified, then defaults will be supplied. Wildcard characters may be used to select a group of files, example: 'A:*.D' If using wildcard characters, the *TargetFile* must be an empty string " (the *TargetFiles* will be automatically determined).
- TargetFile*: The *TargetFile* specifies the chart file to merge with. If a file path is not specified, then defaults will be supplied. If the *TargetFile* is an empty string, then a path and filename will be automatically determined.
- StartDate*: The *StartDate* specifies the starting date for the merge process. If no date is specified, then the entire data file will be merged. The date should have a format of Month, Day, and Year (MM-DD-YY).
- EndDate*: The *EndDate* specifies the ending date for the merge process. The format for the date is MM-DD-YY.
- EXAMPLE: The following program merges an IBM daily chart file from a diskette on the A: drive, into the IBM daily chart file.

```
begin
  Merge('A:\IBM.D', 'IBM.D', '06-01-02', '06-30-02'); {Merges a range of bars}
end;
```

MessageDlg MessageDlgPos

SYNTAX:

MessageDlg(*Message*: string, *Icon*: TMsgDlgType, *Buttons*: TMsgDlgButtons): Integer;
MessageDlgPos(*Message*: string, *Icon*: TMsgDlgType, *Buttons*: TMsgDlgButtons, *Top*, *Left*: integer): Integer;

DESCRIPTION: The **MessageDlg** and **MessageDlgPos** functions are used to open a message box on the screen. A *Message* is displayed in the box. Message boxes are often used to ask a question or to display a warning message. The user closes the message box by clicking on a *Button*. Several *Icons* and *Buttons* are available for use in the message box. *Icons* are used to display a small graphic indicating the type of question or warning that the message box contains. *Buttons* are used to help answer a question or respond to a message. Multiple *Buttons* can be displayed in a message box. The **MessageDlgPos** function can be used to specify the exact pixel location on the screen where the message box will appear (using the *Top* and *Left* parameters). When a *Button* is clicked, the message box will close and the selected button will be returned as a *Result*. The button *Result* can be used in an ESPL program to determine the next program action to perform.

PARAMETERS:

- Top*: Specifies the number of pixels from the top of the screen. The top of the message box will align with this pixel.
- Left*: Specifies the numbers of pixels from the left edge of the screen. The left edge of the message box will align with this pixel.
- Icon*: The following Icons can be displayed in a message box. The icons reinforce the purpose of the message:
mtConfirmation - An icon containing a Question Mark.
mtError - An icon containing an 'X' inside a red circle.

mtInformation - An icon containing the letter 'I'.
mtWarning - An icon containing an Exclamation Mark inside a yellow yield sign.

Buttons: The following buttons can be displayed in a message box. Any combination of buttons can be used.

mbAbort	mbAll	mbCancel	mbIgnore	mbNo
mbOK	mbRetry	mbYes		

Result: When a button is clicked, the function will return one of the following results. Check the result to determine which button was clicked.

mrAbort	mrAll	mrCancel	mrIgnore	mrNo
mrOK	mrRetry	mrYes		

EXAMPLE: The following program opens a message box and asks if you want to exit the Ensign program. If you click YES, then a 2nd message box will open to warn you that the program is closing. Click OK to continue or CANCEL to abort the process. NOTE: The 2nd message box will open at screen pixel position 100, 100.

```
begin
  if MessageDlg('Exit Ensign ?',mtConfirmation,mbYes,mbNo)=mrYes then
    if MessageDlgPos('Program Closing !',mtWarning,mbOK,mbCancel,100,100)=mrOK then
      mnuExit.click;
    end;
end;
```

Mod

SYNTAX: *(Number1 Mod Number2)* :integer

DESCRIPTION: The **Mod** statement is a math operation used to perform Integer Division and return the Remainder as the result. *Number1* is divided by *Number2*. The result is the remainder of the division calculation. The answer is always an integer value (no decimals).

EXAMPLE: The following example performs two integer division calculations. The remainder is returned as the result.

```
var
  X,Y: integer;      {Start of Variable declarations}
begin
  X := 115 Mod 100;  {X and Y are declared as Integers}
  Y := 5 Mod 2;     {Start of Main Programming code}
  writeln(X, ' ',Y); {X is assigned the remainder of 115/100 (remainder=15)}
end;                {Y is assigned the remainder of 5/2 (remainder=1) }
                   {Prints the values of X and Y}
                   {End of program}
```

Mouse

SYNTAX: **Mouse**(var *X, Y*: integer): boolean;

DESCRIPTION: The **Mouse** function is used to determine the current X, Y coordinate position of the mouse on a chart. The function is also used to determine if a mouse button is currently being clicked. When the mouse is over a selected chart window, the **Mouse** function will return a True value if any mouse button is depressed when the function is called. Otherwise, the function returns a False value. The X, Y coordinates of the mouse position are also returned. The horizontal position (X) can be converted to a bar index using the **XToIndex** function. The vertical position (Y) can be converted to a price using the **YToPrice** function. If necessary, use the **FindWindow** command to find the chart before using the **Mouse** function.

PARAMETERS:

X: X is the horizontal screen pixel position of the mouse, with 0 being the left edge of the chart.
Y: Y is the vertical screen pixel position of the mouse, with 0 being the top edge of the chart.

EXAMPLE: The following program opens an IBM daily chart. The program reports the Last price of the selected bar, whenever the mouse is clicked on the chart. Move the mouse to the left edge of the chart to stop the program.

```
var
  x,y,zLast: integer;
  Msg: string;
begin
  Chart('IBM.D');
  repeat
    if Mouse(x,y) then begin
      zLast := Bar(eClose,XtoIndex(X));
      Msg := 'Last = ' + FormatPrice(IntToStr(zLast)) + '    ';
      TextOut(100,20,Msg);
    end;
    Application.ProcessMessages;
  until (X>0) and (X<20);
end;
```

NewsFind

NewsStory

NewsText

NewsTitle

NewsSymbol

SYNTAX: **NewsFind**(*Keyword*: string): integer;
NewsStory(*Index*: integer, var *Story*: string): boolean;
NewsText(*Keyword*: string): boolean;
NewsTitle(*Index*: integer): string;
NewsSymbol(*Symbol*: string, *Days*, *Count*: integer): string;

DESCRIPTION:

- NewsFind:** This function is used to find a news story containing a specified *Keyword*. It is equivalent to clicking the **Find** button in a news window. A news window is opened and all the stories containing the *Keyword* in the news title will be listed. The function returns an Integer number representing the number of stories found.
- NewsStory:** The **NewsStory** function is used to display a news story from the news title list. The titles are referenced by sequential *Index*. The *Index* for the first story in the list would be a 1. The news title list can be established by using the **NewsFind** function. The **NewsStory** function returns a **True** value if the referenced story displays properly, or a **False** value if the story could not be found or decoded properly. The text for the story is returned in the *Story* string variable. If desired, the *Story* text can be assigned into a String List variable using the string list *Text* property.
- NewsText:** The **NewsText** function searches the currently displayed story and returns a **True** value if the search *Keyword* is found in the story. A **False** value is returned if the *Keyword* is not found.

NewsTitle: The **NewsTitle** function returns a news title from the news title list. The news titles are referenced by sequential *Index*. For example, `NewsTitle(1);` will retrieve the title text for the first news title in the list.

NewsSymbol: The **NewsSymbol** function should only be used by 'eSignal' and 'Dtn IQFeed' data-feed users. If you do not subscribe to one of these data-feeds then disregard this function. **NewsSymbol** allows you to download news headlines based on a Stock Symbol. News headlines that have reference to the indicated stock will be downloaded. The number of *Days* to download can be specified. The number of news headlines (*Count*) to download can also be specified. The function returns TRUE if the request was made, and FALSE if the *Symbol* parameter is missing.

PARAMETERS:

Keyword: The *Keyword* specifies the word to use in a news search. The search is not case sensitive.
Index: *Index* is used to reference the news titles in a news title list. The titles are referenced in order from 1 to the end.
Story: The *Story* variable will contain the news story text after using the **NewsStory** function. The variable needs to be declared as a string before using the **NewsStory** function.
Symbol: Specifies the Stock Symbol to retrieve news for.
Days: Specifies how many days of news to download. The default is 14 days.
Count: The maximum number of news headlines to download. The default is 100.

EXAMPLE: The following program searches for all news stories containing 'IBM' in the title. A **FOR** loop is used to display each of these stories, and then print the titles of all stories that contain the word 'Stock'. The news window is then closed. A list of the stories containing 'IBM' in the title and 'Stock' in the story text will be printed in the output window.

```
var                                     {Start of Variable declarations}
  i,Count: integer;                     {Variables declared as Integers}
  Story: string;                         {Story is declared as a String}
begin                                    {Start of Main Programming code}
  Output(eClear);                        {Clear the output window}
  Count:=NewsFind('IBM');                 {Find stories with IBM in title}
  for i:=1 to Count do begin             {Loop through the found stories}
    if NewsStory(i,Story)=True then      {Open each story}
      if NewsText('Stock') then writeln(NewsTitle(i));           {Print titles}
    end;                                  {block end}
  if NewsSymbol('IBM',10,100) then writeln('IBM headlines downloaded');
  mnuCloseWindow.click;                 {Close the news window}
end;                                      {End of program}
```

Now

SYNTAX: **Now:** TDateTime;

DESCRIPTION: The **Now** function returns the current date and time.

EXAMPLE: The following program prints the current date and time (example: 7/23/2002 10:01:23 AM)

```
begin
  writeln('The current date and time is ',Now);
end;
```

Output

SYNTAX: **Output**(eClear): boolean;
 Output(eClipboard): boolean;
 OutPut(ePaste); boolean;
 Output(ePrint [, *Orientation*: boolean]): boolean;
 Output(eSave, *FileName*: string): boolean;
 Output(eLoad, *FileName*: string): boolean;
 Output(eAppend, *FileName*: string): boolean;
 Output(eSort [, *Start*: integer, *Ascending*: boolean]): boolean;

DESCRIPTION: The ESPL editor uses the output window for printing test and results from programs. The **Writeln** and **Write** commands are often used to print text to the output window. The **Output** function is used to clear, print, save, load, append, and sort the contents of the output window. The output window can be displayed with the `btnOutputWindow.click`. It is automatically displayed by the **Writeln** and **Write** commands.

Output(eClear) erases the contents of the output window.

Output(eClipboard) copies the contents of the output window to the Windows Clipboard.

Output(ePaste) will paste the contents of the Windows Clipboard to the output window.

Output(ePrint, *Orientation*) prints the output window to the printer. The *Orientation* parameter should be a `True` or `False` value. `True` will print in portrait mode. `False` will print in landscape mode (sideways). The default is portrait mode.

Output(eSave, *FileName*) saves the output window contents to the specified file. The file will be saved in the `\ENSGN10` sub-directory. This command allows you to save the contents of the output window to a file on the hard disk.

Output(eLoad, *FileName*) loads the specified file's text into the output window. The specified file must reside in the `\ENSGN10` sub-directory. This is a convenient way to load and display an ASCII text file in the output window.

Output(eAppend, *FileName*) appends the contents of the output window to the specified file. The contents are appended to the end of the file. The file must reside in the `\ENSGN10` sub-directory. This allows you to add text to an existing text file.

Output(eSort, *Start*, *Ascending*) sorts the output window lines of text. The sort can be in ascending or descending order. The sort process can sort from any character position in a line. The *Start* parameter specifies the character position to start the sort on. The default is 1. The *Ascending* parameter should be a `True` or `False` value. `True` will sort in ascending order. `False` will sort in descending order. The default is ascending. The lines of text in the output window will be rearranged based upon the specified sort criteria.

EXAMPLE: The following program responds to 5 ESPL button clicks. Click ESPL button 1 to generate a list of 20 random alphabet letters and numbers in the output window. Click ESPL button 2 to sort the output window in ascending order. Click ESPL button 3 to sort the output window in descending order, sorting from the 2nd character on each line. Click ESPL button 4 to print the output window contents to the printer. Click ESPL button 5 to save the contents of the output window to a file named 'Random.txt'. These ESPL buttons are on the Run ESPL form or on the toolbar on the ESPL editor.

```
var                                     {Start of Variable declarations}
  i: integer;                           {i is declared as an Integer}
begin                                    {Start of Main Programming code}
  if ESPL=1 then begin                  {Run this if button 1 is clicked}
    Output(eClear);                     {Clear the output window}
    for i:= 1 to 20 do writeln(Chr(65+Random(26)),Random(10)); {Random print}
  end;                                   {block end}
  if ESPL=2 then Output(eSort,1,True);  {button 2 sorts ascending}
  if ESPL=3 then Output(eSort,2,False); {button 3 sorts descending}
  if ESPL=4 then Output(ePrint,True);   {button 4 prints the window}
```



```

    if ESPL=5 then Output (eSave, 'Random.txt');      {button 5 saves to a file}
end;                                                {End of program}

```

Pause

SYNTAX: **Pause**(*Seconds*: real);

DESCRIPTION: The **Pause** command causes the ESPL program to pause for a specified number of *Seconds*. The program will suspend execution of the script for the specified number of seconds. The Ensign program is still processing data feeds during the script pause.

EXAMPLE: The following program opens a custom Quote page, pauses for 5.5 seconds, and then closes the Quote page.

```

begin
    Quote (eCustom);
    Pause (5.5);
    mnuCloseWindow.Click;
end;

```

Pi

SYNTAX: *Pi*;

DESCRIPTION: *Pi* is a global math variable equal to 3.14159265358. It can be used in calculations where *Pi* is required.

EXAMPLE: The following program calculates the area of a circle that has a Radius of 10.

```

var                                {Start of Variable declarations}
    Area,Radius: real;              {Variables declared as Reals}
begin                                {Start of Main Programming code}
    Radius := 10;                   {Radius assigned a value of 10}
    Area := Pi * Sqr(Radius);        {Area =  $\pi * r^2 = 314.16$ }
    writeln('The Area= ',Area);      {Print the Area to the output window}
end;                                  {End of program}

```

Play

SYNTAX: **Play**(*FileName*: string);

DESCRIPTION: The **Play** command is used to play a sound file by using the Media Player to play the specified file. The Media Player will not be visible.

PARAMETERS: The *FileName* specifies the sound file play. The *FileName* should be a .WAV, .MID, or .RMI file type. If a *FileName* is not specified, then the CHORD.WAV file is played. If the file is not found, then a beep will be played.

EXAMPLE: The following program opens an IBM daily chart. The TADA.WAV file is played if the current bar's High is higher than the previous bar's high. Otherwise, the CHORD.WAV file is played.

```

begin
    Chart (' IBM.D' );

```

```

if High(BarEnd) > High(BarEnd-1) then
  Play('C:\WINDOWS\MEDIA\TADA.WAV')
else
  Play('C:\WINDOWS\MEDIA\CHORD.WAV');
end;

```

Pos

SYNTAX: **Pos**(*Text1*: string, *Text2*: string): integer;

DESCRIPTION: The **Pos** function is used to find a string within another string. The function searches for *Text1* within *Text2*. The function will return a number, representing the starting character position where *Text1* was found in *Text2*. For example, `Pos('New', 'Happy New Year');` will return the number 7 since the word 'New' starts in the 7th character position. The function will return a zero value if *Text1* is not found in *Text2*.

PARAMETERS:

Text1: Any text string value.
Text2: Any text string value.

EXAMPLE: The following program starts a **Timer** by clicking on ESPL button 1. The Timer will **Play** a sound file at the top of each hour. The **Pos** function is used to determine if the Time contains the numbers '00' (example: 03:00). Click ESPL button 2 to stop the Timer. NOTE: The value of the *ESPL* variable is set to 10 when a Timer calls an ESPL program.

```

begin
    {Start of Main Programming code}
    if ESPL=1 then Timer(eStart,60,10); {button 1 starts a 60-second Timer}
    if ESPL=2 then Timer(eStop);       {button 2 stops the Timer}
    if ESPL=10 then                    {Run this code each Timer interval}
    if Pos('00',TimeStr)>0 then Play('C:\WINDOWS\MEDIA\TADA.WAV'); end;
{End of program}

```

Power

SYNTAX: **Power**(*Number*, *Exponent*: real): real;

DESCRIPTION: The **Power** function raises a *Number* to an *Exponent* power. For example, **Power**(2,3) raises the number 2 to the 3rd power (i.e. $2 * 2 * 2 = 8$). NOTE: The **Sqr** function can be used to square numbers.

PARAMETERS:

Number: Specifies a number to raise to a power.
Exponent: Specifies the power to raise a number by.

EXAMPLE: The following program uses a **FOR** loop to print the value of 5 raised to the powers of 0 through 9.

```

begin
  for i:= 0 to 9 do writeln('5^',i,' = ',Power(5,i));
end;

```

Pred Succ

SYNTAX: **Pred**(*Number*): variant;
 Succ(*Number*): variant;

DESCRIPTION: The **Pred** function is used to return the value of *Number*-1. The **Succ** function is used to return the value of *Number*+1. The value of *Number* is not changed in either case. The predecessor or successor of a *Number* can be used in a variety of programming tasks. The **Pred** and **Succ** functions are more efficient and faster than simply programming *Number*+1 or *Number*-1 in the programming code. Any valid *Number* can be used in the parameter.

EXAMPLE: The following program loads a custom quote page file named 'CUSTOM.QUO' into a String List. A **FOR** loop is used to open a daily chart window for each symbol in the list. Each symbol is also printed in the output window. The **Pred** function is used to help determine the last Index item for the String List. NOTE: The Index for String Lists starts at zero, but the Count starts at 1. The `Pred(sList.Count)` statement references the last item in the string list (since Count always equals 1 more than the last index). NOTE: Symbols that are saved in quote files include a leading character that indicates the market group. The leading character must be stripped off to obtain just the Symbol.

```
var                                     {Start of Variable declarations}
  i: integer;                           {i is declared as an Integer}
  Symbol: string;                        {Symbol is declared as a String}
begin                                    {Start of Main Programming code}
  FindWindow(eScript);                  {Find the Script Editor Window}
  mnuMinimize.Click;                    {Minimize the Script Editor}
  Output(eClear);                        {Clear the output window}
  sList.LoadFromFile(sPath + 'QuoFile\Custom.dat'); {Load quote file into sList}
  for i:= 0 to pred(sList.Count) do begin {Loop through the Symbols}
    Symbol:= Copy(sList.Strings(i),2,8); {Strip 1st character to get Symbol}
    if Symbol > '' then begin           {if a valid Symbol then proceed}
      writeln(Symbol);                  {Print the Symbol}
      Chart(Symbol + '.D');             {Open a daily chart for the Symbol}
    end;                                 {end of block 2}
  end;                                   {end of block 1}
  mnuTileVertical.Click;                {Tile chart windows on the screen}
end;                                     {End of program}
```

PriceToY YToPrice

SYNTAX: **PriceToY**(*Price*: real): integer;
 YToPrice(*Y-Coordinate*: integer): real;

DESCRIPTION: The **PriceToY** function is used to convert a chart *Price* level, to its *Y-Coordinate* vertical pixel position. The **YToPrice** function is used to convert a *Y-Coordinate* pixel position to the nearest *Price* level on a chart. Both commands are useful for translating vertical screen position values to and from pixel and price values.

PARAMETERS:

Price: *Price* specifies the price value from a chart to convert into a *Y-Coordinate* screen pixel value.

Y-Coordinate: The top row of screen pixels (dots) has a *Y-Coordinate* value of zero. The count increases vertically down the screen. Each pixel can be converted to the nearest price value.

EXAMPLE: The following program opens an IBM daily chart. A line is drawn between the first and last bars on the chart. The **PriceToY** function is used to obtain screen pixel locations for use by the **MoveToLineTo** function. The **IndexToX** function is used to obtain screen pixel locations for the horizontal position. The *Price* value at Y-pixel 200 is then printed.

```

var                                     {Start of Variable declarations}
  Y1,Y2: integer;                       {Y1 and Y2 are declared as Integers}
begin                                   {Start of Main Programming code}
  Chart('IBM.D');                       {A daily chart is opened for IBM}
  Y1:= PriceToY(Last(BarLeft));          {Y-pixel found for 1st bar on chart}
  Y2:= PriceToY(Last(BarEnd));          {Y-pixel found for last bar on chart}
  MoveToLineTo(IndexToX(BarLeft),Y1,IndexToX(BarEnd),Y2); {Draw line}
  writeln(FormatPrice(YToPrice(200))); {Convert Y-pixel 200 to a price}
end;                                     {End of program}

```

Pt1X, Pt2X, Pt3X, Pt1Y, Pt2Y, Pt3Y

SYNTAX: **Pt1X**: integer;
 Pt1Y: integer;
 Pt2X: integer;
 Pt2Y: integer;
 Pt3X: integer;
 Pt3Y: integer;

DESCRIPTION: The above X,Y screen coordinate variables are automatically set when an ESPL Draw Tool is placed on a chart. ESPL Draw Tools are generally drawn on a chart with a series of 2 or 3 mouse clicks. The X,Y position of each mouse click is saved in the above variables. NOTE: The X,Y coordinate system starts in the top-left corner of the chart at pixel position 0,0. X-Coordinates increment horizontally across the screen. Y-Coordinates increment vertically down the screen. For example, an X,Y coordinate of (100,200) is 100 pixels right and 200 pixels down from the top-left corner of the chart. The values of each X and Y point can be used in ESPL programs.

EXAMPLE: The following ESPL Draw Tool is applied to a chart by clicking ESPL button 1 (*ESPL=11*) in the Draw Tools panel. Three points are then selected on the chart with the mouse. A line will be drawn between the 3 points, creating a triangle. The 3 points can be moved around the chart with the mouse to adjust the size and position of the triangle. The location of the 3 selected points are automatically stored in the predefined global X,Y variables (Pt1X, Pt1Y, Pt2X, Pt2Y, Pt3X, Pt3Y). These variables are used to draw the lines with the **MoveToLineTo** statement. NOTE: Click on any corner of the triangle to reactivate the tool, and move the points to a different location.

```

procedure DrawTriangle;                 {Declares the 'DrawTriangle' subroutine}
begin                                   {Start of the sub-routine code}
  MoveToLineTo(Pt1X,Pt1Y,Pt2X,Pt2Y);   {Draw 1st leg of the triangle}
  MoveToLineTo(Pt2X,Pt2Y,Pt3X,Pt3Y);   {Draw 2nd leg of the triangle}
  MoveToLineTo(Pt3X,Pt3Y,Pt1X,Pt1Y);   {Draw 3rd leg of the triangle}
end;                                     {End of the sub-routine code}

{***Main Program***}
begin                                   {Start of Main Programming code}
  if ESPL=11 then DrawTriangle;        {Call the 'DrawTriangle' procedure}
end;                                     {End of program}

```

PtX1, PtX2, PtX3, PtX4, PtX5, PtX6 PtY1, PtY2, PtY3, PtY4, PtY5, PtY6

SYNTAX: **PtX1 through PtX6** : integer;
 PtY1 through PtY6 : integer;

DESCRIPTION: The above X,Y screen coordinate variables are automatically set when the mouse is clicked on any chart. The variables contain the screen coordinates for the 6 most recent mouse clicks. The variables are global variables and do not need to be declared. User-Defined Studies and Draw Tools can use these variables to draw lines or make calculations. The most current mouse click is stored in the PtX1 and PtY1 variables. They are all shifted down to the next variable as new mouse clicks are made. NOTE: The X,Y coordinate system starts in the top-left corner of the chart at pixel position 0,0. X-Coordinates increment horizontally across the screen. Y-Coordinates increment vertically down the screen. For example, an X,Y coordinate of (100,200) is 100 pixels right and 200 pixels down from the top-left corner of the chart. The values of each X and Y point can be used in ESPL programs.

EXAMPLE: The following program draws 5 lines on a chart. The lines connect the last 6 mouse clicks. First, use the mouse to click on 6 chart points. Example, mark the high and low swings of a 5 wave Elliott wave series. Then click ESPL button 1 in the Script editor to draw the lines on the chart. The X,Y coordinates are converted into 'Index' and 'Price' values for the **AddLine** statement. NOTE: See also the **Index1** through **Index6** global variables.

```
procedure DrawWaves;
begin
  FindWindow(eChart);
  AddLine(eLine,0,XtoIndex(PtX1),YtoPrice(PtY1),XtoIndex(PtX2),YtoPrice(PtY2));
  AddLine(eLine,0,XtoIndex(PtX2),YtoPrice(PtY2),XtoIndex(PtX3),YtoPrice(PtY3));
  AddLine(eLine,0,XtoIndex(PtX3),YtoPrice(PtY3),XtoIndex(PtX4),YtoPrice(PtY4));
  AddLine(eLine,0,XtoIndex(PtX4),YtoPrice(PtY4),XtoIndex(PtX5),YtoPrice(PtY5));
  AddLine(eLine,0,XtoIndex(PtX5),YtoPrice(PtY5),XtoIndex(PtX6),YtoPrice(PtY6));
end;

{****Main Program****}
begin                                     {Start of Main Programming code}
  if ESPL=1 then DrawWaves;              {Call the 'DrawWaves' procedure}
end;                                       {End of program}
```

Quote

SYNTAX: **Quote**(*Feed*: integer [, *Symbol*: string, *Flag*: boolean]): integer;
 Quote(eCustom [, *PageName*: string, *Flag*: boolean]): integer;

DESCRIPTION: The **Quote** function is used to open and display a quote window. A specific *Feed* quote page can be specified. The quote page can be instructed to locate a specific *Symbol* at the top of the page (when the page opens). Use **Quote**(eCustom, *PageName*) to display a custom quote page (where *PageName* is the name of the custom quote page). The **Quote** function will return the window handle for the quote window that is opened. The global *Window* variable is also internally set to the window number.

PARAMETERS:

Feed: *Feed* is one of these predefined constants.

eFXCM	eIB	eSignal	eIQFeed	eNinja	eOpenECry
eTraderBytes	eTransAct	eGlobal	eDBFX	eATCBrokers	eCustom

Symbol: *Symbol* specifies the symbol to display on the top row of a quote page.

Flag: The *Flag* parameter must have a True or False value. If *Flag* is True then the **Quote** function will change the contents of a previously opened quote window. If *Flag* is False then a new quote window will be opened. NOTE: Use the **FindWindow** command to locate previously opened Quote windows.

PageName: Use the *PageName* parameter is opening a Custom Quote page. Enter the name of the custom page.

EXAMPLE: The following program opens an eSignal quote page, with MSFT as the top symbol. The quote page is displayed for 5 seconds. A Custom Quote page named CUSTOM is then opened (in the same quote window). The first 10 symbols from the page are printed in the output window. The quote page is closed after 5 seconds.

```
var                                     {Start of Variable declarations}
  Row: integer;                         {Row is declared as an Integer}
  Symbol: string;                       {Symbol is declared as a String}
begin                                   {Start of Main Programming code}
  Quote(eSignal, 'MSFT');               {Open eSignal quote page}
  Pause(5);                             {Pause for 5 seconds}
  Quote(eCustom, 'Custom', True);       {Open Custom quote page}
  for Row:=1 to 10 do begin              {Loop through first 10 rows}
    Symbol:= GetCell(Row,0);            {Get symbol from quote row}
    writeln('Row ',Row, ' ',Symbol);    {Print symbol to output window}
  end;                                   {end of loop code}
  Pause(5);                             {Pause for 5 seconds}
  mnuCloseWindow.Click;                 {Close the quote window}
end;                                     {End of program}
```

Random Randomize

SYNTAX: **Random**(*Number*: integer):integer;
Randomize;

DESCRIPTION: The **Random** function returns a random number between 0 and *Number*-1. For example, **Random**(10) will generate a random number from 0 to 9. The **Randomize** command initializes the random number generator (using the system clock). Call the **Randomize** command before using the **Random** function to avoid possible duplications in the random number generation.

EXAMPLE: The following program uses a **FOR** loop to generate and print 25 random numbers. The **Randomize** command is used to initialize the random number generator before using the **Random** function.

```
begin                                   {Start of Main Programming code}
  Randomize;                             {Initialize Random numbers}
  for i:= 1 to 25 do writeln(Random(100)); {Loop,Generate, and Print numbers}
end;                                     {End of program}
```

Rectangle RoundRect

SYNTAX: **Rectangle**(x1,y1,x2,y2: integer);
RoundRect(x1,y1,x2,y2, *Width*, *Height*: integer);

DESCRIPTION: The **Rectangle** command is used to draw a rectangle on a chart. The X,Y coordinates should specify the top-left and bottom-right corners of the rectangle. The rectangle will be drawn using the current brush and pen color attributes. Use the **SetPen** and **SetBrush** commands to change the pen and brush attributes. NOTE: Screen pixel coordinates start in the top-left corner of the chart window at 0,0. X-coordinates specify horizontal pixels moving to the right. Y-coordinates specify vertical pixels moving down. The **RoundRect** command draws a rectangle with rounded corners. The top-left corner of the rectangle is at point x1,y1, and the bottom-right corner is at point x2,y2. The rounded corners are drawn as segments of an ellipse with a width of *Width* and a height of *Height*. NOTE: The rectangles drawn by these commands are not remembered by the chart. The lines will disappear if the chart is redrawn or rescaled.

EXAMPLE: The following program opens an IBM daily chart. Two rectangles are drawn on the chart. The pen and brush attributes are then changed. Two additional rectangles are drawn using the new attributes.

```
uses
  Graphics;
begin
  Chart ('IBM.D');
  Rectangle (50,50,200,90);
  RoundRect (200,210,400,300,50,50);
  SetPen (clWhite,1,eDot);
  SetBrush (clWhite,eClear);
  Rectangle (100,100,500,200);
  RoundRect (210,50,400,90,50,50);
end;
```

Register

SYNTAX: **Register**(*List*: integer, *StudyName*: string [; *ESPL*: integer]);

DESCRIPTION: This command is used to add custom studies to the Ensign lists. The custom *StudyName* will appear in the same list as Ensign standard studies, color bars and draw tools. The registration process adds the *StudyName* to the Study, ColorBar or Draw Tool lists, and remembers a *ESPL* value for when the custom study is selected from the list. A previously registered *StudyName* can be removed from the drop-down list by omitting the *ESPL* parameter, or by passing -1 as the value. If *StudyName* is already registered, its current *ESPL* value will be reassigned to the new *ESPL* parameter value.

PARAMETERS:

List: *List* is one of these predefined constants: eStudy, eColorBars, eDrawTool

StudyName: *StudyName* is the text that will display in the ColorBar or Study drop-down list. *StudyName* will also be used in the Study Data panel and the Chart Objects list for the registered study.

ESPL: This is the value that will be assigned to the *ESPL* variable for the indicated Study, ColorBar or Draw Tool.

EXAMPLE: Click ESPL button 1 to Register the following programs. A new study named 'Average Range' is added to the Ensign study list. When the study is activated, the ESPL program will be called with a *ESPL* value of 155. A study named 'Average Volume' is added to the Ensign study list, with a *ESPL* value of 156. The 'Gap' ColorBar study is removed from the Ensign ColorBar study list. The Ensign 'Relative Strength' study is reassigned with a *ESPL* value of 200. The sub-routine procedures for each of the new studies are shown below (with no code). However, programming code could be added to each routine that would respond to the newly registered studies. See the [SetUser](#) statement for an example of a user defined study.

```
procedure AveRange;
```

```

begin
end;

procedure AveVolume;
begin
end;

procedure NewRSI;
begin
end;

begin
    if ESPL=1 then begin
        Register(eStudy, 'Average Range',155);
        Register(eStudy, 'Average Volume',156,2);
        Register(eColorBars, 'Gap');
        Register(eStudy, 'Relative Strength',200);
    end;
    if ESPL=155 then AveRange;
    if ESPL=156 then AveVolume;
    if ESPL=200 then NewRSI;
end;

```

{Start of Main Programming code}
 {Click button 1 to run this code}
 {add study to Study List}
 {add study 2nd row of Study List}
 {remove Gap ColorBar study}
 {reassign RSI to use *ESPL*=200}
 {end of block}
 {Run AveRange if *ESPL*=155}
 {Run AveVolume if *ESPL*=156}
 {Run NewRSI if *ESPL*=200}
 {End of program}

Regression

SYNTAX: **Regression**(*Type, Index, Period, DataSet*: integer, var *Slope*, var *StdError*: real): real;

DESCRIPTION: The **Regression** command is used to calculate Linear Regression (Best Fit) values on a chart. A linear regression value is calculated using chart data as input. The **Regression** function returns the linear regression value for the chart bar referenced by *Index*. The *Period* parameter specifies how many bars to use in the calculation. The slope of the linear regression line is returned in the *Slope* variable, and the Standard Error of Estimate is returned in the *StdError* variable.

The **Regression** function can calculate a linear regression value based on a chart's study or overlay *DataSet* by passing the study or overlay's object number as the *DataSet* parameter, or by passing a number 1, 2, 3 ... for the 1st, 2nd, 3rd ... overlay. If necessary, the study or overlay object numbers can be obtained using the **FindStudy** function.

PARAMETERS:

Type: *Type* is one of the following predefined constants:

eArray	eClose	eHigh	eLast	eLow	eMidPoint
eMid3	eMid4	eNet	eOpen	eOpenInterest	ePercent
eRange	eTrueHigh	eTrueLow	eTrueRange	eVolume	
1	2	3	4		

Refer to the **Bar** function for a complete description of these constants.

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart. Both the host and the overlay use the same indexing.

Period: *Period* is the number of bars to use in the calculation. The data points will include bars from *Period - Index + 1* through and including *Index*.

DataSet: *DataSet* is an optional object number for an overlay data set. The default is to use the chart's bar data set by passing a value of zero.

Linear Regression Formula

Regression := Offset + Slope * x

n := Period

Slope := (n * Sum(x*y) - Sum(x) * Sum(y)) / (n * Sum(x^2) - Sum(x)*Sum(x))

Offset := (Sum(y) - Slope * Sum(x)) / n

StdError := sqrt(Sum(sqr(y - (Offset + Slope * x))) / (n-2))

EXAMPLE: A User-Defined Study named 'RegressionLine' is created in the following program. The study can be applied to a chart by clicking ESPL button 100 on the Run ESPL panel. The study calculates and draws a moving linear regression line on the chart.

```
procedure RegressionLine;                                {RegressionLine procedure is declared}
var                                                       {Start of Variable declarations}
  Slope, StdError: real;                                  {Variables declared as Real}
  i:integer;                                             {Variable declared as an Integer}
begin                                                    {Start of sub-routine code}
  SetUser(ePlot1,True);                                  {Specifies that line is drawn and shown}
  SetUser(eShow1,True);
  SetUser(eName, 'Regression');                          {User-Defined study is given a name}
  for i:= BarBeginLeft to BarEnd do                    {Loop through visible bars on the chart}
    SetUser(1,Regression(eLast,i,10,0,Slope,StdError),i); {Calc Regression}
end;                                                     {End of RegressionLine procedure}

begin                                                    {Start of Main Programming code}
  if ESPL=100 then RegressionLine;                      {if ESPL=100 run RegressionLine}
end;                                                     {End of program}
```

Remove

SYNTAX: **Remove**(*ObjectID*: integer, [*Recalc*: boolean]): boolean;
 Remove(*Type*: integer): boolean;

DESCRIPTION: The **Remove** command is used to remove lines and studies from a chart. Each study or line object on a chart has a unique *ObjectID* identification number (sometimes called a *Handle*). If you know the *ObjectID*, then the object can be specifically removed by providing the *ObjectID* as the parameter. The **FindStudy** command can be used to obtain an *ObjectID*. The *ObjectID* is also returned whenever the **AddStudy**, **AddStudyOnStudy**, **AddLine**, and **AddNote** functions are used. The **Remove** function returns a *True* value if successful, and *False* if the object was not found.

PARAMETERS:

ObjectID: *ObjectID* is the object number returned by the **AddStudy**, **AddStudyOnStudy**, **AddLine**, **AddNote**, and **FindStudy** functions. The number is used to specifically identify and remove an object from a chart.

Recalc: This optional parameter can be added with a *True* value if you want the ESPL program to be recalculated after the object is removed from the chart.

Type: *Type* is one of the following predefined constants. These can be used to remove groups of objects.
 eAll eArrow eLine eNote eStudy eESPL

Remove(eAll) removes all Lines, Arrows, Notes, Labels, Studies, and User-Defined studies from a chart.

Remove(eArrow) removes all Arrow (and Marker) objects from a chart.

Remove(eCircle) removes all Circle objects from a chart.

Remove(eNote) removes all Note objects from a chart.

Remove(eLine) removes all Line objects (including Arrows, Notes, and Labels) from a chart.
Remove(eStudy) removes all Studies from a chart (except User-Defined studies).
Remove(eESPL) removes all ESPL studies from a chart.

EXAMPLE: The following program opens an IBM daily chart, adds an RSI and Stochastic study to a chart. The program pauses for 10 seconds and then removes the Stochastic study from the chart. NOTE: The *ObjectID* for the Stochastic study is saved in the *Handle2* variable. This variable is used to help identify the study when removing it.

```
var                                {Start of Variable declarations}
  Handle1, Handle2: integer;      {Variables declared as Integers}
begin                              {Start of Main Programming code}
  Chart ('IBM.D');                {Open an IBM daily chart}
  Handle1 := AddStudy(eRSI);      {Add RSI study, remember the study ID}
  Handle2 := AddStudy(eSto);      {Add Stochastic, remember the study ID}
  Pause(10);                      {Pause for 10 seconds}
  Remove(Handle2);                {Remove the Stochastic study}
end;                               {End of program}
```

Repeat...Until

SYNTAX: **Repeat**
 {block of statements}
 Until (*ConditionalExpression*);

DESCRIPTION: The **Repeat...Until** statements are used to create a loop that is executed until a expression becomes True. The *ConditionalExpression* is evaluated at the end of each loop. If the expression is False, then the block of statements is re-run. If the expression is True, then the loop is terminated and the ESPL program continues with the next line of code. Since the *ConditionalExpression* is not tested until the bottom of the loop, each statement in the loop will get executed at least once.

EXAMPLE: The following program uses a **Repeat...Until** loop to input numbers from the keyboard. The entered numbers are printed in the output window. The loop continues until the user enters the number 5. The program beeps when finished.

```
begin                                {Start of Main Programming code}
  repeat                              {Initiate Repeat loop}
    Value := InputBox('Hello','Enter a number. Enter 5 to Exit.','0');
    writeln('You entered ',Value);      {Print number}
  until (Value='5');                  {Loop Until Value=5}
  writeln('Done...');                 {Print 'Done' message}
  beep;                               {Beep}
end;                                  {End of program}
```

ResetTrades

SYNTAX: **ResetTrades**(*Commission*: real; *BuyArrow*, *SellArrow*, *OutArrow*: integer; *Boxes*: boolean);

DESCRIPTION: The **ResetTrades** statement is used to define specific Trading System values. The statement is used in conjunction with the **Trade**, and **TradeReport** commands. **ResetTrades** is used to initialize a Trading System. The statement prepares a file to receive the Buy and Sell trades specified by the **Trade** command. The *Commission* per contract or per share is specified. The style of the Arrows (marking the trades on a chart) is also specified.

PARAMETERS:

Commission: *Commission* is the per contract or per share commission rate (dollar amount) for a round-trip trade.

BuyArrow, SellArrow, OutArrow:

The Buy, Sell, and Out arrows mark trades on a chart. The arrows can be one of the following types.

0 = eUpArrow	4 = eOut	Add eOut to print the word 'Out' next to the arrow
1 = eDownArrow	8 = eLong	Add eLong to print the word 'Long' next to the arrow
2 = eLeftArrow	12 = eShort	Add eShort to print the word 'Short' next to the arrow
3 = eRightArrow	255 = eNone	eNone disables the arrows, no arrows will be displayed.

Up Arrows will be positioned below the low of the referenced bar.

Down Arrows will be positioned above the high of the referenced bar.

Left Arrows will be positioned on the right side of the referenced bar, at the Trade Price level.

Right Arrows will be positioned on the left side of the referenced bar, at the Trade Price level.

The Price level and the bar Index are automatically supplied by the **Trade** command.

Boxes: Set the value of *Boxes* to True to display a small box at all entry and exit points. Set the value to False to eliminate the boxes.

EXAMPLE: The following program opens an IBM daily chart and runs a Trading System based on the crossing of two Simple Moving Average lines. The system will buy when the average lines cross up. The system will sell when the average lines cross down. The **ResetTrades** command is used to initialize the trading system with no commissions and specific trade arrows. A **FOR** loop and the **GetStudy** command are used to loop through the bars and determine when the lines cross. The **Trade** command is used to create each trade. A **TradeReport** is printed in the output window when the system is finished. Arrows will mark each trade on the chart.

```
var                                     {Start of Variable declarations}
  i, Handle:integer;                   {Variables are declared as Integers}
begin                                   {Start of Main Programming code}
  Chart('IBM.D');                      {Open an IBM daily chart}
  Handle := AddStudy(eAve,1,5,20);      {Add 2 Simple Moving Averages to chart}
  ResetTrades(0,0,1,3,True);           {Initialize Trading System parameters}
  for i:= BarBeginLeft to BarEnd do    {Loop through bars looking for trades}
  begin                                  {start of loop code}
    if GetStudy(Handle, 9,i) then Trade(eBuy+eIf+eReverse,i); {Look for buys}
    if GetStudy(Handle,10,i) then Trade(eSell+eIf+eReverse,i); {Look sells}
  end;                                   {end of loop code}
  Trade(eOut);                          {Close-out the last open trade}
  TradeReport(True);                    {Print the results in the output window}
end;                                     {End of program}
```

SaveToAscii

SYNTAX: **SaveToAscii**(*ExportFormat*: integer [, *FileName*: string]);

DESCRIPTION: The **SaveToAscii** command is used to export Quote pages or Chart bar data to ASCII text files. The files can be exported into a variety of ASCII (text) formats. The exported data can be used by other programs (like spreadsheets or word processors). The **SaveToAscii** command performs the same function as the **SaveToAscii** menu item for Charts and Quote pages. The exported file for a Quote page will save to the \ENSGN sub-directory, and have a file extension of .TXT. The exported ASCII file for Chart data can be saved to a specified path and *FileName*. If the *FileName* parameter is not supplied, then the file will save to the \ENSGN sub-directory, and have a file extension of .TXT, .PRN, or .CSV.

PARAMETERS:

ExportFormat: To export the active Quote page, use one of the following predefined constants:
eExcelColumns eExcelCommas eMetaStock5 eMetaStock7 eAIQ

To export the active Chart, use one of the following predefined constants:
eExcelColumns eExcelCommas eIDOHLC eCSV

NOTE: The Excel Columns format has 10 character wide columns. The Excel Commas format separates the price fields with commas. The IDOHLC format exports the Date, Time, Open, High, Low, Close, Volume, and Open Interest (separated with commas). Select **Set-Up | Computer** from the menu to specify the number of columns to save.

FileName: The *FileName* parameter is used for saving Chart files. Specify the path and filename for the ASCII file.

EXAMPLE: The following program opens a custom quote page and then saves the quote page data to an ASCII file. The file is saved in a comma delimited format that is compatible with a MS-EXCEL spreadsheet. The quote page data is saved to a file named \Ensign10\CUSTOM.TXT. The program then opens an IBM daily chart and exports the bar data to a file named \Ensign10\IBM.TXT.

```
begin
  Quote (eCustom, 'Custom');
  SaveToAscii (eExcelCommas);
  Chart ('IBM.D');
  SaveToAscii (eIDOHLC, sPath + 'IBM.TXT');
end;
```

Scheduler

SYNTAX: *SCRIPT = FileName*
 SCRIPT = FileName ESPL
 SCRIPT = FileName TIMER Seconds

DESCRIPTION: The Ensign Scheduler can be used to load an ESPL program file and run a program at a specified time. The Scheduler window is accessed by selecting **Set-Up | System | Scheduler** from the Ensign menu. The *ESPL FileName* and *ESPL* variable should be specified in the Scheduler. The ESPL Timer may also be started if necessary.

PARAMETERS:

FileName: Specifies the ESPL file to load into the Script Editor.

ESPL: Specifies the value of the *ESPL* variable. The program that responds to the indicated *ESPL* value will be run.

Seconds: Specifies the number of *Seconds* for the *TIMER* interval. Each time the *TIMER* concludes a time interval, the ESPL program will be called with a *ESPL* value of 10. This allows you to program items which will run every interval.

EXAMPLE: The following examples illustrate how to make entries in the Scheduler to open and run ESPL programs at the specified times. NOTE: You can program and save ESPL program files with any *FileName* that you choose. The following examples presume that *REPORTS.PSC*, *SCANS.PSC*, and *STOCKS.PSC* are available ESPL files on your computer.

√ 15:15 *SCRIPT = REPORTS* The *REPORTS.SPT* file is loaded into the ESPL script editor window at 15:15. The *ESPL* value defaults to a value of zero.

- √ 12:30 SCRIPT = SCANS 5 The SCANS.SPT file is loaded into the ESPL script editor window at 12:30. The program responding to *ESPL=5* is run.
- √ 08:30 SCRIPT = STOCKS TIMER 60 The STOCKS.SPT file is loaded into the ESPL script editor window at 08:30. The ESPL Timer is started with a 60 second timer interval. The ESPL program is called every 60 seconds with a *ESPL* value of 10.

NOTE: It is not necessary to add the .psc file extension to the *FileName* in the Scheduler. All ESPL files should be stored in the \Ensign10\ESPL sub-directory. The *ESPL* variable is used to specify which program to run. If *FileName* is already loaded into the Script Editor, the Scheduler will not reload the file. Instead, the Scheduler will set the *ESPL* variable or Timer and run from the loaded ESPL file. This allows you to run different programs from the already loaded ESPL file.

Screen

SYNTAX: *Screen.Property*

DESCRIPTION: A global variable named *Screen* allows you to get and set screen properties. The variable allows you to change the Cursor type for a particular window. The variable allows you to get and set the Width and Height of a window. The screen *ActiveControl*, *ActiveForm*, *Font*, *FormCount*, *Forms*, *PixelsPerInch*, and *Tag* can also be determined.

PROPERTIES:

- ActiveControl*: *ActiveControl* indicates which control currently has input focus.
- ActiveForm*: *ActiveForm* indicates which form currently has focus.
- Cursor*: Specifies the mouse pointer image.
- Fonts*: *Fonts* is a string list of names for all fonts supported.
- FormCount*: *FormCount* is the number of forms displayed on the screen.
- Forms*: *Forms* is a list of all the forms that are currently displayed.
- PixelsPerInch*: The number of screen pixels that make up a logical inch vertically.
- Height*: *Height* is the vertical size of the screen in pixels.
- Width*: *Width* is the horizontal size of the screen in pixels.
- Tag*: *Tag* can be used to store and retrieve an integer value associated with the screen.

CURSOR TYPES: The following cursor types are available.

crArrow	crCross	crDefault	crDrag	crHourglass
crIBeam	crSize			

EXAMPLE: The following program is a mixture of commands which demonstrate different uses of the *Screen* variable. *Screen* commands are often used in conjunction with *TForm* functions. A 'Form' is simply an open window on the screen. Each form has a name and various properties that can be changed. The following code shows how to change the cursor type, window colors, and *WindowState*. The program will *Minimize* any open Chart windows. Various *Screen* values are printed in the output window. The *Tag* property is set and then read back.

```
var                                {Start of Variable declarations}
  xWindow: TForm;                  {xWindow is declared as a TForm}
begin                               {Start of Main Programming code}
  Screen.Cursor := crHourGlass;    {Change the mouse cursor to an HourGlass}
  xWindow := Screen.ActiveForm;    {xWindow is assigned the active form}
  xWindow.Color := clRed;          {Set the background color to Red}
  Pause(5);                        {Pause for 5 seconds}
```

```

xWindow.Color := clWhite;           {Set the background color to White}
for i := 0 to Screen.FormCount-1 do {Loop through all open forms}
begin                               {start of loop}
  xWindow := Screen.Forms[i];      {xWindow is assigned to an open form}
  writeln(i, ' ', xWindow.Name);    {The name of the form is printed}
                                   {All charts are minimized}

  if pos('Chart', xWindow.Name) > 0 then xWindow.WindowState:=wsMinimized;
end;                                 {end of loop}
writeln();                          {Print a blank line}
writeln('Pixels=',Screen.PixelsPerInch); {Print the Pixels per inch}
writeln('Height=',Screen.Height);    {Print the screen height in pixels}
writeln('Width =',Screen.Width);     {Print the screen width in pixels}
writeln('Tag   =',Screen.Tag);       {Print the value of the screen Tag}
Screen.Tag := 9;                    {Set the Tag value to 9}
writeln('NewTag=',Screen.Tag);      {Print the value, now equal to 9}
Pause(5);                           {Pause for 5 seconds}
Screen.Cursor := crDefault;         {Change the mouse cursor back to default}
end;                                 {End of program}

```

sCustom

SYNTAX: **sCustom** : string;

DESCRIPTION: The **sCustom** variable contains the *FileName* of the currently loaded custom quote page. The variable is set by Ensign when a quote page is opened and a custom quote page selected. This variable can only be read; it cannot be set by ESPL.

EXAMPLE: The following program loads and prints the symbols from an open custom quote page. Note that an extra character appears in front of each symbol in the list. The character represents which Market Group the symbol belongs to.

```

begin                               {Start of Main Programming code}
  Output (eClear);                  {Clear the output window}
  Quote (eCustom);                  {Open a custom quote page}
  sList.LoadFromFile(sPath + 'QuoFile\'+sCustom); {Load current quote page file}
  writeln(sList.text);              {Print the symbol list}
  mnuCloseWindow.Click;            {Close the custom quote page}
end;                               {End of Program}

```

Select

SYNTAX: **Select**(*ConditionalExpression*: boolean, *TrueValue*: variant, *FalseValue*: variant): variant;
Select(*Index*: integer, *Value1*, *Value2*: variant [...*Value99*: variant]): variant;

DESCRIPTION: The **Select** function is used to choose between some values based upon a *ConditionalExpression* or an *Index* value. The first parameter is used to select which of the *Values* to return. If the *ConditionalExpression* is True, then the *TrueValue* is returned. If the *ConditionalExpression* is False, then the *FalseValue* is returned. If the *Index* value is between 1 and 99, then the value corresponding to the *Index* is returned.

PARAMETERS:

ConditionalExpression: This is a logical expression that can be evaluated to a Boolean value of True or False.
TrueValue: The value that will be returned if *ConditionalExpression* is True.
FalseValue: The value that will be returned if *ConditionalExpression* is False.

Index: Specifies the *Index* position of the value to be returned.
Value: The value that will be returned, based upon its position in the list of values.
 NOTE: The list of values may contain up to 100 parameters of any type.

EXAMPLE: The following program contains a User-Defined study named ChartLine. Open a chart and then click ESPL button 100 on the Run ESPL panel to apply the study on the chart. A line is drawn on either the Open or Last price of each bar (depending on which is higher). The Select statement is further illustrated by two additional examples. Click ESPL button 1 on the Run ESPL panel to select the 5th Value from the list. Click ESPL button 2 on the Run ESPL panel to print the Day of the week from the supplied list.

```

procedure ChartLine;                                {ChartLine is declared as a Procedure}
var                                                  {Start of Variable declarations}
  i: integer;                                       {i is declared as an Integer}
  Value: real;                                       {Value is declared as a Real}
begin                                               {Start of Procedure code}
  SetUser(eName, 'ChartLine');                       {Name the study ChartLine}
  SetUser(eClose, 'True');                           {Calculate at close of bar only}
  SetUser(ePlot1, True);                             {Display the Line and its value}
  SetUser(eShow1, True);
  for i:= BarBegin to BarEnd do                     {Loop through all the bars}
  begin                                             {start of loop}
    Value := Select(Open(i) > Last(i), Open(i), Last(i)); {which is higher}
    SetUser(1, Value, i);                           {Store value, causes line to Plot}
  end;                                             {end of loop}
end;                                               {End of Procedure}

begin                                               {Start of Main Programming Code}
  if ESPL=100 then ChartLine;                        {If ESPL=100 then run ChartLine}
  if ESPL=1 then writeln(Select(5, 'Hello', True, 5, 4.4, 3*4)); {Print 5th item}
  if ESPL=2 then                                     {Print Day of Week}
    writeln('Day=', Select(DayOfWeek(Now), 'Su', 'M', 'Tu', 'W', 'Th', 'F', 'Sa'));
end;                                               {End of program}

```

Section

SYNTAX: **Section**(*SectionNumber*: integer [, *Text*: string, *BackColor*, *FontColor*: integer]);

DESCRIPTION: The **Section** command allows you to print some text at the bottom of a chart window. There are 7 sections at the bottom of a chart window that can be used. Each section can have a unique Background color and FontColor. The *SectionNumber* specifies which section to print the *Text* to. Place commas in the *Text* message to break the message into multiple rows. If 'False' is entered as the *SectionNumber*, then all the sections will hide. The *BackColor* and *FontColor* are used to specify colors for the section.

EXAMPLE: The following sample program prints several messages in the sections at the bottom of the chart. Click ESPL button 1 on the Run ESPL form to display the sections. Click ESPL button 2 to hide the sections.

```

begin
  if ESPL=1 then begin
    FindWindow(eChart);
    Section(1, 'Row 1, Row 2, Row 3, Row 4', clRed, clBlack);
    Section(2, 'Stop at 23', clLime, clBlack);
    Section(3, 'Call Broker', clAqua, clBlack);
    Section(4, 'Signal at 22', clYellow, clBlack);
    Section(5, 'Profit at 25', clGray, clWhite);
    Section(6, 'Golf at 3PM', clGreen, clWhite);
  end;
end;

```

```

    Section(7, 'Hi', clBlue, clYellow);
end;

if ESPL=2 then begin
    FindWindow(eChart);
    Section(False);
end;
end;

```

SendKeys

SYNTAX: **SendKeys**(*Keys*: string);

DESCRIPTION: The **SendKeys** command is used to imitate keystrokes from the keyboard. Key commands are sent to the computer, as if you had actually typed them. **SendKeys** generates windows messages, which go to a message queue. It may be necessary to have Application.ProcessMessages; following **SendKeys** if forms are being changed before the intended form receives the keystrokes.

PARAMETERS:

Keys: *Keys* specifies the keyboard sequence to perform.

The following characters send the Alt, Ctrl, and Shift keys.

- & Alt key down. Holds the Alt key down while the next character is sent. This is used to access menu hot-keys. Menu hot-keys are not case sensitive. Example: &F is the same as pressing Alt-F. NOTE: Use {Alt} if you want a full keystroke of the Alt key.
- ^ Ctrl key down. Holds the Ctrl key down while the next character is sent. Example: ^C is the same as pressing Ctrl-C.
- ~ Shift key down. Holds the Shift key down while the next character is sent. Example: ~{Tab} is the same as pressing Shift-Tab.

The following items can be used to send the indicated keys.

{F1}	{F5}	{F9}	{Alt}	{Esc}	{Left}	{Return}
{F2}	{F6}	{F10}	{Backspace}	{End}	{PgDn}	{Right}
{F3}	{F7}	{F11}	{Del}	{Home}	{PgUp}	{Tab}
{F4}	{F8}	{F12}	{Down}	{Ins}	{PrtSc}	{Up}

EXAMPLE: The following program opens an IBM daily chart and uses **SendKeys** to place two studies on the chart. The **SendKeys** command is then used to access the File menu and open a News window.

```

begin
    Chart('IBM.D');      {Open an IBM daily chart}
    SendKeys('NRS');    {N=clears all studies, R=adds RSI, S=adds Stochastics}
    Pause(5);           {Pause 5 seconds}
    SendKeys('&FN');     {Alt-F=open File Menu, N=open News window}
end;

```

EXAMPLE: The following program will loop through all open charts and request a refresh for each. A chart pop-up menu has hot keys for refreshing, and SendKeys is issuing a hot key sequence.


```

var
  xForm: TForm;
begin
  for i := 0 to Screen.FormCount-1 do      {Loop through all open forms}
  begin                                    {start of loop}
    xForm := Screen.Forms[i];             {xForm is assigned to an open form}
    xForm.BringToFront;                   {change the z-order position}
    if pos('Chart', xForm.Name) > 0 then  {window found is a chart}
    begin                                  {begin block}
      FindWindow(eChart);                 {set window variable to this chart}
      SetMyFocus;                         {chart needs focus for keyboard}
      SendKeys('&2');                      {Alt-2 refresh menu, refresh 2 days}
      writeln(i, ' ', GetVariable(eName)); {log some feedback}
      pause(3);                          {wait 3 seconds for refresh}
    end;                                  {end of block}
  end;                                    {end of loop}
end;
end;

```

SetArray

SYNTAX: **SetArray**(*Index*: integer, *Value1*: variant [, *Value2* , *ValueN*: variant]);

DESCRIPTION: The **SetArray** statement is used to store values into the *vArray* global array. *vArray* is a single dimension variant array with a lower boundary of zero. *vArray* must be dimensioned before it is used. Use the **DimArray** command to set the upper boundary of the array. *vArray* can hold values of any variable type. See the documentation on *vArray* for more details on array usage. **SetArray** is used to store a value in *vArray* at position *Index*. When more than one value is provided, they fill the array sequentially beginning at position *Index*.

EXAMPLE: The following example dimensions *vArray* with an *UpperLimit* of 10. The array is filled with random numbers and then printed.

```

begin                                {Start of Main Programming code}
  DimArray(10);                       {Dimensions vArray for 10 elements}
  for n := 0 to 10 do SetArray(n, Random(100)); {Assigns random}
  for n := 0 to 10 do writeln(vArray(n));    {Prints the elements of the array}
end;                                  {End of program}

```

SetBar

SYNTAX: **SetBar**(*Field*: string, *Index*: integer, *Value*: real): boolean;

DESCRIPTION: **SetBar** is used to set bar values and colors on a chart. The *Field* parameter is used to specify which bar value will be modified. The *Index* parameter specifies which bar to modify. The *Value* parameter is used to specify the new value. NOTE: Use the **ColorBars**(eNone) command to prevent Ensign layouts from clearing bar coloring applied by the **SetBar** command.

PARAMETERS:

Field: The *Field* parameter can be one of the following predefined constants:

eColor - Sets the Bars Color.	SetBar(eColor, BarEnd, clBlue);
eColorBars – Same as using eColor.	SetBar(eColorBars, BarEnd, clBlue);
eColorVolume - Sets the Volume Color.	SetBar(eColorVolume, BarEnd, clBlue);

eColorAsk - Sets the Ask/Bid Color.	SetBar(eColorAsk, BarEnd, clBlue);
eDate - Sets the Bars Date.	SetBar(eDate, BarEnd, 1041225);
eDateTime - Sets the Bars Time/Date.	SetBar(eDateTime, BarEnd, Now);
eHigh - Sets the Bars High.	SetBar(eHigh, BarEnd, 125.33);
eInterest - Sets the Bars Open Interest.	SetBar(eInterest, BarEnd, 553232);
eLast - Sets the Bars Last Price.	SetBar(eLast, BarEnd, 55.43);
eLow - Sets the Bars Low Price.	SetBar(eLow, BarEnd, 45.00);
eOpen - Sets the Bars Open Price.	SetBar(eOpen, BarEnd, 50.00);
eVolume - Sets the Bars Volume.	SetBar(eVolume, BarEnd, 70000);

Index: An Each bar on a chart is stored in an array from 1 to the last bar on the chart. The *Index* parameter is used to specify which bar to modify. If *Index* is less than or equal to zero, the function will use as an offset from the last bar on the chart. If *Index* is out of range, the function will return zero. An *Index* value of 100 will modify the 100th bar on the chart.

Value: The *Value* parameter specifies the new value for the bar *Field*.

When *Field* is eColor, the *Value* parameter may be one of the following numbers (indicating a bar color). These bar color values can be set on the chart's property form.

- 0 = Normal
- 1 = Bullish color
- 2 = Bearish color
- 3 = Big Cross color
- 4 = Volume color
- 5 = OpenInt color
- 6 = Grid color
- 254 = Background color. NOTE: Setting a bar's color to the background color hides a bar.

A predefined color constant can be used. Example: clRed

NOTE: clBlack has a value of 0. Using it would color the bar with the Normal color instead of black.

EXAMPLE: The following program demonstrates how to change the color of chart bars. An IBM daily chart is opened and a 9 period Simple Moving Average study is applied to the chart. The program then loops through the bars and colors the bars red or green. If the closing price of the bar is below the moving average, then the bar is colored red. If the closing price of the bar is above the moving average, then the bar is colored green.

```

uses Graphics;
begin
    Chart('IBM.D');
    AveHandle := AddStudy(eAve,1,9,1);
    for I := BarBegin to BarEnd do
    begin
        if Last(i) >= GetStudy(AveHandle,1,i) then
            SetBar(eColor,i,clLtGreen)
        else
            SetBar(eColor,i,clRed);
        end;
    end;
    ChartRefresh(True);
end;
```

{Start of Main Programming code}
 {Open an IBM daily chart}
 {Apply Moving Average to the chart}
 {Loop through bars}
 {start of loop code}
 {Test if Last>=Average value}
 {Color the bar green if greater}
 {Color the bar red is less than}
 {end of loop code}
 {Refresh the chart to show colors}
 {End of program}

SetBrush

SetPen

SYNTAX: **SetBrush**(*Color*: integer, *Style*: integer);
 SetPen(*Color*: integer [, *Thickness*: integer, *Style*: integer, *Mode*: integer]);

DESCRIPTION: **SetBrush** is used to set the *Color* and *Style* of the Brush object. **SetPen** is used to set the *Color*, *Thickness*, *Style*, and *Mode* of the Pen object. Both the Pen and Brush properties are used when drawing lines and objects on a chart.

PARAMETERS:

Color: *Color* may be a predefined color constants or a hex BlueGreenRed number:

Style: The Pen *Style* can be eSolid, eDot, eThickness (Dashed Line) or eHidden.
 The Brush *Style* can be eSolid, eHorizontal, eVertical, or eClear. The default is eSolid. eClear will cause the drawn object to be transparent and not fill with a color. eHorizontal and eVertical will fill-in the object with horizontal or vertical lines.

Thickness: *Thickness* specifies the thickness of the line in pixels. The default is 1 pixel wide.

Mode: The Pen *Mode* can be set to pmXOR or pmCopy. Set the Pen Mode to determine how the color of the pen interacts with colors and lines already on the chart. pmXOR draws a line using a combination of colors in either pen or chart background, but not both. If two lines are drawn in the same place, they will disappear. pmCopy draws a line using the Pen color specified in the Color parameter. The lines will always show.

EXAMPLE: The following program changes the Pen and Brush settings to illustrate different effects when drawing objects on a chart.

```
uses
  Graphics;
begin
  Chart('IBM.D');           {Start of Main Programming Code}
  Pause(2);                 {Open an IBM daily chart}
  {Set Pen color to White, thickness to 1 pixel, Style to Dotted}
  SetPen(clWhite, 1, eDot);
  {Brush color to White, style to Clear (affects the fill style)}
  SetBrush(clWhite, eClear);
  {Draw a Rectangle (white, dotted lines, and transparent fill)}
  Rectangle(50,10,250,110);
  {Set Pen color to Yellow, thickness to 10 pixels, style to Solid}
  SetPen(clYellow, 10, eSolid);
  Ellipse(50,110,250,210);   {Draw an Ellipse (yellow, solid)}
  {Set Pen color to White, thickness to 1 pixel, and style to Dashed}
  SetPen(clWhite, 1, eThickness);
  Ellipse(100,110,200,210);  {Draws a Circle}
  {Set Pen color to Blue, thickness to 2 pixels, style to Solid}
  SetPen(clBlue, 2, eSolid);
  Arc(50,10,250,110,50,60,250,10);  {Draw an Arc (blue, solid lines)}
  {Set Brush to Red with Horizontal line fill}
  SetBrush(clRed, eHorizontal);
  Pie(200,10,400,110,400,110,400,10);  {Draw a Pie (red, horizontal line fill)}
  {Set Pen color to Red, thickness to 3 pixel, and style to Dotted}
  SetPen(clRed, 3, eDot);
  {Set Brush to White with Vertical line fill}
  SetBrush(clWhite, eVertical);
  Chord(300,100,600,200,500,100,400,200);  {Draw a Chord with Vertical filling}
end;                          {End of program}
```

SetData

SYNTAX: **SetData**(*Field*: integer, *Price*: real): boolean;
 SetData(*eName*: constant, *Text*: string): boolean;

DESCRIPTION: The **SetData** function is used to change or edit values for a quote symbol. Use the **Find** command to find and retrieve a quote record. **SetData** will change values in the last retrieved quote record.

PARAMETERS:

Field: The *Field* parameter can be one of the following predefined constants:
 eAsk eBid eClose eHigh eInterest
 eLast eLow eName eOpen eVolume
 eYesterday

Price: The *Price* parameter specifies the new value to store in the record.

Text: The *Text* parameter is used to set the *Name* field in a quote record. Up to 10 characters can be assigned.

EXAMPLE: The following program resets the Volume for IBM to 50000 shares, and changes the Name field to 'I.B.M.'.

```
begin
  Find(eSignal, 'IBM');
  SetData(eVolume, 50000);
  SetData(eName, 'I.B.M. ');
end;
```

SetDateTime

SYNTAX: **SetDateTime**(*Year* [, *Month*, *Day*, *Hour*, *Minute*, *Second*]: integer);

DESCRIPTION: The **SetDateTime** command will set the Date and Time values on the computer clock. Use this command if you ever need to change the computer's date and time settings. NOTE: To set the Time but not the Date, pass zero values for the *Year*, *Month* and *Day* parameters. To set the Date but not the Time, omit the *Hour*, *Minute* and *Second* parameters.

PARAMETERS:

Year: *Year* is in the format of yyyy (example: 2002).
Month: *Month* is in the range of 1-12.
Day: *Day* is in the range of 1-31.
Hour: *Hour* is in the range of 0-23.
Minute: *Minute* is in the range of 0-59.
Second: *Second* is in the range of 0-59.

EXAMPLE: The following program sets the computer date and time to December 10th, 2010 at 3:00 pm.

```
begin
  SetDateTime(2010, 12, 10, 15, 0, 0);
end;
```

SetMyFocus

SYNTAX: **SetMyFocus**([*ShowLayer*: boolean]);

DESCRIPTION: The **SetMyFocus** command causes a window to receive the active focus. Once a window has the focus, then the menus, features, and buttons associated with that window can be accessed. **SetMyFocus** will make the window referenced by the global *Window* variable the active window with the focus. **SetMyFocus** will bring a window in a stack to the surface of the stack. The optional *ShowLayer* flag passed as a True will change layers if the window is on a different layer than the one being viewed. The default for *ShowLayer* is False.

EXAMPLE: The following program assumes that several charts and quote pages are open on the screen (including an IBM daily chart). The program finds the IBM daily chart and sets the focus on the chart window. Once the window has the focus, the window is printed using the **btnPrint** command.

```
begin                                     {Start of Main Programming code}
  FindWindow(eChart, 'IBM.D');           {Find the IBM daily chart window}
  SetMyFocus;                             {SetMyFocus on the chart window}
  btnPrint.click;                         {Print the chart}
end;                                       {End of program}
```

SetLength

SYNTAX: **SetLength**(var *StringVariable*: string, *NewLength*: Integer);

DESCRIPTION: The **SetLength** command is used to resize a string variable. The *NewLength* parameter specifies the new length of the string. Any existing characters in the string are preserved, but the contents of any newly allocated space is undefined. If there is not enough memory available to reallocate the string, an EOutOfMemory error will occur.

PARAMETERS:

StringVariable: A string variable should be entered as the parameter. The variable should have been declared as a string.

NewLength: The *NewLength* parameter should be a number that specifies the length of the *StringVariable*. For example, a value of 16 would cause the *StringVariable* to have a length of 16 characters.

EXAMPLE: The following program declares a string variable named *Text*. The variable is then filled with alphabet letters. The **SetLength** command is used to resize the *Text* variable to 10 characters.

```
var
  Text: string;
begin
  Text:= 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  writeln(Text);
  SetLength(Text,10);
  writeln(Text);
end;
```

SetLine

SYNTAX: **SetLine**(*Handle*, *Row*, *CheckBox1*, *PercentLevel*, *Color*, *Style*, *LeftSide*, *RightSide*, *CheckBox2*: integer);

DESCRIPTION: The **SetLine** command is used to change the color, style, and markers for a specified DrawTool line. The command will only change the settings for the specified *Row* on the Properties window. Some Draw Tools have as many as 11 *Rows* on their Properties window. The **SetLine** command allows you to change the color, style, and markers after the Draw Tool has already been placed on a chart. The **SetStudy** command can also be used to change colors, styles, and markers. The **SetLine** command changes just 1 *Row* at a time.

PARAMETERS:

Handle: *Handle* is the line Object ID number (Handle) returned by the **FindStudy**, **AddStudy**, and **AddLine** functions. If the value of *Handle* is set to Zero, then the **SetLine** command will default to the first line object it finds. Use the **FindStudy** command to find a Draw Tool on a chart, and then use the returned *Handle* as the *Study* parameter when using the **SetLine** command.
Example: `Handle:= FindStudy(eFibonacci);` {Find the Fib study on the chart. *Handle* identifies the lines}
`SetLine(Handle,1,0,clRed,1,22,23);` {Change the settings for *Row* 1 in the Fib Properties Window}

Row: Enter the *Row* of the Properties window that you want to change. Make sure to count blank rows. Some properties windows do not have entry boxes on every *Row*.

CheckBox1:
CheckBox2: Enter a True value to place a check mark in the first CheckBox. Enter False to uncheck the box. The **eGannSquare** draw tool has 2 checkboxes on each *Row*.
Example: `SetLine(Handle, 3, True, 25, clRed, 1, 2, 3, True);`

PercentLevel: Enter a percent level number for the line. Example: 50

Color: The *Color* parameter can be a predefined color constant. Example: `clDkRed`

Style: Enter a number from 0-9 based on the position in the *Style* drop-down list on the Properties window. Example: 7 is a dotted line style.

LeftSide:
RightSide: Enter a number from 0-96 based on the position in the *LeftSide* and *RightSide* markers drop-down lists. Example: 22 will select a 'Star' as the marker

EXAMPLE: The following program opens an IBM daily chart and then draws a line on the chart using the **AddLine** command. The **SetLine** command is then used to change the line Color to Aqua, the line Style to dotted, and to place a 'Date' marker on the left, and a 'Price' marker on the right side of the line. The checkboxes for the 2nd and 3rd rows are then unchecked to make sure that the line is not extended ahead, or extended back.

```
uses Graphics;
begin
  Chart('IBM.D');
  Handle := AddLine(eLine,1,BarEnd-15, Last(BarEnd-15), BarEnd, Last(BarEnd));
  SetLine(Handle,1,True,0,clAqua,7,5,2);
  SetLine(Handle,2,False);
  SetLine(Handle,3,False);
  ChartRefresh(True);
end;
```

SetStudyLine

SYNTAX: **SetStudyLine**(*Study, Row, CheckBox, PercentLevel, Color, Style, Marker, MColor: integer*);

DESCRIPTION: The **SetStudyLine** command is used to change the color, style, and markers for a specified Study. The command will only change the settings for the specified *Row* on the Properties window. Some Studies have as many as 11 *Rows* on their Properties window. The **SetStudyLine** command allows you to change the color, style, and markers after the Study has already been applied to a chart. The **SetStudy** command can also be used to change colors, styles, and markers. The **SetStudyLine** command changes just 1 *Row* at a time.

PARAMETERS:

Study: *Study* is the line Object ID number (Handle) returned by the **FindStudy**, **AddStudy**, and **AddLine** functions. If the value of *Study* is set to Zero, then the **SetStudyLine** command will default to the calling study. Use the **FindStudy** command to find a Study on a chart, and then use the returned *Handle* as the *Study* parameter when using the **SetStudyLine** command.

```
Example: Handle:= FindStudy(eRSI);
         SetStudyLine(Handle, 1, True, 0, clRed,1,22,clWhite);
```

Row: Enter the *Row* of the Properties window that you want to change. Make sure to count blank rows. Some properties windows do not have entry boxes on every *Row*.

CheckBox: Enter a True value to place a check mark in the CheckBox. Enter False to uncheck the box.
Example: SetStudyLine(Handle, 3, True);

PercentLevel: Enter a percent level number for the Study. Example: 50

Mcolor,Color: The *Color* and *MColor* parameters can be a predefined color constants. Example: clRed.

Style: Enter a number from 0-9 based on the position in the *Style* drop-down list on the Properties window.
Example: 7 is a dotted line style.

Marker: Enter a number from 0-96 based on the position in the *LeftSide* and *RightSide* markers drop-down lists.
Example: 22 will select a 'Star' as the marker

EXAMPLE: The following program opens an IBM daily chart and then adds an RSI study to the chart. The **SetStudyLine** command is then used to change the 1st RSI line Color to Aqua, the line Style to dotted, and to place a white 'Star' marker on the on the line. The **ChartRefresh** command is used to recalculate and redraw the study after the line changes have been made.

```
uses Graphics;
begin
  Chart('IBM.D');
  Handle := AddStudy(eRSI);
  SetStudyLine(Handle, 2, True, 0, clAqua, 7, 22, clWhite);
  ChartRefresh(True,GetStudy(Handle,13),Handle);
end;
```

ShellExecute

SYNTAX: **ShellExecute**(*Operation, File, Parameters, Directory*: string; *ShowMessage*: boolean): integer;

DESCRIPTION: Performs an operation on a specified file.

PARAMETERS:

Operation: Specify what you want to do to with the named file. Valid operation options are:

edit - Launches an editor and opens the document for editing. If *File* is not a document file, the function will fail.
 explore - Explores the folder specified by *File*.
 find - Initiates a search starting from the specified directory.
 open - Opens the file specified by the *File* parameter. If the file is not a document file, the function will fail.
 print - Prints the document file specified by *File*. If *File* is not a document file, the function will fail.

File: A string that specifies the file or object on which to execute the specified AOperation.

Parameters: If the *File* Specifies an executable file, *Parameters* specifies the parameters to be passed to the application. The format of this string is determined by the *Operation* to be invoked. If *File* specifies a document file, *Parameters* should be "".

Directory: A string that specifies the default directory.

ShowMessage: Pass True if you want Ensign to show a message in the event of a run-time or exception error.

If the return value is greater than 32, then it is successful, otherwise it failed. The following table lists the possible errors:

ERROR_FILE_NOT_FOUND	The specified file was not found.
ERROR_PATH_NOT_FOUND	The specified path was not found.
ERROR_BAD_FORMAT	The .exe file is invalid (non-Microsoft Win32 .exe or error in .exe image).
SE_ERR_ACCESSDENIED	The operating system denied access to the specified file.
SE_ERR_ASSOCINCOMPLETE	The file name association is incomplete or invalid.
SE_ERR_DDEBUSY	The Dynamic Data Exchange (DDE) transaction could not be completed because other DDE transactions were being processed.
SE_ERR_DDEFAIL	The DDE transaction failed.
SE_ERR_DDETIMEOUT	The DDE transaction could not be completed because the request timed out.
SE_ERR_DLLNOTFOUND	The specified DLL was not found.
SE_ERR_NOASSOC	There is no application associated with the given file name extension. This error will also be returned if you attempt to print a file that is not printable.
SE_ERR_OOM	There was not enough memory to complete the operation.
SE_ERR_SHARE	A sharing violation occurred.

EXAMPLE: The following example uses the ShellCommand function to launch a PDF help file.

```

procedure btnHelpClick(Sender: TObject);           { Button Click Event }
var sFile: string;                                { String Variable - Optional}
var iResult: integer;                              { Integer Variable - Optional}
begin
  sFile := sPath + 'Manuals\ESPL.pdf';             { Set the sFile variable }
  iResult := ShellExecute('open', sFile, '', '', false); {open the file}
end;
```

Show

SYNTAX: **Show**(*Item*: integer [, *Visible*: boolean]);

DESCRIPTION: The **Show** command is used to show or hide an *Item* on a chart. The **Show** command accomplishes the same show or hide effect available on a chart's pop-up menu for many chart elements. This is equivalent to checking or unchecking the Show menu items with the mouse.

PARAMETERS:

Item: The *Item* parameter can be one of the following predefined constants:

eBar:	Show or Hide the Bars on a chart.
eBarData:	Show or Hide the BarData panel on a chart.
eGrid:	Show or Hide the GridLines on a chart.
eInterest:	Show or Hide the Open Interest values on a chart.
eStudyData:	Show or Hide the StudyData panel on a chart.
eVolume:	Show or Hide the Volume bars on a chart.
eVolumeGrid:	Show or Hide the Volume grid lines on a chart.

The numbers **1** through **9** can be entered as the *Item* parameter to control the display of study sub-windows below a chart. If a study sub-window is not shown, then the study will overlay on the chart.

Visible: The *Visible* parameter specifies whether to Show or Hide an *Item*. A *True* value will Show the *Item*. A *False* value will Hide the *Item*. The default is *True*.

EXAMPLE: The following program displays an IBM daily chart, adds a Stochastic study to the chart, hides the Volume bars, and hides the Stochastics study panel (causing the Stochastics study to plot on the chart, instead of in its study panel).

```
begin
  Chart ('IBM.D');
  AddStudy (eSto);
  Show (eVolume, False);           {Hide the volume sub-window}
  Show (1, False);                 {Hides study sub-window 1}
end;
```

ShowMessage ShowMessagePos

SYNTAX: **ShowMessage**(*Message*: string);
ShowMessagePos(*Message*: string, *Left*, *Top*: integer);

DESCRIPTION: The **ShowMessage** and **ShowMessagePos** commands are used to display a MessageBox on the screen. A message can be displayed in the box. Click the OK button in the box to close the box. The **ShowMessage** command will open a MessageBox in the center of the screen. The **ShowMessagePos** can open the MessageBox at a specified location on the screen. The *Top* and *Left* parameters specify the pixel positions for the top-left corner of the MessageBox.

PARAMETERS:

Message: The *Message* parameter is a text string message to display in the MessageBox.
Left: The *Left* parameter specifies the location of the left edge of the MessageBox in pixels (counting from the left edge of the screen).
Top: The *Top* parameter specifies the location of the top edge of the MessageBox in pixels (counting down from the top of the screen).

EXAMPLE: The following program uses the **ShowMessage** command to display a message. The **ShowMessagePos** command is also used to display a message near the left edge of the screen.

```
begin
  ShowMessage('This is a Test Message');
  ShowMessagePos('Test Message', 10, 100);
end;
```

sList

SYNTAX: **sList**: TStringList;

DESCRIPTION: The **sList** string list is created when Ensign runs. Using this global string list variable can simplify your programs because Ensign will automatically create, load, save and free the string list. If desired, the **sList** string list can be automatically loaded with the contents of a file when Ensign runs. Create a file name `SLIST.DAT` and store it in the `\Ensign10` directory. If the file exists when Ensign runs, then the contents of the file will be automatically loaded into **sList**. The contents of **sList** will also save to `SLIST.DAT` when Ensign is closed. See the documentation for 'String Lists' for more details on how to use string lists.

EXAMPLE: The following program uses a **FOR** loop to fill **sList** with random text values. The values are sorted and then printed in the output window.

```
begin
  sList.Clear;
  OutPut(eClear);
  writeln('UnSorted');
  for Count := 0 to 10 do begin
    sList.Add(IntToStr(Random(100)));
    writeln(sList.Strings[Count]);
  end;
  writeln('Sorted');
  sList.Sort;
  for Count := 0 to 10 do writeln(sList.strings[Count]);
end;
```

sLog sStudyLog sLineLog sSoundLog

SYNTAX: **sLog**: TStringList;
 sStudyLog: TStringList;
 sLineLog: TStringList;
 sSoundLog: TStringList;

DESCRIPTION: The **sLog** string list is created when Ensign runs. This string list holds the contents of the Ensign Alerts log. Click the **Alert** button in Ensign to display the Alerts log. The Alerts log shows the symbols that have hit price alerts. Use the **sLog** string list to examine or manipulate the contents of the Alerts log. See the documentation for 'String Lists' for more details on how to use string lists.

The **sStudyLog**, **sLineLog**, and **sSoundLog** maintain logs for studies, lines, and alert sounds.

EXAMPLE: The following program prints the contents of the Alerts log to the output window.

```
begin
  writeln(sLog.Text);
end;
```

sPath

SYNTAX: *sPath*;

DESCRIPTION: The *sPath* variable is automatically assigned the Path for the Ensign program directory. For example, if the Ensign program is installed in the C:\Ensign10 folder, then *sPath* will equal 'C:\Ensign10'. The path can be used whenever accessing files in the program directory.

EXAMPLE: The following program loads a custom quote page file named CUSTOM.DAT into the global **sList** string list. The *sPath* variable is used since custom quote page files are located in the Ensign program folder. The contents of the **sList** are then printed to the output window. NOTE: Custom quote page files contain the symbols that appear on the custom quote page. The first character of each line in the file represents the feed group for the symbol.

```
begin
  sList.LoadFromFile(sPath + 'QuoFile\Custom.dat');
  writeln(sList.Text);
end;
```

Speak

SYNTAX: Speak(*Message*: string);

DESCRIPTION: The Speak command sends the Message sting to the voice output thread which Speaks the phrase.

PARAMETER:

Message: The *Message* parameter specifies the message you want spoken.

EXAMPLE: The following example uses the Voice function speak the passed phrase.

```
begin
  Speak('Mike is cool and awesome');
end;
{ Start of program }
{ Have the computer say this }
{ End of program }
```

Spreadsheet

SYNTAX: **Spreadsheet**(eClear): boolean;
 Spreadsheet(eReset): boolean;
 Spreadsheet(eRecalc): boolean;
 Spreadsheet(ePrint [, *GridOnly*: boolean]): boolean;
 Spreadsheet(eLoad, *FileName*: string): boolean;

DESCRIPTION: The ESPL editor may use the Spreadsheet window for posting results. The **SetCell** command is often used to post text in a spreadsheet cell.

Spreadsheet(eClear) erases the contents of the spreadsheet form without prompting to proceed.

Spreadsheet(eReset) prompts for confirmation before erasing the contents of the spreadsheet form.

Spreadsheet(eRecalc) causes the spreadsheet to recalculate. This is the same as clicking the Recalculate toolbar button.

Spreadsheet(ePrint, *GridOnly*) prints the spreadsheet. The *GridOnly* parameter should be a True or False value. True will print just the Grid sheet. False will print the entire spreadsheet form which includes the caption and toolbar.

Spreadsheet(eLoad, *FileName*) loads the specified spreadsheet. This is the same as selecting the specified spreadsheet using the spreadsheet form's combo box.

EXAMPLE: The following program responds to 5 ESPL button clicks. Click ESPL button 1 to open a spreadsheet. Click ESPL button 2 to load a sheet named 'Daily Report'. Click ESPL button 3 to put text and color in a spreadsheet cell. Click ESPL button 4 to erase the spreadsheet. Click ESPL button 5 to print the spreadsheet form. These ESPL buttons are on the Run ESPL form or on the toolbar on the ESPL editor.

```
begin                                     {Start of Main Programming code}
  case ESPL of
    1: btnSpreadSheet.Click;             {opens a spreadsheet}
    2: Spreadsheet(eLoad, 'Daily Report'); {loads Daily Report sheet}
    3: begin
      FindWindow(eSpread);               {finds a spreadsheet window}
      SetCell(3, 2, 'Hello', clRed);     {column 3, row 2, text, cell color}
    end;
    4: Spreadsheet(eClear);               {erases the spreadsheet}
    5: Spreadsheet(ePrint, false);       {print entire form}
  end;                                    {end of case statement}
end;                                      {End of program}
```

Sqr Sqrt

SYNTAX: **Sqr**(*Number*: real): real;
 Sqrt(*Number*: real): real;

DESCRIPTION: The **Sqr** function is used to Square the specified *Number* (example: *Number* * *Number*). The **Sqrt** function is used to determine the Square Root of the specified *Number* (example: The square root of 16 is 4).

EXAMPLE: The following program prints the Square of 10 random numbers, and prints the Square Root of 10 random numbers.

```
begin
  for Count := 1 to 10 do begin
    Number:= Random(100);
    writeln('The Square of ', Number, ' is ', Sqr(Number));
    Number:= Random(1000);
    writeln('The Square Root of ', Number, ' is ', Sqrt(Number));
  end;
end;
```

Std

StdDev

SYNTAX: **Std**(*Number*: real, var *SumX2* , *SumX*: real, var *Count*: real, *Flag*: boolean): real;
StdDev(*Type*: integer, *Index*: integer, *Count*: integer, *Flag*: boolean [, *Dataset*: integer]): real;

DESCRIPTION: The **Std** and **StdDev** functions are used to calculate the Standard Deviation for a set of data points.

The **Std** function returns the standard deviation with *Number* added to the data points sample. To calculate the standard deviation for 20 *Numbers*, call the **Std** function 20 times and pass a new *Number* each time. Use the result of the 20th call as the answer. NOTE: *Count*, *SumX2*, and *SumX* should be initialized to zero before the first call. Each call to the function will automatically increment the *Count* value.

The **StdDev** function returns the standard deviation for a set of Chart Bar data points. *Type* specifies the data point to use. *Count* specifies how many bars to use, ending at bar *Index*. The **StdDev** function calculates the answer with only one call.

PARAMETERS:

Number: Specifies the new *Number* to add to the data point sample.
SumX2: This variable contains the sum of all the *Numbers* squared.
SumX: This variable contains the sum of all the *Numbers*.
Count: This variable is a *Count* of how many *Numbers* or bars are in the data point sample.
Flag: True = use formula for a specific population. False = use formula for a sample population.

Type: *Type* is used to specify a Chart Bar data point. The following *Types* can be used:

eArray	eClose	eHigh	eLast	eLow	
eMidPoint	eMid3	eMid4	eNet	eOpen	
eOpenInterest	ePercent	eRange	eTrueHigh	eTrueLow	
eTrueRange	eVolume	1	2	3	4

Refer to the **Bar** function for a complete description of these constants.

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart.
DataSet: The default is to use the chart's bar data set.. *DataSet* is an optional object number for an overlay data set..

Standard Deviation Formula

SumX2 := SumX2 + Sqr(x)

SumX := SumX + x

specific population: Result = sqrt((n * sumX2 - sqr(sumX)) / sqr(n))

sample population: Result = sqrt((n * sumX2 - sqr(sumX)) / n * (n - 1))

EXAMPLE: The following program uses the **Std** function to calculate and draw a Standard Deviation line on a chart. Click ESPL button 100 on the Run ESPL panel to apply the study to the active chart.

```
procedure StdExample;
var
  i,j,k: integer;
  dev,sx,sx2,n,scalelow: real;
begin
  k := GetUser(eParm1);
  Scalelow := GetVariable(eScaleLow);
  if BarBeginLeft>k then
    for i:=BarBegin to BarEnd do begin
      sx := 0;
```

```

    sx2 := 0;
    n := 0;
    for j := i-k+1 to i do dev:=Std(Last(j),sx2,sx,n,True);
    SetUser(1,dev+scalelow,i);
end;
end;

{****Main Program****}
begin
  if ESPL=100 then StdExample;
end;

```

EXAMPLE: The following program uses the **StdDev** function to calculate and draw a Standard Deviation line on a chart. Click ESPL button 100 on the Run ESPL panel to apply the study to the active chart.

```

procedure StdDevExample;
var
  i,Period: integer;
  dev,scalelow: real;
begin
  Period := GetUser(eParm1);           {Number of bars parameter}
  scalelow := GetVariable(eScaleLow);  {Offset from bottom of chart}
  if BarBeginLeft>Period then         {be sure you have enough bars}
    for i:= BarBegin to BarEnd do begin
      dev := StdDev(eLast,i,Period,true); {Calculate Standard Deviation}
      SetUser(1,dev+scalelow,i);        {Plot line offset from bottom}
    end;
end;

{****Main Program****}
begin
  if ESPL=100 then StdDevExample;
end;

```

Str

SYNTAX: **Str**(*Number*: real, *Width* , *DecimalPlaces*: integer, var *Text*: string);

DESCRIPTION: The **Str** command is used to convert a number into a string value. The value of *Number* is returned in the *Text* variable as a string. The *Width* of the string, and the *NumberOfDecimalPlaces* to include in the value can be specified. Use a Zero value for *Width* to specify no padding or truncation of the *Number*.

PARAMETERS:

Number: Specifies a *Number* to convert to a string.
Width: Specifies the *Width* of the resulting string.
DecimalPlaces: Specifies how many decimal places to include in the string value.
Text: A string variable that will receive the string output from the **Str** command.

EXAMPLE: The following program converts the number 888.777 to a string value of '888.78'. The program then converts the number 999.9 into a string value of ' 999.9'

```

var
  s: string;
begin

```

```

Str(888.777,7,2,s);
writeln(s);
Str(999.9,10,1,s);
writeln(s);
end;

```

String Lists

DESCRIPTION: A String List is used to store and manipulate a list of strings. A String is simply some ASCII text (like Symbols, words, sentences, etc.). A String List could be used to store and manipulate a list of Symbols, Alert messages, or Report values, etc. A String List must be declared as a *TStringList* variable type. The String List must be created before being used. After using the String List, it should be Freed. This releases the computer memory that the String List occupied. The following program shows how to declare, create, use, and free a String List.

```

var                                     {Start of Variable Declarations}
  MyQuotes: TStringList;                {MyQuotes declared as a TStringList}
begin                                   {Start of Main Programming code}
  MyQuotes := TStringList.Create;       {MyQuotes String List is created}
  MyQuotes.LoadFromFile(sPath + 'QuoFile\Custom.dat'); {Quote page is loaded}
  writeln(MyQuotes.Text);               {The list is printed}
  MyQuotes.Free;                       {The String List is freed}
end;                                    {End of program}

```

In addition to storing strings, a String List has the following functions:

- Add or delete strings at specified positions in the list.
- Access the string at a particular location.
- Sort the strings in the list.
- Rearrange the strings in the list.
- Read the strings from or write the strings to a file
- Create comma delimited strings

STRING LIST METHODS AND PROPERTIES: The following methods and properties can be used with String Lists. Add the method or property to the end of the String List variable (separated with a period) to get the desired effect. Example: `MyQuotes.Clear;` will clear all entries in the example 'MyQuotes' String List.

METHODS

Add: Add inserts a string at the end of the list.
AddStrings: AddStrings adds a group of strings to the list.
Clear: Deletes all the strings in the list.
Create: Construct and initialize a String List before it is first used.
Delete: Deletes a specified string from the list.
Exchange: Exchange swaps the position of two strings in the list.
Find: Finds the Index for a string in a sorted list, indicates whether a string with that value exists.
Free: Destroys a String List and frees its associated memory.
IndexOf: Returns the position of a string in the list.
IndexOfName: Returns the position of the first string with the form Name=Value with the specified name part.
Insert: Inserts a string at a specified position in the list.
LoadFromFile: Loads a String List with the contents of the specified text file.
Move: Moves a string to a different position in the list.
SaveToFile: Saves the String List contents to the specified text file.
Sort: Sorts the strings in the list in ascending order.

PROPERTIES

CommaText: Returns the String List as a single comma-delimited string.
 Count: Reports the number of strings in the list.
 Duplicates: Specifies whether duplicate strings can be added to the list, or not.
 Sorted: Specifies whether the strings in the list should be automatically sorted.
 Strings: Provides access to individual Strings in the list (the list starts at position 0)
 Text: Returns the String List as a single string (a Carriage Return/Line Feed separates each list entry).
 Names: For strings in the list of the form Name=Value, Names returns the name part of the string.
 Values: Returns the Value portion of a string associated with a given Name.

The following documentation describes each Method and Property in greater detail.

METHODS:

Add: Add(*S*: string): Integer;
 Use Add to add a string to the end of the list. Add returns the index of the new string.
 Example: `MyQuotes.Add('IBM');`

AddStrings: AddStrings(*Strings*: TStringlist);
 Use AddStrings to add the strings from another TStringlist.
 Example: `MyQuotes.AddStrings(OtherQuotes);`

Delete: Delete(*Index*: Integer);
 Delete removes a string from the list. *Index* specifies the position of the string, where 0 is the first string, 1 is the second string, and so on. Example: `MyQuotes.Delete(5);` deletes the string at position 5 in the list.

Exchange: Exchange(*Index1*, *Index2*: Integer);
 Use Exchange to swap two strings in the list. The strings are specified by their index values in the *Index1* and *Index2* parameters. Indexes are zero-based, so the first string in the list has an index value of 0, the second has an index value of 1, and so on. Example: `MyQuotes.Exchange(1, 5);` swaps the strings in positions 1 and 5.

Find: Find(*S*: string; var *Index*: Integer): Boolean; virtual;
 Use Find to obtain the Index position in a sorted list where the string *S* should be added. If the string *S* already exists in the list, Find returns True. If the list does not contain *S*, Find returns False. The index where *S* should go is returned in the *Index* parameter. Only use Find with sorted lists. For unsorted lists, use the IndexOf method instead. Example: `MyQuotes.Find('IBM', Index);` reports the position where IBM should be inserted in a sorted list.

IndexOf: IndexOf(*S*: string): Integer;
 Use IndexOf to find the position of the first occurrence of the string *S*. IndexOf returns the 0-based index of the string. If *S* is not found in the String List then -1 is returned. If the string appears in the list more than once, the position of the first occurrence will be returned.
 Example: `MyQuotes.IndexOf('IBM');`

IndexOfName: IndexOfName(*Name*: string): Integer;
 Use IndexOfName to locate the first occurrence of a string with the form Name=Value where the name part is equal to the *Name* parameter. If the indicated name is not found, then -1 is returned.
 Example: `MyQuotes.IndexOfName('IBM');` returns the position of the 'IBM' name.

Insert: Insert(*Index*: Integer; *S*: string);
 Use Insert to add the string *S* at the specified *Index*. If *Index* is 0, the string is inserted at the beginning of the list. Example: `MyQuotes.Insert(0, 'MSFT');` inserts 'MSFT' at the start of the list.

LoadFromFile: `LoadFromFile(FileName: string);`
 Use `LoadFromFile` to load the contents of the specified text file into the list. Each line in the file is appended as a string in the list.
 Example: `MyQuotes.LoadFromFile(sPath + 'Custom.Quo');`

Move: `Move(CurrentIndex, NewIndex: Integer);`
 Use `Move` to move the string at position `CurrentIndex` to the position of `NewIndex`. The following example moves the string in the beginning position to the last position.
 Example: `MyQuotes.Move(0, MyQuotes.Count-1);`

SaveToFile: `SaveToFile(FileName: string);`
 Use `SaveToFile` to save the String List to the specified text file. Each string in the list is written to a separate line in the file. Example: `MyQuotes.SaveToFile(sPath + 'MYFILE.TXT');`

Sort: `Sort;`
 Use `Sort` to sort the strings in a list that has the `Sorted` property set to `False`. String lists with the `Sorted` property set to `True` are automatically sorted in ascending order. Example: `MyQuotes.Sort;`

PROPERTIES:

CommaText: `CommaText: string;`
 Returns the String List as a single comma-delimited string.
 Example: `writeln(MyList.CommaText);`

Count: `Count: integer;`
 Reports the number of strings in the list. NOTE: Since the first item in the list starts at position 0, the last item in the list will be at position `Count-1`.
 Example: `writeln(MyQuotes.Count);`

Duplicates: `Duplicates: constant;`
 Set `Duplicates` to enable or disable duplicate entries in a sorted string list. Set the value of `Duplicates` before adding any strings to the list. The value of `Duplicates` is ignored if the String List is not sorted. The value for `Duplicates` should be one of the following *Constants*:
 `dupIgnore`: Ignore attempts to add duplicate strings to a sorted list.
 `dupError`: An error occurs when an attempt is made to add duplicate strings to a sorted list.
 `dupAccept`: Permit duplicate strings in the sorted list.
 NOTE: Setting the value of `Duplicates` to `dupIgnore` or `dupError` does not correct duplicate strings that are already in the list. Example: `MyQuotes.Duplicates := dupError;`

Sorted: `Sorted: Boolean;`
 Set `Sorted` to `True` to cause the strings in the list to be automatically sorted in ascending order. Set `Sorted` to `False` to allow strings to remain where they are inserted. When `Sorted` is `False`, the strings in the list can be put in ascending order at any time by calling the `Sort` method. When `Sorted` is `True`, do not use `Insert` to add strings to the list. Instead, use `Add`, which will insert the new strings in the appropriate position. When `Sorted` is `False`, use `Insert` to add strings to an arbitrary position in the list, or `Add` to add strings to the end of the list. Example: `MyQuotes.Sorted := True;`

Strings: `Strings[Index: Integer]: string;`
 Use `Strings` to read or modify the string at a particular position. `Index` gives the position of the string, where 0 is the position of the first string, 1 is the position of the second string, and so on. To locate a particular string in the list, call the `IndexOf` method.
 Example: `MyQuotes.Strings[4] := 'This message is stored in position 4';`

Text: `Text: string;`

Use `Text` to get or set all the strings in a single string delimited by carriage return, line feed pairs. When reading `Text`, the strings in the list will be separated by carriage return, line feed pairs. If any of the strings in the list contain a carriage return and line feed pair, the resulting value of `Text` will appear to contain more strings than is indicated by the `Count` property. When setting `Text`, the value will be parsed by separating into substrings whenever a carriage return or linefeed is encountered. (The two do not need to form pairs). Example: `writeln(MyQuotes.Text);`

Names: `Names[Index: Integer]: string;`
 When the list of strings includes strings of the form `Name=Value`, read `Names` to access the name part of a string. `Names` is the name part of the string at the position indicated by `Index`. If the string at the specified position is not of the form `Name=Value`, `Names` returns an empty string. Strings of the form `Name=Value` are commonly found in .INI files. The Name that identifies the string is to the left of the equal sign (=), and the current Value of the Name identifier is on the right side. There should be no spaces present before or after the equal sign. Example: `MyQuotes.Add('IBM=400');` IBM would be the Name, and 400 would be the Value.

Values: `Values[Name: string]: string;`
 When the list of strings includes strings of the form `Name=Value`, use `Values` to get or set the value part of a string associated with a specific name part. If the list does not contain any strings of the proper `Name=Value` form, or if none of those strings matches the Name index, `Values` returns an empty string. Example: `writeln(MyQuotes.Values['IBM']);` will print the Value of IBM (equals 400).

EXAMPLE: The following program demonstrates the use of a String List. A list is declared and created. Several uses of the list are demonstrated. The list is freed before the program ends.

```
var                                     {Start of Variable Declarations}
  j: integer;                           {J is declared as an Integer}
  MyList: TStringList;                  {MyList is declared as a String List}
begin                                    {Start of Main Programming code}
  MyList := TStringList.Create;         {MyList String List is created}
  MyList.Sorted := False;               {Sorted property is set to False}
  MyList.Add('IBM=79');                  {'IBM=79' is added to the list}
  MyList.Insert(0, 'AAPL=42');           {'AAPL=42' is inserted at beginning}
  MyList.Add('MSFT=38');                 {'MSFT=38' is added to end of list}
  writeln(MyList.Strings[0]);            {The first list item is printed}
  ShowMessage(MyList.Text);              {List is printed in a MessageBox}
  writeln(MyList.CommaText);             {Prints a Comma delimited list}
  MyList.Strings[1] := 'DELL=55';        {String 1 is replaced with 'DELL=55'}
  writeln('Number of Items ', MyList.Count); {Prints the items in list}
  writeln('First entry Name is ', MyList.Names[0]); {Prints Name of 1st item}
  writeln('Value for DELL is ', MyList.Values['DELL']); {Prints Value of DELL}
  for j := 0 to MyList.Count-1 do        {Loop and Print each item in list}
    writeln(MyList.Names[j], ' ', MyList.Values[MyList.Names[j]]);
  MyList.Free;                           {Free the MyList String List}
end;                                       {End of program}
```

StringToDate StrToDate

SYNTAX: `StringToDate(Text: string): integer;`
`StrToDate(Text: string): TDateTime;`

DESCRIPTION: The **StringToDate** and **StrToDate** functions are used to convert string values into Integer or TDateTime values. For example, the **StringToDate** command will convert a string date of '12-31-01' into an integer value of 20011231.

For **StrToDate**, the date in the string must be a valid date. The string must consist of two or three numbers, separated by a 'DateSeparator' character. For example, **StrToDate**('12/31/2001') returns a TDateTime value of 12/31/2001. If the string contains only two numbers, it is interpreted as a date (m/d or d/m) in the current year. The order for month, day, and year is determined by the 'ShortDateFormat' windows variable. Possible combinations are m/d/y, d/m/y, and y/m/d. To change the 'DateSeparator' character or 'ShortDateFormat', open the Windows Control Panel and select the 'Regional' settings window and view the 'Date' screen.

PARAMETERS:

Text: Specifies a string value to convert to either an Integer or TDateTime value.

EXAMPLE: The following program converts a string value of '10/31/2001' into a TDateTime value. The program then converts the string value of '12-31-01' into an integer value of 20011231.

```
begin
  writeln(StrToDate('10/31/2001'));           {Prints 10/31/2001}
  writeln(StringToDate('12-31-01'));         {Prints 20011231}
end;
```

System

SYNTAX: **System**(*Request*:constant): string;

DESCRIPTION: The **System** function is used to retrieve various system values. These items can be used to help identify a user or a computer. They can also be incorporated into security measures to help prevent users from using a Script without your permission. For example, a script could be written that will only run on a specific computer. If necessary, make use of the .LIB file extension to encrypt the programming code.

PARAMETERS:

Enter one of the following constants to *Request* a specific item.

eAccount: Reports the Ensign ID number found on the Security screen. This is unique on all computers.

eCustom: Reports the Computer ID number found on the Security screen. This is unique on all computers.

eProgramID: Reports the Program ID number found on the Security screen as the NID. This is unique for all installations and is intended to replace the Ensign ID and the Computer ID. Use this NID to uniquely identify a user.

eESPL: Reports the Log In UserName used by the eSignal Data Manager or DTN IQfeed data-feed software. This command only works if you are using either of these data-feeds.

eTimeZone: Returns an integer value that represents the Time Zone setting on the Setup Computer screen: 0=Eastern, 1=Central, 2=Mountain, 3=Pacific, up to 23, etc.

eVolume: Reports the Volume ID of the hard disk. This can be used to specifically identify a computer for security.

Example: Click ESPL button 1 to retrieve and print the indicated system information.

```

begin
  if ESPL=1 then begin
    writeln(System(eAccount));
    writeln(System(eCustom));
    writeln(System(eProgramID));
    writeln(System(eESPL));
    writeln(System(eTimeZone));
    writeln(System(eVolume));
  end;
end;

```

Template

SYNTAX: **Template**(*Action*: constant, *Name*: string, [*Tab*: integer]): boolean;

DESCRIPTION: The **Template** command is used to apply a Study Template to a chart. The *Action* parameter can be eLoad or eSave. The *Name* parameter specifies which Template to load or save.

PARAMETERS:

Action: Enter either eLoad or eSave.

Name: Enter the name of the Template that you want loaded or saved.

Tab: Enter the template tab (folder) for the Template. The default is to use the currently selected tab.

EXAMPLE: The following program opens an IBM daily chart, applies two studies to the chart, and then saves those studies as a Template named 'MyStudies'.

```

begin
  Chart('IBM.D');
  AddStudy(eRSI); AddStudy(eSto);
  Template(eSave,'MyStudies',2); {Saves the studies as a Template in folder 2}
end;

```

TCP Connections

SYNTAX: **tcpConnect**(*Port*, *Remotehost* : string);
tcpConnected : boolean;
tcpDisconnect;
tcpSend(*Text* : string);
tcpReceive;

DESCRIPTION:

The **TCP** commands allow Ensign to communicate with other application via a TCP connection. Use the **tcpConnected** command to see if a connection already exists. Use the **tcpConnect** command to create a connection on the specified *Port* number, and to the specified *Remotehost* IP address. Use the **tcpSend** command to send text data to the remotehost. Use the **tcpReceive** command to access data that is sent back from the remotehost. The data is retrieved from a receive buffer and the buffer is cleared. If the Remotehost sends continuous data to the TCP connection, then ESPL code would need to be written to retrieve the data from the receive buffer in a timely manner for processing. A **TIMER** is suggested.

EXAMPLE: The following sample program makes a TCP connection, then sends and receives data on the connection. Then the connection is closed.

```

begin
  if tcpConnected = False then begin

```

```

tcpConnect('1000', '127.0.0.1');
Pause(2);                               {wait 2 seconds for connection}
tcpSend('Hello');
Pause(1);                                 {wait for a response}
writeln(tcpReceive);                       {print the response}
tcpDisconnect;
end;
end;

```

TextAdd

TextBox

TextCaption

TextClear

SYNTAX: **TextAdd**(*Text*: string [, *Centered*: boolean]): boolean;
 TextBox([*Caption*: string, *Left*, *Top*, *Width*, *Height*: integer]): integer;
 TextCaption(*Caption*: string): boolean;
 TextClear: boolean;

DESCRIPTION: The **TextBox** command is used to display a message window. The caption, size, and position of the window can be specified. Multi-line messages can be displayed in the TextBox by using the **TextAdd** command. TextBoxes can be used to output custom reports, Symbol lists, Alert Messages, or any text message that you wish to view. The **TextBox** function returns a value of 1 if the window was displayed properly, otherwise a 0 value is returned.

TextCaption changes the caption of the last opened TextBox window. The function returns True if successful, and False otherwise.

TextAdd is used to add a line of text to the last opened TextBox window. The text will be centered in the window, unless the optional *Centered* parameter is False. The function will return True if successful, and False otherwise.

TextClear erases the text in the last opened TextBox window. The function will return True if successful, and False otherwise.

NOTE: The font, font color, font size and font style of the TextBox can be changed by using a *TFont* variable. The background color of the TextBox can be changed by using a *TForm* variable.

PARAMETERS:

Text: Specifies the text string to add to the TextBox.

Centered: Specifies the center justification. Set to True to Center the text. Set to False to Left justify the text.

Caption: Specifies the caption of the TextBox window.

Left,Top: The *Left* and *Top* parameters specify the top-left corner of the TextBox.
 Top indicates how many pixels down from the top of the screen.
 Left indicates how many pixels from the left edge of the screen.

Width,Height: The *Width* and *Height* parameters specify the width and height of the TextBox in pixels.

EXAMPLE: The following program opens a `TextBox`. The caption is set to 'Alert'. The background color of the `TextBox` is changed to Aqua. The font color for the `TextBox` is changed to Black. Two lines of text are added in the window. After 8 seconds the text is cleared.

```
var
  Form: TForm;
  Font: TFont;
begin
  TextBox('Alert',1,1,300,200);           {Open TextBox}
  Form := Activechild;                   {Point to active window}
  Form.color := clAqua;                   {Change background color}
  Font := Form.Font;                       {Load font settings}
  Font.color := clBlack;                   {Change font color}
  Font.size := 18;
  TextAdd('First line of Text',False);    {Add text to the box, not centered}
  TextAdd('Second line of Text');
  Pause(8);
  TextClear;                               {Clear all the text in the box}
end;
```

TextOut

SYNTAX: **TextOut**(*X*, *Y*: integer, *Text*: string);

DESCRIPTION: The **TextOut** command is used to print text in a chart window. The location of the *Text* is specified by the *X,Y* parameters. The font and font color of the chart scale is used. NOTE: The printed text is not saved nor remembered by the chart. Any movement or redraw of the chart will erase the text. Use the **AddNote** command to add permanent notes to a chart.

PARAMETERS:

Text: Specifies the *Text* to print on the chart.
X: Specifies the horizontal position (in pixels), starting from the left edge of the chart window.
Y: Specifies the vertical position (in pixels), counting down from the top of the chart window.

EXAMPLE: The following program uses the **TextOut** command to print the Bid and Ask prices in the top left corner of the chart on a continual basis. The program contains a subroutine procedure named 'PrintBidAsk'. The program is applied to a chart by clicking ESPL button 100 on the Run ESPL panel.

```
procedure PrintBidAsk;                       {Declares PrintBidAsk as a procedure}
var                                           {Start of Variable declarations}
  Bid,Ask:string;                             {Bid and Ask are declared as Strings}
begin                                         {Start of procedure code}
  SetUser(eName,'BidAsk');                     {Names the Study}
  SetUser(eClose,False);                       {Calculates on every tick}
  Find(eChart);                                 {Finds the charts quote data}
  SetBrush(GetVariable(eColorChart),eSolid);   {Erase the Background}
  Bid := FormatPrice(GetData(eBid));            {Get the current Bid price}
  Ask := FormatPrice(GetData(eAsk));           {Get the current Ask price}
  TextOut(0,5,' Bid= ' + Bid);                 {Print the Bid on the chart}
  TextOut(0,25,' Ask= ' + Ask);                {Print the Ask on the chart}
end;                                           {End of the procedure}

{*****Main Program*****}
begin                                         {Start of Main Programming code}
```

```

    if ESPL=100 then PrintBidAsk;           {Call PrintBidAsk if ESPL=100}
end;                                       {End of program}

```

TextWidth

SYNTAX: **TextWidth**(*Text*: string): integer;

DESCRIPTION: The **TextWidth** function is used to determine the width of a string (in pixels) for a chart. The chart's Font and FontSize are used to determine the number of pixels. This command can be used on a chart to align *Text* strings that have varying widths.

PARAMETERS:

Text: The *Text* is measured to determine the width of the text in pixels. Pass the *Text* as the parameter. The result will equal how wide the text is on the chart in pixels.

EXAMPLE: The following program measures the pixel width of three text strings. The **TextWidth** command allows the text to be right-justified on the chart since the width of the text can be determined in pixels. The starting horizontal print position is adjusted so that the right edges of the text strings are all aligned.

```

var                                           {Start of Variable Declarations}
  Pixels1, Pixels2, Pixels3: integer;       {Variables are declared as Integers}
begin                                        {Start of Main Programming code}
  Chart('EUR/USD.D');                       {Open a daily chart}
  Pixels1 := TextWidth('Ensign Software');  {Measure pixel width of the text}
  Pixels2 := TextWidth('Programming');     {Measure pixel width of the text}
  Pixels3 := TextWidth('Power!');          {Measure pixel width of the text}
  TextOut(400,10,'Ensign Software');       {Print text on the chart}
  TextOut(400+Pixels1-Pixels2,30,'Programming'); {Print right edges align}
  TextOut(400+Pixels1-Pixels3,50,'Power!'); {Print right edges align}
end;                                         {End of program}

```

TFont

DESCRIPTION: *TFont* is the variable type for all fonts. A *Font* represents the style, size, and color of text that is used in the program. For example, the Font used for this sentence is 'Times New Roman'. The FontSize is '10'. The FontColor is 'Black'. The FontStyle is 'Regular'. Use *TFont* to change Font properties.

PROPERTIES:

Color: Specifies the *Color* of the text. Example: clBlack, clRed, clWhite, clYellow, etc.
Name: Specifies the *Name* of the font (typeface). Example: 'Courier New', 'Times New Roman', etc.
Size: Specifies the pixel *Size* of the font (the height of the characters).
Style: Specifies the *Style* of the font. Example: 0=Regular, 1 = fsBold, 2 = fsItalic, 4 = fsUnderline

EXAMPLE: The following program changes several *Font* properties in the ESPL Script Editor Window. The program pauses for 5 seconds and then changes the *Font* properties again.

```

var                                           {Start of Variable Declarations}
  Font: TFont;                               {Font declared as a TFont}
  Form: TForm;                              {Form declared as a TForm}
begin                                        {Start of Main Programming code}
  btnOutputWindow.click;                    {Open the output window}

```

```

Form := Activechild;           {Script Window points to Form}
Font := Form.Font;           {Font is assigned Font properties}
Font.Name := 'Times New Roman'; {Font Name is changed}
Font.Color:= clBlue;         {Font Color is changed to Blue}
Font.Size := 18;             {Font Size is changed to 18 pixels}
Font.Style:= fsItalic;       {Font Style is changed to Italics}
writeln('This is a Font Test'); {Print some text}
Pause(5);                    {Pause 5 seconds}
Font.Name := 'Courier New';   {Font is changed to Courier New}
Font.Color:= clBlack;        {Font Color is changed to Black}
Font.Size := 10;             {Font Size is changed to 10 pixels}
Font.Style:= 0;              {Font Style is changed to Regular}
end;                          {End of program}

```

TForm

DESCRIPTION: *TForm* is the variable type that controls window properties. Declare a variable of type *TForm* and then change the window properties. Use the *Self* variable to change properties in the Main Ensign form. The following Properties and Methods are available for all forms.

PROPERTIES:

Active: Active is True when the form has focus.
Caption: Caption is the window's title text.
ClientHeight: ClientHeight is the height (in pixels) of the form's client area.
ClientWidth: ClientWidth is the width (in pixels) of the form's client area.
Color: Color is the background color of the form.
Cursor: Specifies the mouse pointer image when cursor is on the form.
Enabled: If Enabled is False, the form ignores mouse and keyboard events.
Font: Font controls the attributes of text written on or in the form.
Height: Height is the vertical size of the form in pixels.
Hint: The text string that can appear when the mouse is on the form.
Left: The horizontal coordinate of the left edge of a form.
Name: The Name of the form as referenced in the application's code.
ShowHint: When True, the form will display a Help Hint.
Tag: Tag stores an integer value. Can be used for miscellaneous variable storage.
Top: The vertical coordinate of the top edge of a form.
Width: Width is the horizontal size of the form in pixels.
WindowState: Specifies the current State of the window (either Minimized, Maximized, or Normal).
WindowState can be changed to one of the following: *wsMinimized wsMaximized wsNormal*

METHODS:

Close: Close closes a form.
Hide: Hide sets the form's Visible property to False.
Print: Print prints the form.
Show: Sets the form's Visible property to True, brings the form to front of other forms on the screen.
SetBounds: Changes the Size and Position of the window using the following format:
`SetBounds(Left, Top, Width, Height: Integer);`
Use SetBounds to change all of the form's boundary properties at one time. The same effect can be achieved by setting the Left, Top, Width, and Height properties separately, but SetBounds changes all four properties at once.

EXAMPLE: The following program illustrates several of the *TForm* properties. A News window and Quote page are opened. The names of the form windows are printed. The size of a window is changed. The windows are then closed. Several **Pause** commands are included in the program so that you can see the effect of the *TForm* commands.


```

begin
  btnOutputWindow.Click;           {Open the output Window}
  Output(eClear);
  btnNews.click;                   {Open a News window}
  btnQuote.click;                  {Open a Quote Page window}
  for j := 0 to ChildCount-1 do begin {Loop through the open windows}
    t := Child(j);
    writeln(t.Name);               {Print the name of the window}
  end;
  t := Screen.ActiveForm;          {Find the active window}
  t.SetBounds(5,15,500,400);       {Resize the window}
  Pause(3);                        {Pause 3 seconds to see the change}
  t.Close;                          {Close the window}
end;

```

Time

SYNTAX: **Time**: TDateTime;

DESCRIPTION: The **Time** function returns the current computer time.

EXAMPLE: The following program prints the current time.

```

begin
  writeln(Time); {Example: 11:22:41 AM}
end;

```

Timer

SYNTAX: **Timer**(eStart or eStop [, *Interval*, *Timer*, *ESPL*: integer]);

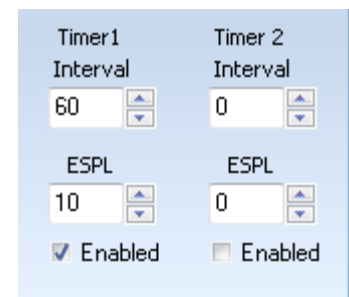
DESCRIPTION: The **Timer** command is used to Start or Stop a Timer clock. The ESPL program can respond to the Timer at the end of each time *Interval*. The *Interval* parameter specifies the number of seconds for each time interval. A default of 60 seconds is used if the *Interval* parameter is not specified. The ESPL program will be called at the end of each time interval with the *ESPL* value set to the *ESPL* parameter.

Timer(eStart, *Interval*) - Starts the Timer
 Timer(eStop) - Stops the Timer

NOTE: The Timer clock can be started manually by checking the Enabled check box on the Run ESPL form. A Timer can be stopped manually by unchecking the Enabled check box on the Run ESPL form.

PARAMETERS:

Interval: Specifies the number of seconds between each call to the ESPL program. This parameter sets the interval shown on the Run ESPL form.
Timer: Select timer 1 or timer 2. If the *timer* parameter is not specified, then timer 1 is used.
ESPL: Set the ESPL variable value. The parameter sets the ESPL selection shown on the Run ESPL form.



EXAMPLE: Click ESPL button 0 (RUN) to start a Timer that will call the ESPL program once per minute. Click ESPL button 1 to stop the Timer. When the Timer is running, a user-defined symbol named 'BASKET' is updated in the eSignal

group. The 'BASKET' symbol is the average of three stocks. The symbol will only update during market hours. The *ESPL* value is set to 10 by the Timer when the program is called. This allows the program to determine that the Timer called the program.

```
begin
  if ESPL=0 then Timer(eStart,60,1,10);      {if ESPL=0 then Start Timer}
  if ESPL=1 then Timer(eStop);              {if ESPL=1 then Stop Timer}
  if ESPL=10 then begin                     {if ESPL=10 then update Symbol}
    if (TimeStr > '08:30') and (TimeStr < '15:00') then begin
      Feed := eSignal;
      Price := (Get('MSFT') + Get('INTC') + Get('DELL')) / 3 ;
      Update('BASKET',Price,1,2);
    end;
  end;
end;
```

EXAMPLE: This example simulates a playback by stepping the chart leftward at the timer interval by sending a period character to the chart.

```
begin
  if ESPL=0 then Timer(eStart,5,1,10);      {if ESPL=0 then Start Timer}
  if ESPL=1 then Timer(eStop);              {if ESPL=1 then Stop Timer}
  if (ESPL=10) and (not Drawing) and (ActiveChart=ActiveChild) then begin
    SetMyFocus;                             {Active form gets the keyboard}
    SendKeys('.');                           {this steps chart bars leftward}
  end;
end;
```

TimeStr

SYNTAX: **TimeStr**: string;

DESCRIPTION: **TimeStr** returns the current time as a string in the format 'hh:mm'.

EXAMPLE: See the programming example for the **Timer** command. The **TimeStr** command is used to limit the updating of the user-defined symbol to market hours.

Top100

SYNTAX: **Top100**(*SearchType*: integer [, *HighPrice*, *LowPrice*: real, *VHigh*, *VLow*: integer]): boolean;

DESCRIPTION: The **Top100** command causes a Top100 scan to run on the currently selected feed group. The scan can be filtered by price and volume. Omit the price and volume parameters to include all symbols in the scan. Enter a zero value in the parameters to ignore the minimum and maximum tests. NOTE: The **Top100** scan can only be run on a feed group (example: eSignal). It cannot be run on a custom quote page. Make sure that the quote page is displaying a feed group before running the scan.

PARAMETERS:

SearchType: The *SearchType* specifies one of the following Top100 scan choices:

eAlphabet	eBetaHigh	eBetaLow	eDailyHigh
eDailyLow	eDividend	eDividendPercent	eDownNet
eDownNetOpen	eDownPercent	eDownPercentOpen	eEPS

eEPSPercent	ePEHighPELow	eTickVolume	eUpNet
eUpNetOpen	eUpPercent	eUpPercentOpen	eVolume
eYearlyHigh	eYearlyLow		

HighPrice: Symbol prices must be lower than this price to be included in the Top100 scan.

LowPrice: Symbol prices must be higher than this price to be included in the Top100 scan.

VHigh: Specifies that symbol volumes must be lower than this volume amount to be included in the scan.

VLow: Specifies that symbol volumes must be higher than this volume amount to be included in the scan.

EXAMPLE: The following program opens an eSignal quote page and runs a Top100 scan looking for symbols that have the greatest percentage gain for the day. Only symbols between 10 and 80 dollars are included in the scan. Only symbols that have at least 50,000 volume are included in the scan. The top 10 symbols are printed in the output window.

```
begin                                {Start of Main Programming code}
  btnOutputWindow.Click;            {Opens the output window}
  Output (eClear);                  {Clear the output window}
  Quote (eSignal);                  {Open the eSignal feed group}
  Top100 (eUpPercent, 80, 10, 0, 50000); {Run Top100 scan for UpPercent stocks}
  for Row:= 1 to 10 do begin        {Loop through first 10 scan symbols}
    Symbol:= GetCell (1, Row);      {Get the symbol in column 1}
    writeln (Row, ' ', Symbol);     {Print the symbol}
  end;                               {end of for loop block}
// mnuCloseWindow.Click;          {remove comment to Close the Quote page}
end;                                {End of program}
```

Trade

SYNTAX: **Trade**(*Signal*: integer [, *Index*: integer, *Price*: real, *Quantity*: integer]): boolean;

DESCRIPTION: The **Trade** command is used to back-test trading systems on a chart. *Buy*, *Sell*, *Reverse*, and *Out* trade signals can be generated. As trades are generated, the chart will be marked with bullets at the trade prices. The trade prices can also be marked with arrows. Click the **Results** button (in the ESPL Script Editor window) when the trading system is finished to view a detailed listing of the trades (you can also select **Chart | Trade Detail** from the menu to view the list).

The **Trade** function returns a `True` value when an open trade is closed out or reversed.

The **Trade** function returns a `False` value when a new position is established (no current open positions).

PARAMETERS:

Signal: The *Signal* parameter is used to specify the trade type.

A *Signal Modifier* may also be added to the *Signal*.

Use one of the following *Signal* constants.

eBuy: A Buy trade will be generated, regardless of the current position.

eSell: A Sell trade will be generated, regardless of the current position.

eOut: A trade position will be closed out.

Signal Modifiers: The following modifiers can be added to the *Signal* parameter to create specific trades:

eIf: A Buy trade will be generated if the trade position is not already Long.

A Sell trade will be generated if the trade position is not already Short.

eReverse: A Buy trade position will be closed out, and then a Sell trade (short) generated.

A Sell trade position (short) will be closed out, and a Buy trade will be generated.

eStop: If the bar's Open price is above the Buy Price, then the Open price will be used.
If the bar's Open price is below the Sell Price, then the Open price will be used.

Example:

eBuy+eIf+eReverse If the position is Out, establish a long position.
If the position is Short, close out, and establish a long position.
If the position is Long, don't do anything.

eBuy+eIf+eReverse+eStop Same as 1st example, except check the Open price. Use Open price if the market gaps above (Buy) or below (Sell) the trade price.

Index: *Index* is the array index for the bars. If *Index* is zero, or omitted, then *Ensign* will use the index of the last bar as the default. The *EntryDate* for the trade will be set to the date of the bar at the index position. The bullets and arrows that mark the chart will be positioned on the indexed bar.

Price: *Price* is the price to execute the trade at. *Price* does not have to be in the range of the bar. If *Price* is zero, or omitted, then *Ensign* will use the Close price of the indexed bar as the default.

Quantity: *Quantity* specifies the number of contracts or shares to trade. If the trade *Signal* indicates a Reverse, then the current position will be closed out, and the new trade position will have a size of *Quantity*. If *Quantity* is omitted, a default of 1 is used.

EXAMPLE: The following program opens an IBM daily chart and runs a Trading System based on the crossing of two Stochastics lines. The system will buy when the Stochastics lines cross up. The system will sell when the Stochastics lines cross down. The **ResetTrades** command is used to initialize the trading system with no commissions and specific trade arrows. A **FOR** loop and the **GetStudy** command are used to loop through the bars and determine when the lines cross. The **Trade** command is used to create each trade. A **TradeReport** is printed in the output window when the system is finished. Arrows will mark each trade on the chart.

```
var                                {Start of Variable declarations}
  i, Handle:integer;              {Variables are declared as Integers}
begin                               {Start of Main Programming code}
  Chart('IBM.D');                 {Open an IBM daily chart}
  Handle := AddStudy(eSto);         {Add the Stochastics study to the chart}
  ResetTrades(0,0,1,3);           {Initialize Trading System parameters}
  for i:= BarBeginLeft to BarEnd do {Loop through looking for trades}
  begin                             {start of loop code}
    if GetStudy(Handle,5,i)=1 then Trade(eBuy+eIf+eReverse,i); {buys}
    if GetStudy(Handle,5,i)=2 then Trade(eSell+eIf+eReverse,i); {sells}
  end;                             {end of loop code}
  Trade(eOut);                     {Close-out the last open trade}
  TradeReport(True);               {Print the results in the output window}
end;                               {End of program}
```

TradeReport

SYNTAX: **TradeReport**(*Enabled*: boolean): real;

DESCRIPTION: The **TradeReport** command will print a trading system summary report in the output window. The function will return the value of the Total Profit. Set the value of *Enabled* to *True* to generate the printed TradeReport.

Set the value of *Enabled* to False to disable the TradeReport (but the Total Profit will still be returned by the function).
Sample Report:

SP1Z	Profit	Trades	Average	Ratio
Wins	95500.00	7	13642.86	46.67
Loss	28650.00	8	3581.25	53.33
Total	66850.00	15	4456.67	3.33

The Profit column displays the Win, Loss, and Total totals. The Trades columns display how many winning and losing trades were generated. The Average column divides the Profit column with the Trades column. The Ratio column divides the number of winning or losing trades with the Total trades. See the **Trade** function to view a sample program that includes the **TradeReport** command.

Trim

TrimLeft

TrimRight

SYNTAX: **Trim**(Text: string): string;
 TrimLeft(Text: string): string;
 TrimRight(Text: string): string;

DESCRIPTION: The **Trim** function trims leading and trailing spaces and control characters from the *Text* string. The **TrimLeft** function trims leading spaces and control characters from the *Text* string. The **TrimRight** function trims trailing spaces and control characters from the *Text* string.

EXAMPLE: The following program trims spaces from some sample text.

```
var
  Text: string;
begin
  Text:= '   This string has leading spaces.' ;
  writeln(TrimLeft(Text));
  Text:= 'This string has trailing spaces.   ' ;
  writeln(TrimRight(Text));
  Text:= '   This string has spaces on both ends.   ' ;
  writeln(Trim(Text));
end;
```

UDP Connections

SYNTAX: **UdpConnect**(Port, Remotehost : string);
 UdpConnected : boolean;
 UdpDisconnect;
 UdpSend(Text : string);
 UdpReceive;

DESCRIPTION: The UDP commands allow Ensign to communicate with other application via a UDP connection. Use the **UdpConnected** command to see if a connection already exists. Use the **UdpConnect** command to create a connection on the specified *Port* number, and to the specified *Remotehost* IP address. Use the **UdpSend** command to send text data to the remotehost. Use the **UdpReceive** command to access data that is sent back from the remotehost. The

data is retrieved from a receive buffer and the buffer is cleared. If the Remotehost sends continuous data to the UDP connection, then ESPL code would need to be written to retrieve the data from the receive buffer in a timely manner for processing. A TIMER is suggested.

EXAMPLE: The following sample program makes a UDP connection, then sends and receives data on the connection. Then the connection is closed.

```
begin
  if UdpConnected = False then begin
    UdpConnect('1000', '127.0.0.1');
    Pause(2);                {wait 2 seconds for connection}
    UdpSend('Hello');
    Pause(1);                {wait for a response}
    writeln(UdpReceive);    {print the response}
    UdpDisconnect;
  end;
end;
```

Put Update

SYNTAX: **Put**(*Symbol*: string , *Price*: real [,*Volume*, *Scale*, *Feed*: integer, *TimeStamp*: TDateTime]): boolean;
 Update(*Symbol*: string , *Price*: real [,*Volume*, *Scale*, *Feed*: integer, *TimeStamp*: TDateTime]): boolean;

DESCRIPTION: **Put** and **Update** are the same function. The function is used to create and update a User-Defined symbol. This function can be used to manually update and simulate price ticks for a Symbol. If the symbol does not already exist, then the first tick will insert the specified symbol into a feed group and initialize the record. If the symbol already exists, then the existing record will be updated with a tick from the supplied Price. The quote record is also retrieved and decoded, ready for use by the GetData function.

PARAMETERS:

Symbol: Specifies the *Symbol* to update. Example: 'AABB'

Price: The *Price* will update the Last price for the symbol, and post a tick to the tick pool used by intraday charting. This allows the User-Defined symbol to be charted in real-time. Charts, quote tables, and alerts will be updated, as if the symbol and tick had been provided by a data feed. The Symbol's Open, High, and Low prices will also be updated.

Volume: Specifies the tick *Volume*. If no *Volume* is specified then the default will be 1.

Scale: *Scale* is the scale factor for decimal placement. Use 4 for 10,000ths, 2 for 100ths, -1 for grains in 8ths, and -3 for bonds in 32nds. The default is 2.

Feed: Specifies the feed group to save the quote to. This parameter can be eFXCM, eIB, eSignal, eIQFeed, eNinja, eOpenECry, eTraderBytes, eTransAct, eGlobal, eDBFX, or eATCBrokers. The default is the value assigned to the FEED global variable.

TimeStamp: If a TimeStamp is included, then the tick pool can be stocked with imported or simulated historical ticks for whatever Date and Time that you want. The TimeStamp should always be Eastern Time Zone. The tick will be timestamped with the TdateTime that you specify.

EXAMPLE: The following sample program adds 2 stock symbols together and creates a new symbol named DEAP9.

```

begin
  Feed := eSignal;           {Assign a vendor feed as the default}
  Price:= Get('DELL') + Get('AAPL');
  Put('DEAP9', Price, 1, 2);
end;

```

EXAMPLE: The following program inserts a specific Price into the tick pool with a June, 30th, 2010, at 9:45 AM (Eastern time) Timestamp for the symbol named ABCD. The symbol and price tick will be saved in the eSignal feed group

```

begin
  Price := 123.55;
  TimeStamp := EncodeDate(2010, 06, 30) + EncodeTime(9, 45, 0, 0);
  Update('ABCD', Price, 1, 2, eSignal, TimeStamp);
end;

```

Val

SYNTAX: **Val**(*TextValue*: string, var *Number*: real, var *Index*: integer);

DESCRIPTION: The **Val** statement is used to convert a numeric Text string into a Real number. The numeric value represented by *TextValue* is returned in the *Number* variable. If *Index* returns as 0 then the conversion was successful, otherwise *Index* will point to the character in *TextValue* where the conversion failed.

EXAMPLE: The following program converts a Text string number into a Real number.

```

var                                     {Start of Variable Declarations}
  Text: string;
  Number: real;
  xIndex: integer;
begin                                   {Start of Main Programming code}
  Text:= '99.567';                       {Assign a string number to Text}
  writeln('The value of TEXT is ', Text); {Print the text}
  Val(Text, Number, xIndex);             {Convert the value to a Real number}
  writeln(Number);                       {Print the number}
end;                                     {End of program}

```

Var

SYNTAX: **Var** *VariableName* [, *VariableName* [,...]]: *VariableType*; [*VarName* ...]

The **Var** statement is used to define and declare variables for use in a program. Declaring variables before they are used is optional. If the **Var** statement is part of a procedure or function, then the declared variables are Local to the procedure or function. A Local variable cannot be accessed from other procedures or functions in the program. If the **Var** statement is ahead of all functions and procedures, then the declared variables are Global in scope. A global variable can be accessed from any procedure or function in the program.

PARAMETERS:

VariableName: Specifies the name of the declared variable.

VariableType: Specifies the variable type. One of the following types can be specified.

Boolean	Integer	Real	String	Variant
TDateTime	TstringList	Tform	TScreen	TArray

NOTE: Numeric variables are initialized to zero. String variables are not initialized. Boolean variables are not initialized. Multiple *VariableNames* can be declared on the same line. *VariableNames* cannot be reserved words used for statements, procedures, functions and predefined constants.

EXAMPLE: The following program declares several Integer and String variables for use in a simple math calculation.

```
var                                {Start of Variable Declarations}
  a, x, y, z : integer;           {Variables declared as Integers}
  s1, s2, s3 : string;          {Variables declared as Strings}
begin                              {Start of Main Programming code}
  x := 5; y := 6; z := 24;
  a := x + y + z;
  s1 := IntToStr(x);             {Converts Integer to a String}
  s2 := IntToStr(y);
  s3 := IntToStr(z);
  writeln('x =', x, ' y=', y, ' z=', z);
  writeln(s1 + ' ' + s2 + ' ' + s3);
end;                               {End of program}
```

VarToStr

SYNTAX: **VarToStr**(*Value*: variant): string;

DESCRIPTION: The **VarToStr** function returns the String representation of any variable type passed in the *Value* parameter. This function can be used to obtain the string equivalent of any variable.

EXAMPLE: The following program converts an Integer value into a string.

```
var
  j: integer;
  s: string;
begin
  j := 15;
  s := 'The string value of j is ' + VarToStr(j);
  writeln(s);
end;
```

vArray

DESCRIPTION: **vArray** is a predefined global variable that is an array of variant. The array can be used to store values needed in the program. Using **vArray** has four advantages over **TArray** variables (see **Array** documentation):

1. Ensign automatically creates and frees the array.
2. **vArray** can be redimensioned for any size.
3. **vArray** is variant, and can hold any mix of variable types.
4. **vArray** will be persistent while Ensign is running (the values are always remembered).

Use the **DimArray** and **SetArray** commands to dimension and fill the **vArray**. **vArray** is a single dimension variant array with a lower index boundary of zero. **DimArray** redimensions **vArray** and sets the upper index boundary. **vArray** must be dimensioned before it is used. New elements added by redimensioning will be initialized to a value of zero.

SetArray is used to store a value in vArray. When more than one value is provided, they fill the array sequentially. vArray can be used in an expression by referencing vArray with an *Index* in parenthesis (example: vArray(20) will reference the 20th item in the array). Index values outside the range of 0 to the upper index boundary will cause an error.

The following Math functions can be used on the vArray values: **Average**, **Summation**, **ExpAverage**, **Highest**, **Lowest**, **Regression**, and **StdDev**. See the documentation for these function for more details. NOTE: Pass the constant eArray as the 1st parameter in the functions.

EXAMPLE: The following program uses the **DimArray** function to dimension vArray for 6 items. The **SetArray** command is used to fill the vArray with various values. A loop is then used to print the contents of the array.

```
begin
  DimArray(6);
  SetArray(1, 'Hello', True, 57.89, False, 92, 0);
  for Count := 1 to 6 do writeln(vArray(Count));
end;
```

VarType

SYNTAX: **VarType**(*Variable*: Variant): integer;

DESCRIPTION: The **VarType** function is used to determine the variable Type for a given *Variable*. The result will report one of the following values.

Integer	= 3	- Numeric types: byte, integer.
Real	= 5	- Floating-point type.
Date	= 7	- Date and time (type TDateTime).
Boolean	= 11	- 16-bit boolean
String	= 258	- Dynamically allocated string.

EXAMPLE: The following program determines the variable Type for some supplied values.

```
begin
  writeln(VarType('Hello'));      {Prints 258 since 'Hello' is a String}
  writeln(VarType(555));          {Prints 3 since 555 is an Integer}
  writeln(VarType(True));        {Prints 11 since True is a Boolean value}
end;
```

Volatility

SYNTAX: **Volatility**(*Index*, *Period*, [*Annual*] : integer): real;

DESCRIPTION: The **Volatility** function is used to calculate the Historical Volatility of *Period* number of data points which end at *Index*. The data points are Close prices for the bars on the referenced chart.

PARAMETERS:

Index: *Index* is the bar array subscript between 1 and the number of bars on the chart.

Period: *Period* is the number of data points to use. The data points are obtained from the chart bars with indexes from (*Index* - *Period* + 1) through and including (*Index*).

Annual: The *Annual* value is used to annualize the Volatility calculation. The default is 255 (trading days). You can optionally enter a different value to annualize the Volatility calculation. Another commonly used value is 365 (calendar days). The Options model page in Ensign uses 365.

Historical Volatility Calculation:

```
a:=0; b:=0;
for j:= Index-Period+1 to Index do a:=a + ln( close(j) / close(j-1) );
a:= a / Period;
for j:= Index-Period+1 to Index do b:=b + sqrt( ln( close(j) / close(j-1) ) - a);
Volatility:= 100 * sqrt( 255 / (Period - 1) * b);
```

EXAMPLE: The following program opens a daily chart and prints the Historical Volatility for the last 20 bars on the chart (annualized with 365 calendar days). The Historical Volatility is often used in Options analysis. The value indicates how volatile the given market has been during the specified time period.

```
begin
  Chart('EUR/USD.D');
  writeln(Volatility(BarEnd, 20, 365)); {Calc Volatility for last 20 bars}
end;
```

WWW

SYNTAX: **WWW**(*WebAddress*: string);

DESCRIPTION: The **WWW** command is used to open your default web browser and display the specified *WebAddress*. This command might be useful to open the browser to an on-line Trading web site, when a trade is triggered.

PARAMETERS:

WebAddress: Specifies the web page to display in the browser.

EXAMPLE: The example opens various web pages when ESPL buttons 1, 2, and 3 are clicked on the Run ESPL form.

```
begin
  if ESPL=1 then www('www.ensignsoftware.com');
  if ESPL=2 then www('http://www.yahoo.com');
  if ESPL=3 then www('www.google.com');
end;
```

While...Do

SYNTAX: **While** *ConditionalExpression* **Do** {statements to do if *ConditionalExpression* is True}

DESCRIPTION: The **While** statement is used to loop through some code. The loop is executed until the *ConditionalExpression* is False. The *ConditionalExpression* is reevaluated at the beginning of the loop. A block of code can be executed by encasing the code with **Begin** and **End** statements. Each statement in the **Begin...End** block should end with a semicolon.

PARAMETERS:

ConditionalExpression is a logical expression that can be evaluated to a Boolean value of True or False.

EXAMPLE: The following program increments and prints the value of *j* until *j* >= 7.

```
begin                                {Start of Main Programming code}
  j:=0;                              {Initialize j with a value of zero}
  while j<7 do begin                {Loop while j is less than 7}
    inc(j);                          {Increment the value of j by 1}
    writeln(j);                      {Print j}
  end;                                {end of while loop block}
end;                                  {End of program}
```

Window

SYNTAX: *Window*: integer;

DESCRIPTION: The *Window* variable specifies which Window to work with. The *Window* variable must be set before an ESPL function (which uses the *Window*) is called. The *Window* variable is automatically set by the **FindWindow**, **Chart**, and **Quote** functions. The *Window* variable can be assigned or read. *Window* is set to zero to default to the window which called the script.

EXAMPLE: The following program opens a chart (which automatically sets the *Window* variable so that it points to the chart). The **TextOut** command prints some text on the chart (since the *Window* variable has been set). The **TextOut** command know which chart to print on based on the *Window* variable. The **FindWindow** command is used to set the *Window* variable so that it points to a specific window.

```
begin
  Chart('EUR/USD.D');                {The Window variable points to the Chart}
  Pause(1);                          {wait one second for chart to load and draw}
  TextOut(5,10,'Ensign Software');    {TextOut uses Window to print on Chart}
end;
```

WinExec

SYNTAX: **WinExec**(*Program*: string): integer;

DESCRIPTION: The **WinExec** function is used to RUN another windows program. This allows you to start a completely different application using the ESPL programming language. For example, the ESPL language could be used to start a Spreadsheet program, Word Processor, Analysis program, etc. If the function succeeds, the return value will be a number greater than 31. If the function fails, the return value will be one of the following error values:

- 0 - The system is out of memory or resources.
- 2 - The specified file was not found.

PARAMETERS:

Program: The *Program* parameter is a text string containing the path and FileName of the program to run. If the parameter does not include a directory path, Windows searches for the executable file in this sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

EXAMPLE: The following program runs the eSignal Turbo Data Manager program.

```
begin
  WinExec('C:\ESIGNAL\DATA MANAGER\WINROS.EXE');
end;
```

WinExist

SYNTAX: **WinExist**(*WindowCaption*: string): integer;

DESCRIPTION: The **WinExist** function returns True or False if a particular window is open (or program is running). The function will search all open programs and return True if there is a window caption that matches the *WindowCaption* specified in the parameter.:

PARAMETERS:

WindowCaption: This text string contains the window caption for a program..

EXAMPLE: The following code will return True if the SnagIt program is running.

```
begin
  if WinExist('SnagIt') then
    writeln('SnagIt is running')
  else
    writeln('SnagIt is not running');
end;
```

Write Writeln

SYNTAX: **Write**(*Expr1* [, *Expr2* ..., *ExprX*]);
Writeln(*Expr1* ..., *ExprX*);

DESCRIPTION: The **Write** and **Writeln** statements are used to print text to the output window. **Write** prints on a row and does not move to the next row. **Writeln** terminates a line after printing its text, and starts print on a new row.

PARAMETERS:

Expr1, Expr2: **Write** and **Writeln** can print any numeric, string, or boolean value.
Integer expressions/variables are printed as integers.
Real expressions/variables are printed using decimals.
Boolean expressions/variables print the string 'True' or the string 'False'.

NOTE: Use the **Str**, **Format**, and **Align** functions to format numbers with decimal places, and print the string. Use the tab character, #9, to tab to the next column. Columns are eight characters wide in the output window. Use `writeln;` to print a blank line in the output window.

EXAMPLE: The following program uses the **Write** and **Writeln** commands to print text in the output window.

```
var
  i, j, k: integer;
begin
```

```
write('Sample Text: ');
writeln('Same line, but ends with linefeed.');
```

`i := 56;`
`j := 17;`
`k := 5;`

```
writeln(i, #9, j, #9, k);      {Align the values with Tab columns}
end;
```

ESPL Sample Programs

The following ESPL programs can be used as example programming for Ensign.

Plotting Study Lines on a Chart

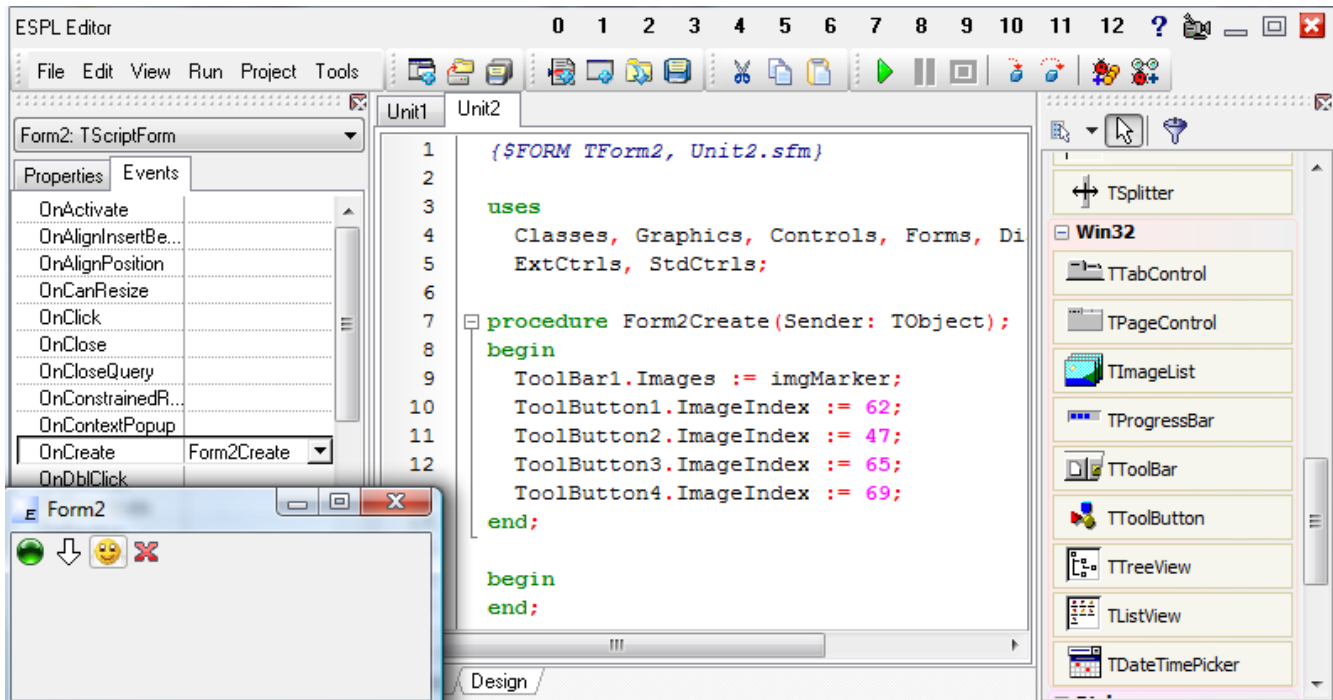
The following code will plot 2 lines on a chart. Click the RUN ESPL 100 button to apply the study to a chart.

```
procedure PlotLine;
var
  i:integer;
  MyValue,MyValue2: real;
begin
  for i:= BarBegin to BarEnd do
  begin
    MyValue := (Low(i) + High(i) + Open(i))/3;
    MyValue2:= Low(i) ;
    SetUser(1,MyValue,i);
    SetUser(2,MyValue2,i);
  end;
end;

begin
  if ESPL = 100 then PlotLine;
end;
```

ToolBar and ToolButton

This example shows a form using TToolBar, TToolButton and imgMarker.



At design time, a TToolBar object was added to the form and aligned for alTop. Then four TToolButtons were added to the ToolBar.

At run time, the imgMarker list of images from Ensign 10 is assigned to the ToolBar images property, and the ImageIndex properties assigned for the ToolButtons.

Five predefined TImageLists used by Ensign 10 have been exposed. They have these names: imgList16, imgList24, imgList32, imgMarker, and imgLine. See the Appendix for the [imgMarker](#) indexes and the [imgList16](#) indexes.

Of course, a TImageList can be added to the form, and one's own images can be added to the component. At design time, the TToolBar images property can be set to the TImageList on the form, and the ToolButton imageindex property set.

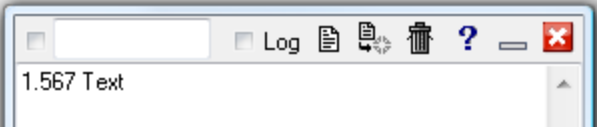
The click event for each of the toolbuttons can be written to perform a desired action.

TStringGrid Example

ESPL supports a variety of arrays, each with different characteristics. Let me summarize the choices.

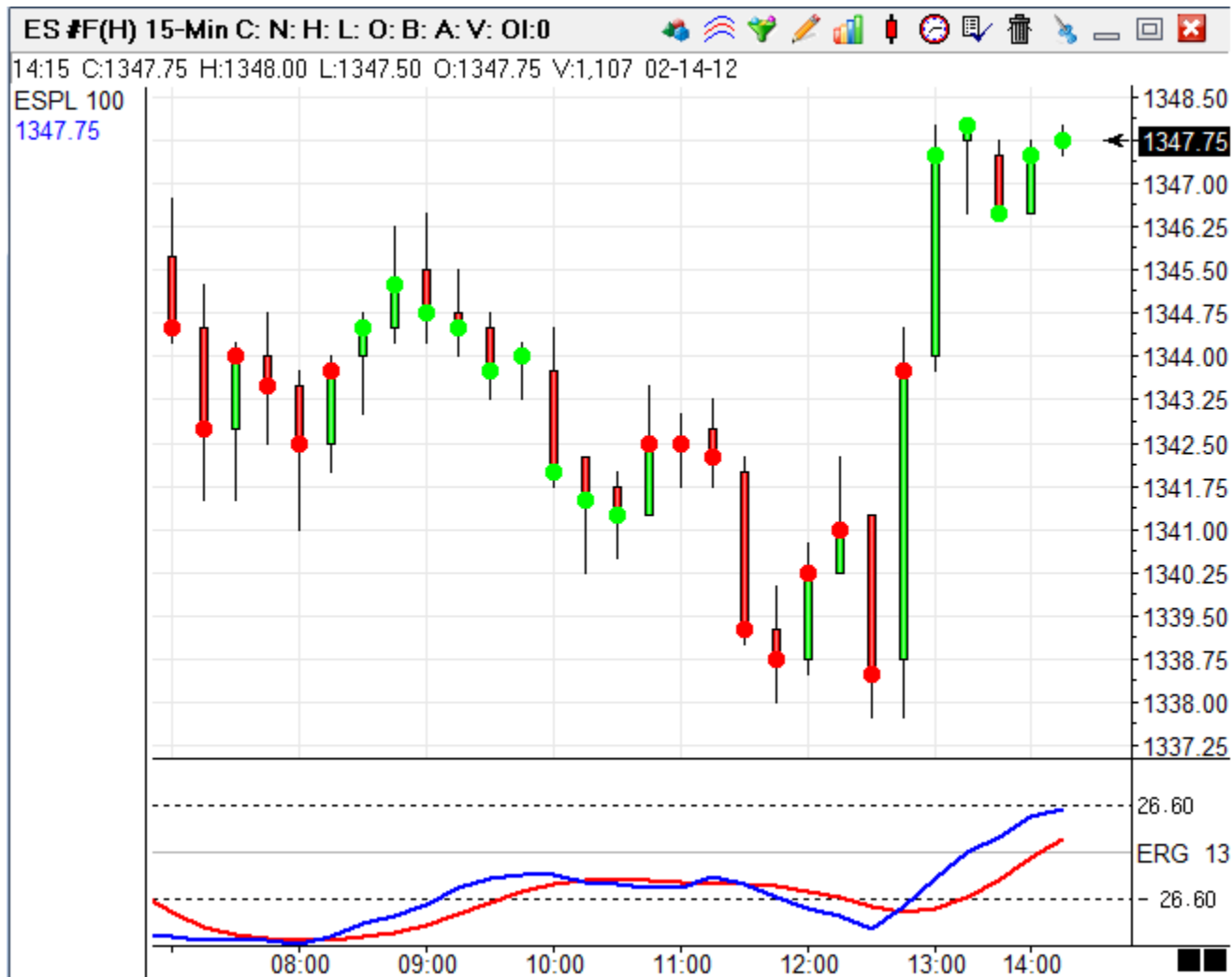
- Multi-dimensional arrays are declared with the [] construct at compile time. See the [Arrays](#) topic.
- [TArrays](#) are single dimensional and semi-automatic in their dimension.
- [Variant arrays](#) can be dimensioned and redimensioned, but they also are single dimensional.
- TLists and [TStringLists](#) are single dimension. They dynamically grow as elements are added.
- The TStringGrid object is 2-dimensional with RowCount and ColCount properties. And these properties can be set at run time. The values are written and read in the string grid cells. See this example.

```
Unit1
1  uses
2    Classes, Grids;
3
4  begin
5    grid := TStringGrid.Create(Application);
6    grid.RowCount := 5;
7    grid.ColCount := 3;
8    grid.Cells[0,1] := 1.567;
9    grid.Cells[1,3] := 'Text';
10
11    writeln(grid.Cells[0,1], ' ', grid.Cells[1,3]);
12    grid.Free;
13  end;
```

A screenshot of a console window with a toolbar containing icons for Log, Print, Copy, Paste, Delete, Help, and Close. The text '1.567 Text' is displayed in the console area.

Study Rising Falling Flag

The following ESPL example will plot GREEN circles when the Study Average is rising, and RED circles when the Study Average is falling. This chart has an Ergodic study and the ESPL study applied.



GetStudy

The [GetStudy](#) statement references for 1st line, 2nd line, 3rd line, and 4th line refer to the various Lines that can be plotted by a Study.

- The 1st line is the main study line.
- The 2nd line is most often the Average of the main study line.
- Sometimes there isn't a 3rd line for a study....sometimes there is.
- The 4th line is most often the Spread line values.

The example below is testing the GetStudy 15 value.... (Is the 2nd line Rising?)

```

uses
  Classes, Graphics, Controls, Forms, Dialogs;
procedure CheckErg;
var
  i, iHandle:integer;
begin
  iHandle := FindStudy(eERG);
  for i := BarBegin to BarEnd do
    begin
      if GetStudy(iHandle,15,i) then
        Plot(1,Last(i),i,0,0,21,clLime)
      else
        Plot(1,Last(i),i,0,0,21,clRed);
    end;
  end;
begin
  if ESPL=100 then CheckErg;
end;

```

Click ESPL button 100 to apply and run the ESPL study on the chart.

Creating ESPL DLLs

The Delphi used in this article is Embarcadero Delphi 2010. Any compiler and language can be used to make a DLL as long as the output is in flat “C”-type format (eg. No Objects, Classes or Strings, being passed as parameters or returned).

Step 1: Make the DLL in Delphi

Click File | New | Other | Dynamic Link Library. This is the code for a DLL named ESPL_DLL.

```
library ESPL_DLL;
const ESPL_NAME = 'ESPL_DLL.DLL';

{$R *.res}

function ESPL_Sum(AValue1: real; AValue2: real): real; stdcall;
begin
  result := AValue1 + AValue2;
end;

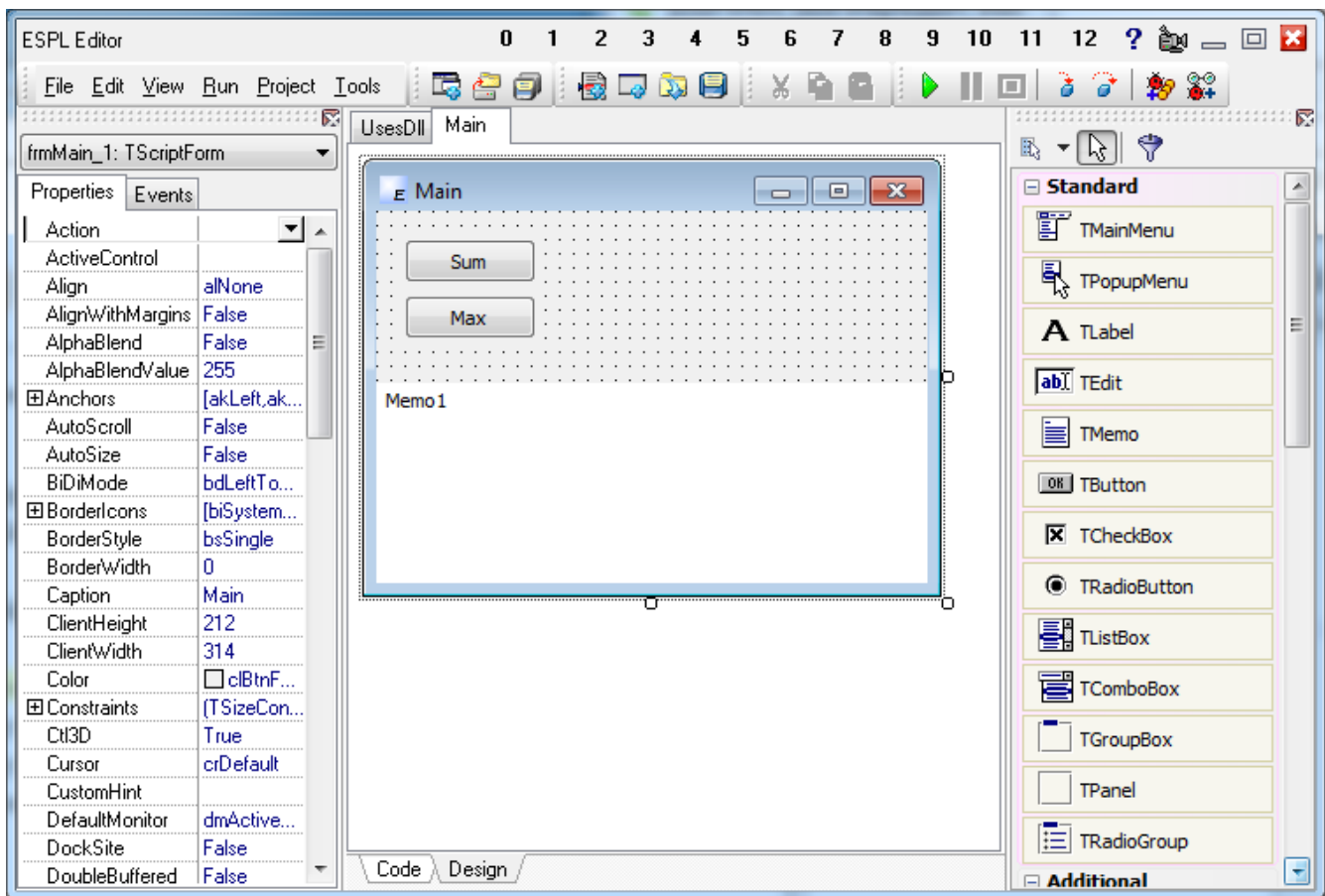
function ESPL_Max(AValue1: real; AValue2: real): real; stdcall;
begin
  if (AValue1 >= AValue2) then
    result := AValue1
  else
    result := AValue2;
end;

exports
ESPL_Sum,
ESPL_Max;

begin
end.
```

Step 2: Make the ESPL Form

Here is the form in Ensign 10's ESPL IDE, with 2 buttons and a memo component.



Step 3: Add the source code

```
{$FORM TfrmMain, Main.sfm}

uses Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

function ESPL_Sum(AValue1: real; AValue2: real): real; stdcall; external
  'ESPL_DLL.dll';

function ESPL_Max(AValue1: real; AValue2: real): real; stdcall; external
  'ESPL_DLL.dll';

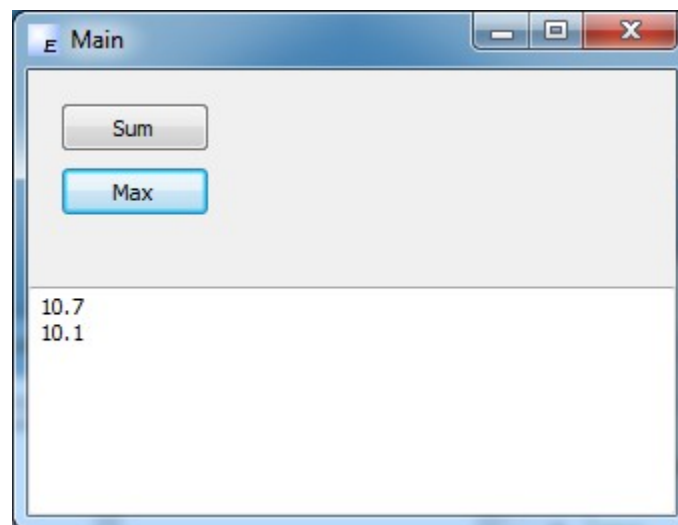
procedure btnSumClick(Sender: TObject);
var rValue: real;
begin
  rValue := ESPL_Sum(3.5, 7.2);
  Memo1.Lines.Add(rValue);
end;

procedure btnMaxClick(Sender: TObject);
var rValue: real;
begin
  rValue := ESPL_Max(8.7, 10.1);
  Memo1.Lines.Add(rValue);
end;
```

```
procedure frmMainShow(Sender: TObject);  
begin  
    Memo1.Clear();  
end;  
  
begin  
end;
```

Step 4: Run the program script

1. Click the ESPL Run button
2. Click the Sum button
3. Click the Max button



Appendix

USES Clause Libraries

Buttons Classes ComCtrls CommDlg Controls Dialogs ExtCtrls	Forms Graphics Grids ImgList IniFiles Menus Messages	StdCtrls StrUtils System SysUtils Types Windows
--	--	--

StrUtils Library Statements

AnsiReplaceStr AnsiResemblesText AnsiReverseString AnsiContainsStr AnsiContainsText AnsiLeftStr AnsiMidStr AnsiRightStr AnsiStartsStr AnsiStartsText AnsiEndsStr AnsiEndsText AnsiReplaceText	ContainsStr ContainsText DecodeSoundexInt DecodeSoundexWord DupeString EndsStr EndsText IfThen LeftStr LeftBStr MidStr MidBStr PosEx ReplaceStr ReplaceText	ResemblesText ReverseString RightStr RightBStr SearchBuf Soundex SoundexCompare SoundexInt SoundexProc SoundexWord SoundexSimilar StartsStr StartsText StuffString
---	---	---

Additional ESPL Statements

Append Assigned AssignFile CreateOleObject FilePos FileSize	GetActiveOleObject Int Interpret Machine Odd Raise ReadLn	Reset Rewrite Scripter SetOf VarArrayCreate VarArrayHighBound VarArrayLowBound VarIsNull
--	---	---

SysUtils Library Statements

Abort	ElementToCharIndex	HashName	StrPas
AdjustLineBreaks	ElementToCharLen	IncAMonth	StrPos
AnsiCompareFileName	EncodeDate	IncludeTrailingBackslash	StrRScan
AnsiCompareStr	EncodeTime	IncludeTrailingPathDelimiter	StrScan
AnsiCompareText	ExcludeTrailingBackslash	IncMonth	StrToBool
AnsiDequotedStr	ExcludeTrailingPathDelimiter	IsAssembly	StrToBoolDef
AnsiExtractQuotedStr	ExtractFileDir	IsDelimiter	StrToCurr
AnsiLastChar	ExtractFileDrive	IsLeadChar	StrToCurrDef
AnsiLowerCase	ExtractFileName	IsLeapYear	StrToDate
AnsiPos	ExpandFileNameCase	IsPathDelimiter	StrToDateDef
AnsiQuotedStr	ExpandUNCFileName	IsValidIdent	StrToDateTime
AnsiSameStr	ExtractFilePath	Languages	StrToDateTimeDef
AnsiStrAlloc	ExtractRelativePath	LastDelimiter	StrToFloat
AnsiStrComp	ExtractShortPathName	LoadStr	StrToFloatDef
AnsiStrLComp	FileAge	MsecsToTimeStamp	StrToInt64
AnsiStrLComp	FileClose	NextCharIndex	StrToInt64Def
AnsiStrLastChar	FileCreate	OutOfMemoryError	StrToIntDef
AnsiStrLComp	FileDateToDateTime	QuotedStr	StrToTime
AnsiStrLower	FileExists	RemoveDir	StrToTimeDef
AnsiStrUpper	FileGetAttr	RenameFile	StrUpper
AnsiUpperCase	FileGetDate	ReplaceDate	SysErrorMessage
AppendStr	FileIsReadOnly	ReplaceTime	SystemTimeToDateTime
BoolToStr	FileOpen	SameFileName	TextPos
ByteToCharIndex	FileRead	SameStr	TextToFloat
ByteToCharLen	FileSearch	SameText	TimeStampToDateTime
ByteType	FileSeek	SetCurrentDir	TimeStampToMsecs
ChangeFileExt	FileSetAttr	Sleep	TimeToStr
ChangeFilePath	FileSetDate	StrAlloc	TryEncodeDate
CharInSet	FileSetReadOnly	StrBufSize	TryEncodeTime
CharLength	FileWrite	StrByteType	TryFloatToCurr
CharToByteIndex	FindClose	StrCat	TryFloatToDateTime
CharToByteLen	FindFirst	StrCharLength	TryStrToBool
CharToElementIndex	FindNext	StrComp	TryStrToCurr
CharToElementLen	FloatToCurr	StrCopy	TryStrToDate
CheckWin32Version	FloatToDateTime	StrDispose	TryStrToDateTime
CompareStr	FloatToStr	StrECopy	TryStrToFloat
CompareText	FloatToStrF	StrLComp	TryStrToInt
CreateDir	FloatToText	StrLCat	TryStrToInt64
CurrentYear	FormatCurr	StrLComp	TryStrToTime
CurrToStr	FormatDateTime	StrLComp	UIntToStr
CurrToStrF	FormatFloat	StrLCopy	WideCompareStr
DateTimeToFileDate	FloatToDecimal	StrNextChar	WideCompareText
DateTimeToStr	FloatToTextFmt	StrPCopy	WideLowerCase
DateTimeToString	GetCurrentDir	StrPLCopy	WideSameStr
DateTimeToSystemTime	GetFileVersion	StrEnd	WideSameText
DateTimeToTimeStamp	GetFormatSettings	StrLen	WideStrAlloc
DateToStr	GetLocaleChar	StrLower	WideUpperCase
DecodeDateFully	GetLocaleFormatSettings	StrMove	
DeleteFile	GetLocaleStr	StrNew	
DirectoryExists	GetModuleName		
DiskFree	GetTime		
DiskSize			

Study Constants

eAcc	Accumulation Distribution	eMom	Momentum
eAdx	Directional Movement Index	eMrg	Mid Range
eArn	Aroon Study	eMacd	MACD Oscillator
eAsh	Heikin-Ashi	eOvr	Overlay
eAsi	Accumulation Swing Index	ePac	Price Action
eAtr	Average True Range	ePaf	Point & Figure
eAve	Moving Average	ePar	Parabolic Stop
eBal	On Balance Volume	ePda	Predictive Average
eBol	Bollinger Bands	ePvi	Price Volume Indicator
eBow	Rainbow Histogram	ePvp	Pesavento Patterns (Swing Lines)
eBnd	Color Band	eReg	Regression Channel
eCci	Commodity Channel Index	eRoc	Rate of Change
eChi	Chaikin Indicator	eRsi	Relative Strength Index
eCyc	Cycle Forecast	eSmi	Stochastic Momentum
eDon	Donchian Channel	eSto	Stochastics
eDvg	Divergence	eSwg	Swing Lines (Pesavento Patterns)
eDyo	Design Your Own	eTex	Triple Average
eGrid	Study Grid Object	eTnd	Auto Trend Lines
eErg	Ergotic Indicator	eTrl	Trailing Stop
eEspl	ESPL Study	eTrx	Trix Oscillator
eEsplTool	ESPL Draw Tool	eUlt	Ultimate Oscillator
eHlo	High Low Stop	eUni	Uniform Channel
eHst	Price Histogram	eUsr	User defined symbol
eHull	Hull Moving Average	eVlt	Volatility Stop
eKel	Keltner Channel	eVwap	Volume Weighted Average Price
eMap	Ensign Map	eWlm	William's %R
eMfi	Money Flow Index	e3pb	3 Point Break

Data Point Constants

0	eLast, eClose	Close	18	eTrueRange	True Range
1	eHigh	High	19	eTrueHigh	True Range High
2	eLow	Low	20	eTrueLow	True Range Low
3	eOpen	Open	21	eMidPoint	$(H+L)/2$
4	eNet	Net	22	eMid3	$(H+L+C)/3$
5		Abs(Net)	23	eMid4	$(H+L+C+O)/4$
6	eVolume	Volume	24		$(H+L+C+C)/4$

7	eTickCount	Tick Count	25		(C-O)
8	eInterest	Open Interest	26		Abs(C-O)
9	eAskVol	Ask Volume	27		(C+O)/2
10	eBidVol	Bid Volume	28		(C-O)/(H-L)
11	eAskRatio	Ask Ratio	29		Abs(C-O)/(H-L)
12	eBidRatio	Bid Ratio	30		(C-L)/(H-L)
13	eBuyPress	Buy Pressure	31		(O-L)/(H-L)
14	eSellPress	Sell Pressure	32	ePercent	100*(C-L)/(H-L)
15	eBuyRatio	Buy Ratio	33		100*(O-L)/(H-L)
16	eSellRatio	Sell Ratio	34		(C-(H+L)/2)/(H-L)
17	eRange	Range			

Bar Constants

eColor	Colorbar color	eMonth	MonthOf(timestamp)
eDate	Date as long integer	eSecond	Seconds portion of timestamp
eDateTime	Adjusted for local timezone	eTime	Timestamp in minutes format
eDay	DayOf(timestamp)	eYear	YearOf(timestamp)

























Feed Constants

e1stInternet	Refresh source	eIQFeed	IQFeed
e2ndInternet	Refresh source	eNinja	Ninja Trader
eATCBrokers	ATC Brokers	eOpenECry	OpenECry
eBarChart	Bar Chart	eSignal	eSignal
eEnsign	Ensign Internet	eTraderBytes	Trader Bytes
eFXCM	FXCM	eTransAct	TransAct Futures
eIB	Interactive Brokers		

Markers

0												
13												
26												
39												
52												
65												
78												
91												
104												
117	3	4	5	6	7	8	9	A	B	C	D	E
130	H	L	X	a	b	c	d	e	f	×	LH	=H
143	HL	=L	LL	←	←	←	←	←	←	←	←	←
156			BAR	CAL	DEG	SLP	STY	UNIT	VOL	VOL	BGD	HLT
169		A/B	VOL	LBL	D-T	Δ\$	‰\$	‰	\$			

Image List

												
0	1	2	3	4	5	6	7	8	9	10	11	12
												
13	14	15	16	17	18	19	20	21	22	23	24	25
												
26	27	28	29	30	31	32	33	34	35	36	37	38
												
39	40	41	42	43	44	45	46	47	48	49	50	51
												
52	53	54	55	56	57	58	59	60	61	62	63	64
												
65	66	67	68	69	70	71	72	73	74	75	76	77
												
78	79	80	81	82	83	84	85	86	87	88	89	90
												
91	92	93	94	95	96	97	98	99	100	101	102	103
												
104	105	106	107	108	109	110	111	112	113	114	115	116
												
117	118	119	120	121	122	123	124	125	126	127	128	129
												
130	131	132	133	134	135	136	137	138	139	140	141	142
												
143	144	145	146	147	148	149	150	151	152	153	154	155
												
156	157	158	159	160	161	162	163	164	165	166	167	168
												
169	170	171	172	173	174	175	176	177	178	179	180	181
												
182	183											
