Appendix 5.3 User Guide

Expression
User Guide
March 2004

Contents

1. What Is Expression?

1.1 Overview 4
1.2 Applications for Expression 4

2. Expression Programming

2.1 Overview 5
2.2 Labels 5
2.3 Commands 5
2.4 Functions 6
2.5 Assignments 6
2.6 Strings 6
2.7 Variables 7
2.8 String Lists 7
2.9 Using Lists 8
2.10 Forms
- 2.10.1 Form Programming 9
- 2.10.2 Creating Forms 10
- 2.10.3 Events 10
- 2.10.4 Form Controls 11

3. User Interface

3.1 Development Environment 12
3.2 Form Designer 13
3.3 Application Builder 14
3.4 Debugging Scripts 15

4. Reference Guide

4.1 Command Reference
- 4.1.1 Parameter-Less Commands 16
- 4.1.2 Conditional Statements / 4.1.3 Iteration 16
- 4.1.4 Displaying Messages 17
- 4.1.5 Running Applications 17
- 4.1.6 Working With The Clipboard 18
- 4.1.7 Branching Through Scripts 18
- 4.1.8 Working With Files / Directories / 4.1.9 Shutting Down Windows 19
- 4.1.10 Form Programming 20
- 4.1.11 List Programming 21
- 4.1.12 Language Modifiers 22
- 4.1.13 Parsing Strings 22
- 4.1.14 Play 23
- 4.1.15 Using The Registry 23
- 4.1.16 Shell 24
- 4.1.17 Delaying Scripts 24

4.2 Function Reference
- 4.2.1 Integer / 4.2.2 Floating Point Arithmetic 25
- 4.2.3 String Manipulation / 4.2.4 Comparison Functions 26
- 4.2.5 Return Data Functions 27
- 4.2.6 List Functions 28
- 4.2.7 Dialog Functions 29

4.2.8 Windows / File Functions 30
4.2.9 Registry Read Function / 4.2.10 System Information Function 31
4.2.11 Volume Information Function 32

5. Troubleshooting

5.1 Error Messages 34
5.2 Debugging Applications 35
5.3 Frequently Asked Questions 35
5.4 Support 36

1. What Is Expression?

1.1 Overview

Expression is a programming tool that enables you to quickly develop simple batch procedures or dialog-based programs for Windows as easily as writing a batch file or Basic program for DOS. It includes an interactive editor and debugger for creating programs, called scripts, which are written in the Expression scripting language.

The package also includes tools such as a Form Designer which enables you to develop Windows Applications in a WYSIWYG environment.

Expression has a simple syntax, with English-like commands, spreadsheet-like formula functions and type less variables. Scripts can be tested instantly using the development environment. You can then create an EXE which can then be run just like any other Windows application.

Expression is not intended to be an alternative to programming languages like C, Basic or Pascal for application development - but it is a better choice when you need a simple utility or a quick solution to a problem. Most Expression scripts can be written and tested in a matter of minutes once you are familiar with the language.

Together with Example Scripts in the \Examples directory in the folder where you installed Expression, you should be familiar in no time!

1.2 Applications for Expression

The following are some tasks which could be created with the Expression language:

- Create an 'intelligent' start-up script to load applications or perform housekeeping tasks at start-up.
- Create a start-up menu with buttons to select which applications are loaded;
- Create software installation programs.
- Create simple utilities such as resource monitors.
- Create front-ends for DOS programs, which can run invisibly in the background so that it looks as if a Windows application is running.
- Produce interactive multimedia applications that display bitmaps and play sounds.

2. Expression Programming

2.1 Overview

The Expression scripting language has been designed to be simple, flexible and easy to use. The language has three main elements:

- Labels
- Commands
- Assignments

2.2 Labels

Labels are used as the target of GOTO and GOSUB commands. They start with a colon and are followed by the label name.

This is an example of a label:

```
:LABEL
```

Labels, GOTO and GOSUB commands are used to change the order of execution of script commands.

2.3 Commands

It is recommended that commands are indented using spaces (or tabs) for readability.

A command consists of the command name (see Command Reference) followed optionally by a string.

The string is used as the argument (or parameters) to the command. Most commands have only a single argument, but some have more than one in which case commas are used to separate them. A space must separate the command from the string. Commands are not case-sensitive.

Here are some examples of commands:

```
TITLE "My First Application"
INFO "Hello and welcome"
```

Strings may include variable and function references, which are evaluated before the command is carried out. Here is an example of commands containing variable and function references:

```
MyFile = "C:\Autoexec.bat"
IF FileExists(MyFile)
  INFO "File " + MyFile + " exists."
ENDIF
```

2.4 Functions

Expression contains a range of functions (see Function Reference) which are evaluated at run time and return a string containing information.

Functions start with their name. The argument(s) to the function are in the form of a string enclosed in parentheses. The parentheses must be present even if the function takes no arguments. For functions that take more than one argument the arguments are separated by commas.

Here are some examples of functions:

```
Ask("Do you want to continue?")

Equal(MyFile,"WIN.INI")
```

Because Expression evaluates functions from left to right, by detecting left hand and right hand brackets, you cannot use brackets to alter priority of evaluations. This however will not pose a problem with anything you wish to do with Expression as maths functions such as Addition and Subtraction are evaluated like a normal function.

2.5 Assignments

Like commands, it is recommended that they be indented using spaces (or tabs) for readability.

An assignment consists of a variable name, an equals symbol and a string, (which may contain variable or function references) each separated by spaces.

Here are some examples of assignments:

```
MyFile = Input("Please enter a file name")
Message = "Backing up to drive " + Drive
Total = Sum(SubTotal, VAT)
```

If you find that you get errors while trying to place an assignment, it may be due to the lack of spaces around the equal symbol.

2.6 Strings

In Expression, all variables and parameters to commands and functions are strings. Strings can contain embedded variables and functions which are substituted or evaluated as the string is processed from left to right.

So if MyDrive contains "C:" then the string:

```
"Drive " + MyDrive + " has " + VolInfo(MyDrive,"F") + "Kb free"
```

would be evaluated to:

Drive C: has 32456Kb free

Note the presence of the symbols " and +. Any string must be surrounded by quotes; otherwise Expression will try to substitute each word for a variable. Secondly, if you wish to place more than one string on the same line, such as the first example above, you join them using the + symbol. The + symbol is the equivalent of a "String Concatenate" function.

Text within double quotes is interpreted simply as text: no variable substitution or function evaluation will be performed. Quotes can appear anywhere in a string to prevent substitution or evaluation for a particular section, as in this example:

If you wish to utilise quote characters in your strings, you may use the Chr() function, passing to it 34.

2.7 Variables

Expression in theory allows you to utilise an unlimited number of variables.

Variable names start with a letter, and may have a variable number of letters and numbers after that. No special characters are permitted. (Special Characters can be defined as anything that is not alpha-numeric.)

Expression variables are type less strings of unlimited length. Because variables are type less it is up to you to ensure that, for example, where a function requires a numeric value the variable passed to it contains a string which is a valid number. There are functions available to check if a string can be classed as a number. (See Numeric() function)

Expression also supports string lists. You can think of these as arrays of strings, which can be maintained in sorted or unsorted order and accessed either sequentially or by item number.

There are two reserved variables in Expression, "true" and "false". True is a more suitable word for the digit "1" which also signifies true. False is a more suitable word for "" (null). Expression treats anything none-null as true.

2.8 String Lists

String lists can be used to hold the contents of text files as well as lists of ASCII data. The length and number of strings in a string list are limited only by available memory.

Expression supports an unlimited number of string lists, created at run-time. In addition, the list box and combo box form controls are also string lists, and can be manipulated using the same list commands.

String lists can be unsorted, in which case strings remain in the order they were loaded and items may be added at the end using LIST ADD, inserted using LIST INSERT or replaced using LIST PUT. Using the LIST SEEK and LIST PUT commands, and the Match() and Item() functions, lists can be used as random access files.

Alternatively lists may be sorted, in which case Expression sorts items upon addition.

For more information on how lists are used, see [2.9] Using Lists.

2.9 Using Lists

Syntax:

```
LIST ADD,<list>,<text>
LIST ASSIGN,<list>,<list2>
LIST CLEAR,<list>
LIST CLOSE,<list>
LIST COPY,<list>
LIST CREATE,<list>{, SORTED}
LIST DELETE,<list>
LIST INSERT,<list>
LIST PASTE,<list>
LIST PUT,<list>,<text>
LIST SEEK,<list>,<record number>
LIST WRITE,<list>,<text>
```

Description:

These LIST commands are used to modify, save and dispose of string lists. The parameters <list> and <list2> must be either the name of a string list created at run time or the name of the form list box to which the command will apply. An error will occur if the list does not already exist or the record number (for the SEEK command) is out of range.

ADD is used to add an item to the list. You use this to add items to a sorted list (they are inserted in the correct position according to the sort order) or to append items to an unsorted list, much as you would write successive lines to a text file.

ASSIGN copies the contents of <list2> to <list>

CLEAR is used to remove all items from a list and reset the item pointer to -1.

CLOSE must be used to dispose of a list that you have CREATED once you have finished with it. This releases the memory it used.

COPY causes the contents of <list> to be copied to the Clipboard.

CREATE creates a new, empty string list. The option SORTED specifies whether the list is to be maintained in ASCII code order. The item pointer (index) points to -1.

DELETE deletes the item at the position in the list indicated by the index (pointer). When a list item is deleted, the index (item pointer) is reset to -1. This can be modified by means of the LIST SEEK command.

INSERT inserts a new item in front of the current pointer position. After the insertion, the index position is the item following the one just inserted. Note that you should not use INSERT with sorted lists.

PASTE causes <list> to be filled with the contents of the Clipboard (as text.)

PUT is used to write the specified text to the position in the list indicated by the index (pointer). PUT lets you treat a list as a random access file. Note that you cannot use PUT with sorted lists.

SEEK is used to set the index pointer to a specific item number.

OK:

Set to true if the command is successful, false if it fails.

Example:

```
LIST CREATE, MyList, Sorted
LIST LOADFILE, MyList, "c:\unsortedfile.txt"
LIST SAVEFILE, MyList, "c:\sortedfile.txt"
LIST CLOSE, MyList
```

Example 2:

```
LIST CREATE, MyList
LIST LOADFILE, "names.txt"
LIST SEEK,1, 3
LIST WRITE,1,"This is item 4 in the list"
LIST SAVEFILE,1,LIST.TXT
LIST CLOSE,1
```

2.10 Forms

2.10.1 Form Programming

Expression allows you to create a form which functions as a window for your script application. The form may contain a number of controls, such as label controls and a status panel for displaying captions and other information, and edit controls, check boxes and list boxes which can not only display information but allow interaction with the user, plus buttons which tell you when to process information by generating events.

You create a Form using the FORM CREATE command. You can design the form interactively using the Form Designer, which will then generate the correct Expression code to create the form. The form typically includes buttons which users can press when they want the script to do something with the information in the form.

Note that the language makes it possible to write more sophisticated programs in which conditional statements and/or calculations are performed between the FORM CREATE and the FORM SHOW to vary the appearance of the dialog in real-time.

When you close a form, either by clicking its Close button or by executing a FORM CLOSE command, it does not close straight away. A FORMCLOSE event is generated. A script can respond to this event by saving information in the dialog, then issuing an EXIT command which will close the application.

When a button is pressed it generates an event. For a user-defined button the name of the event is the name of the button followed by CLICK; for example, when the OK button is

pressed an OKCLICK event occurs. The form close command (and selecting Close from the system menu) generates a FORMCLOSE event. Another example of an event is the EXIT event, which can occur when a control loses focus. See Events for more information.

There are two ways to process events. You can use WAIT EVENT. This halts the script entirely until an event occurs. When it does, you can test it using the Event() function, carry out whatever processing is required, and if appropriate loop back to the WAIT EVENT command to wait for the next event.

The form will remain present until the program terminates. You can however HIDE it using FORM HIDE, although this will not free up resources.

The simplest way to write a dialog-based Expression program is to use the Form Designer. This lets you design the dialog using the dialog designer and then generates a skeleton program with labels for all the possible events. All you need do is write the code to respond to each event.

2.10.2 Creating Forms

The FORM command is used to create a form for the program and manage its controls. The form becomes the program's main window, and will exist until the program terminates.

The form definition takes the basic form of:

`FORM CREATE, ...`	define form window, title, size and properties
`FORM ADD, ...`	define the form controls
`FORM SHOW`	show the form

Form controls begin with the control type appended to the FORM ADD command, which is optionally followed by details such as the control name, size, position and initial value. See [2.10.4] Form Controls for more information.

Expression's form designer allows you to design a form interactively, and then generates the FORM CREATE command for you.

2.10.3 Events

Events occur when the user interacts with a form which you have created using the FORM CREATE command. Some events occur by default, such as those generated by buttons. Others only occur if you specify a style in a form control definition.

You can halt your script and wait for an event to occur using the WAIT EVENT command. You can find out the type of event using the Event() function.

BUTTON events occur when a button is pressed. The type of event is <name>CLICK, where <name> is the name of the button. So when the user presses a button labelled OK an OKCLICK event occurs.

CLICK events are optional. They cause a <name>CLICK event to occur when the form control is clicked with the mouse. In the case of elements such as LIST or CHECK you can use the event to perform some action dependent on the new setting of the control.

DOUBLECLICK events are optional. They cause a <name>DOUBLECLICK event to occur when the form control (a list) is double clicked with the mouse.

FORMCLOSE events occur when the user closes the dialog from the system menu or using the FORM CLOSE command, or when the user shuts down Windows. The script should respond to this event by saving any unsaved data and terminating.

EXIT events are optional. They cause a <name>EXIT event to occur when the dialog element loses the input focus. This event is available for EDIT and COMBO dialog elements. It can be used to trigger a procedure to validate the data that has been entered in the control's input field.

2.10.4 Form Controls

Form controls can be seen as typical commands that all start with FORM ADD. The FORM ADD command is invalid until the FORM CREATE command has been successfully executed. If one would try to execute a FORM ADD command without first executing FORM CREATE they would be greeted with an error.

Most form controls have parameters, which are appended to control name. The parameters are separated by commas. The name parameter, where required, is mandatory and is used to address the control when you want to write text to it or read text from it. Note that control names are never run through variable replacement or function parser. This means you cannot store a control name in a variable or calculate a name at run time. Most of the remaining parameters are optional, and may be left as null or omitted; when omitted, Expression will use suitable defaults (or zero, depending on circumstance). Position co-ordinates are relative to the client area of the form, rather than the desktop.

The following form controls are available:

```
BITMAP,<name>,<top>,<left>,<width>,<height>,<filename>{, <styles>}
BUTTON,<name>,<top>,<left>,<width>,<height>,<caption>
CHECK,<name>,<top>,<left>,<width>,<height>,<caption>,<value>{, <styles>}
COMBO,<name>,<top>,<left>,<width>,<height>,<value>{, <styles>}
EDIT,<name>,<top>,<left>,<width>,<height>,<value>,<style>{, <styles>}
LIST,<name>,<top>,<left>,<width>,<height>,<style>{, <styles>}
PROGRESS,<name>,<top>,<left>,<width>,<height>,<value>
STATUS,<name>,<value>,<style>
LABEL,<name>,<top>,<left>,<width>,<height>,<value>{, <styles>}
```

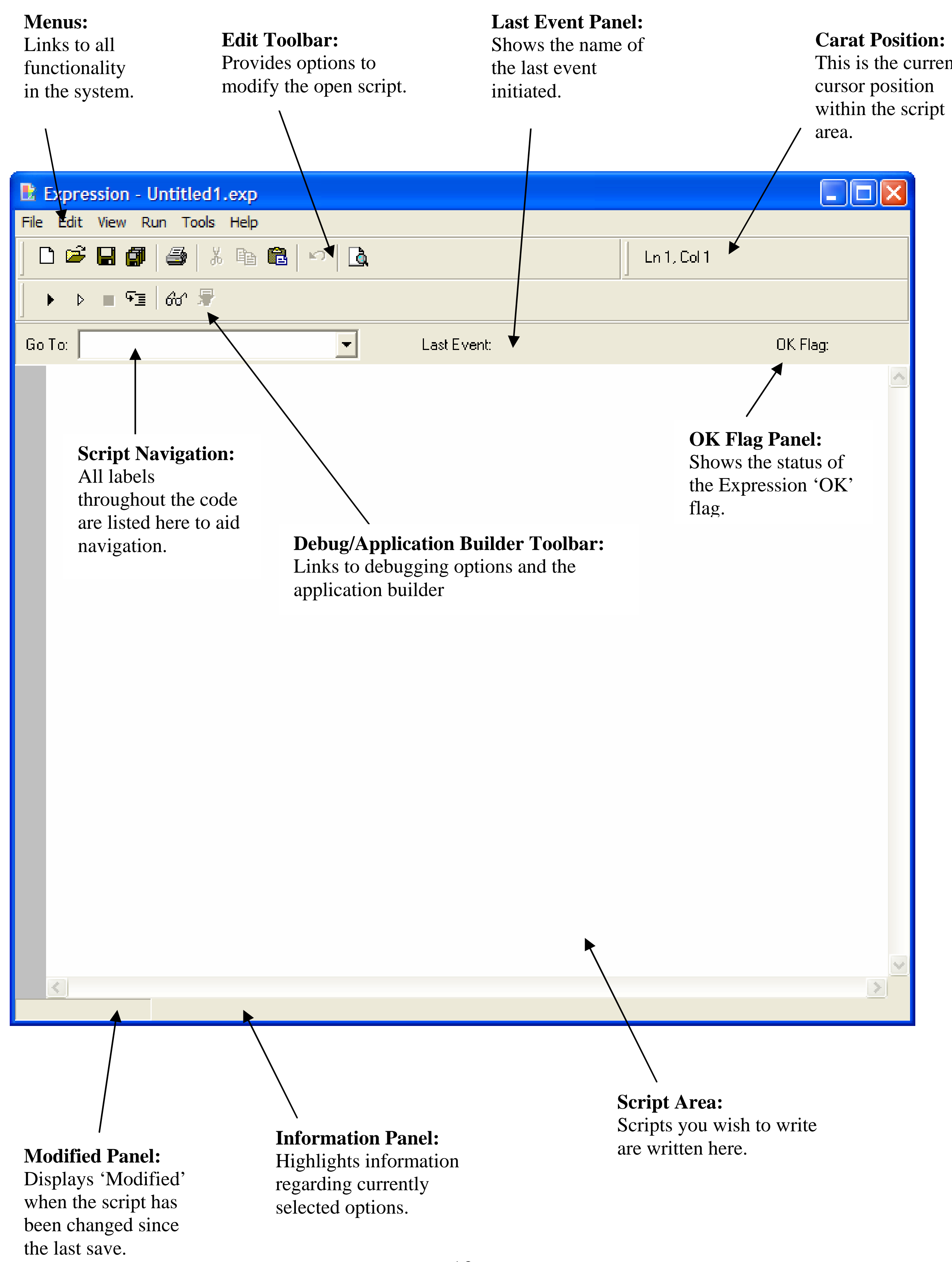
The order of specifying the form controls can be important. Imagine the form to be 3D. The latter controls are placed on top of those previous, which means if any controls overlap, the ones created first can be hidden.

An easier way to create a form than by working out a list of form controls is to use the form designer.

3. User Interface

3.1 Development Environment

The following is what you will see when you first load Expression. All typical editing options such as saving, printing, opening and creating files are supported:

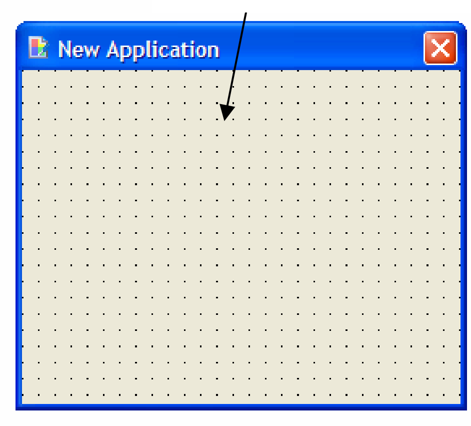


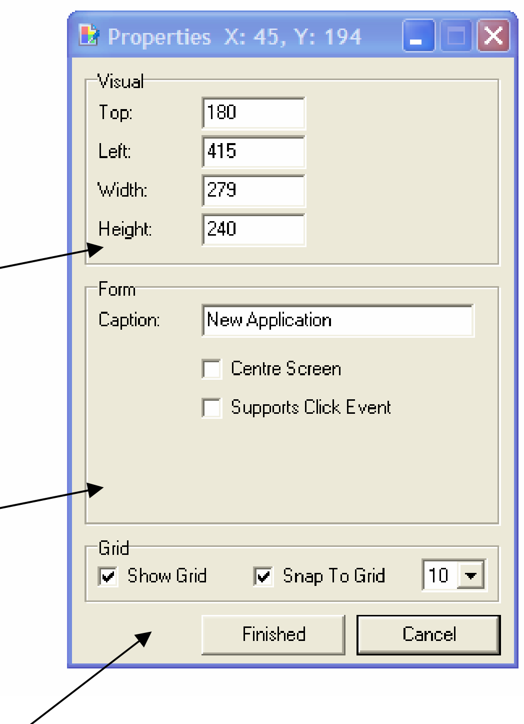
3.2 Form Designer

The form designer is accessible by selecting the 'Tools' Menu from the Development Environment and clicking 'Form Designer'. When you do this, the following is displayed. The form designer allows you to quickly and easily design forms for use with your scripts.

Controls Window
This window displays buttons corresponding to the controls you can place on the form. There is no limit to how many of each control you can have on your form

Designer Window
This window is the equivalent of what your users see when they execute your application. You may modify any controls on this form by selecting them, moving them and sizing them.





Visual Properties Panel
This panel displays the co-ordinates of the currently selected control (or form).

Control Options Panel
The options on this panel change according to the selected control. The options on this panel alter the behaviour of the currently selected control.

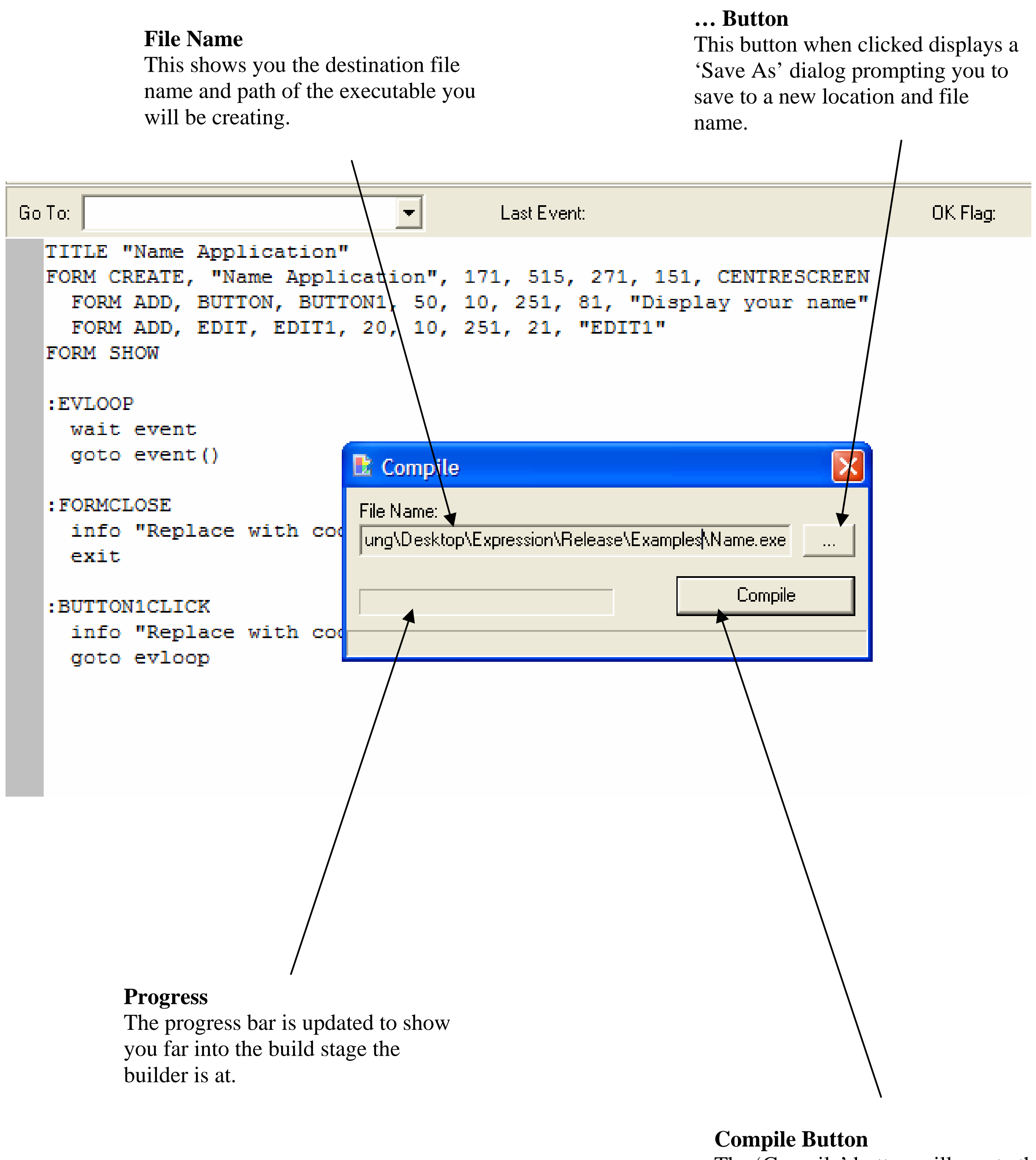
Grid and Closure
The grid options effect the designer form. If a grid is showing and snap to grid is selected, then controls moved around a form will be sized according to the size of the grid.

The Finished button generates code based on the designer form, and the Cancel button loses any changes.

3.3 Application Builder

The application builder is accessible by selecting the 'File' Menu from the Development Environment and clicking 'Create Executable from Script' or by pressing the build icon on the Debug /Application Builder toolbar.

When you do this, the following is displayed. The Application Builder allows you to create self-sufficient executable files for the Windows platform.

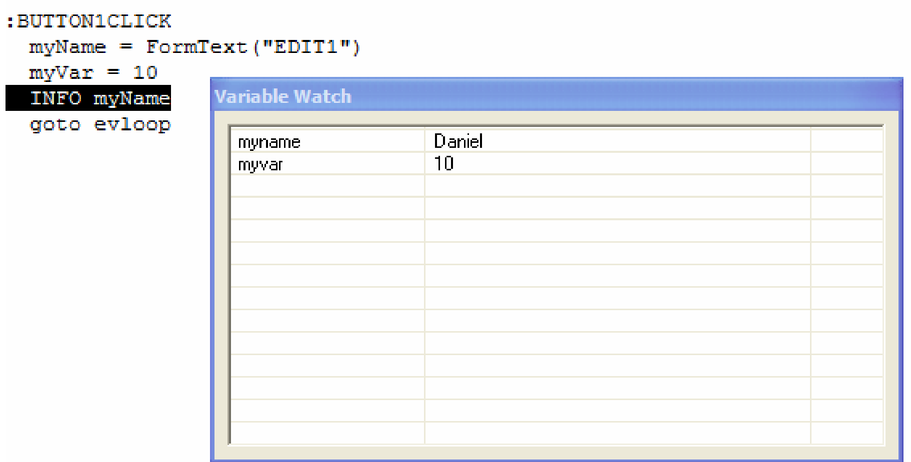


3.4 Debugging Scripts

There are several debugging options available in Expression. They are outlined below:

Variable Watch Window

The variable watch window is accessible by pressing the variable watch icon on the Debugging toolbar:



The variable watch window lists all variables in use by the script and their contents. You can use this to monitor how the script modifies these variables at run time.

Step Through Code

Pressing F8 or pressing the step-through icon on the Debugging Toolbar allows you to step through your script line by line so you can monitor jump conditions and changes to variables. The line currently being executed is highlighted as shown below:

```
:BUTTON1CLICK
  myName = FormText("EDIT1")
  myVar = 10
  INFO myName
  goto evloop
```

Executing Scripts

The Run and Trace icons allow you to simulate the application as if it was being executed on the target system. The difference between Run and Trace is that trace highlights the current line being executed – this leads to a performance disadvantage but it allows you to monitor the script.

Execution can be stopped at any time using the Stop button.

4. Reference Manual

4.1 Command Reference

4.1.1 PARAMETER-LESS COMMANDS

Commands:

BEEP
Sounds a Windows beep.

EXIT
When executed after a GOSUB command, EXIT causes execution to continue at the line following the GOSUB. Otherwise, EXIT causes execution of the script to terminate.

STOP
Halts execution of a script unconditionally. It is similar to EXIT but will terminate a script even from within a subroutine.

OK:

Unchanged.

4.1.2 CONDITIONAL STATEMENTS

Syntax:

```
IF  <string>
   ... commands executed if string not null (true)
ELSEIF <string>
   ... commands executed if string not null (true)
ELSE
   ... commands executed if string is null (false)
ENDIF
```

OK:
Unchanged.

Example:

```
IF ask("Do you want to continue?")
  info "You answered YES"
ELSE
  IF ask("Are you sure?")
    info "You answered NO"
  ELSE
    info "Make your mind up"
  ENDIF
ENDIF
```

4.1.3 ITERATION

Syntax:

```
REPEAT
   ... commands ...
UNTIL <string>
```

Description:

Commands between the REPEAT and UNTIL lines are repeatedly executed while the result of <string> evaluates to null (false). If the string is non-null then script execution continues on the line following UNTIL.

REPEAT commands may be nested. For clarity it is a good idea to indent the commands nested within REPEAT ... UNTIL as shown in the example.

OK:

Leaves unchanged.

Example:

```
Test = 0
REPEAT
  Test = @SUM(Test,1)
UNTIL @EQUAL(Test,6)
```

4.1.4 DISPLAYING MESSAGES

Commands:

```
INFO <string>
Displays a dialog box containing an information symbol icon and the
message <string>.  Execution of the script continues when the OK
button is pressed.

WARN <string>
Displays a dialog box containing an exclamation symbol icon and the
message <string>.  Execution of the script continues when the OK
button is pressed.
```

OK:

Set to true.

Example:

```
WARN "Critical Error!"
```

4.1.5 RUNNING APPLICATIONS

Syntax:

```
RUN  <filename>{, <parameters>}
```

Description:

Runs the program <filename>, with the additional parameters specified.

The variants of the command are:

RUN	- run as a normal window;
RUNH	- run in a hidden window;
RUNM	- run as a maximized window;
RUNZ	- run minimized as an icon;

If you run a DOS program and want it to close its window when it finishes make sure the program's "close window on exit" property is set.

Note: If there are any issues running an application that utilises a long file name format, try using the ShortName function to convert the file name into a DOS-compatible short name.

OK:

Unchanged

Example:

```
RUN "winword.exe"
RUNZ "c:\utils\pkzip.exe"
```

4.1.6 WORKING WITH THE CLIPBOARD

Syntax:

```
CLIPBOARD APPEND,  <string>
CLIPBOARD CLEAR
CLIPBOARD SET, <string>
```

Description:

The CLIPBOARD command is used to put data into the Windows clipboard.

CLIPBOARD APPEND adds the contents of <string> to whatever is already in the clipboard. Successive appends add the text on a new line.

CLIPBOARD CLEAR empties the clipboard of any data that was in it.

CLIPBOARD SET sets the contents of the clipboard to the text <string>.

OK:

Unchanged.

Example:

```
CLIPBOARD SET, "Hello Clipboard!"
```

4.1.7 BRANCHING THROUGH SCRIPTS

Commands:

GOTO <label>

Causes script execution to continue at the command following the label :<label>.

GOSUB <label>

Causes script execution to continue at the command following the label :<label>. When an EXIT command is encountered, execution will jump back to the command following the GOSUB.

OK:

Unchanged.

Example:

```
GOSUB sayhello
INFO "Goodbye"
EXIT
```

```
:sayhello
  INFO "Hello"
  EXIT
```

4.1.8 WORKING WITH FILES / DIRECTORIES

Syntax:

```
DIRECTORY CHANGE, <path>
DIRECTORY CREATE, <path>
DIRECTORY DELETE, <path>
DIRECTORY RENAME, <path1>, <path2>
FILE DELETE, <file_path>
FILE RENAME, <file_path_1>, <file_path_2>
```

Description:

The DIRECTORY command is used to change, create, delete or rename directories.

DIRECTORY CHANGE changes the current directory to the one named in <path>. If <path> is on a different drive to the current drive then this is changed as well. This command is similar in operation to the MS-DOS CHDIR command.

DIRECTORY CREATE creates a new directory named <path>. If necessary, it will recursively create all the subdirectories in the path. This command is similar in operation to the MS-DOS MKDIR command.

DIRECTORY DELETE removes the directory named in <path>. The directory must be empty or it will not be removed.

DIRECTORY RENAME renames the directory named in <path1> to the directory named in <path2>.

FILE DELETE erases the file named in <file path>. This command is similar in operation to the MS-DOS DEL command.

FILE RENAME renames the file named in <file_path_1> to <file_path_2>. A file can be renamed from one directory to another (in other words, moved) only if both directories reside on the same logical drive. This command is similar in operation to the MS-DOS REN command.

OK:

True if the operation is successful; false if not.

Example:

```
directory delete, "C:\tmp1"
directory create, "C:\test\subdir1\subdir2"
```

4.1.9 SHUTTING DOWN WINDOWS

Syntax:

```
EXITWIN <exit_option>
```

Description:

This command allows you to shut down Windows.

The valid options are:

SHUTDOWN	A normal shutdown.
REBOOT	Shuts down Windows and reboots the system.
LOGOFF	Logs the user off the system.
FORCE	Forcibly logs off Windows without allowing programs to cancel the log off request.

OK:

Unchanged.

Example:

```
EXITWIN REBOOT
```

4.1.10 FORM PROGRAMMING

Syntax:

```
FORM ADD, <control_type>,<control_name>, <control_description>
FORM CLEARSEL, <control_name>
FORM CLOSE
FORM CREATE, <title>, <top>, <left>, <width>, <height>{, <styles>}
FORM CURSOR{, <parameter>}
FORM DISABLE, <control_name>
FORM ENABLE, <control_name>
FORM FOCUS, <control_name>
FORM HIDE, <control_name>
FORM SET, <control_name>, <text>
FORM SETPOS, <control_name>, <top>, <left>, <width>, <height>
FORM SHOW, <control_name>
FORM TITLE, <title>
```

Description:

FORM ADD adds a new control to the form. For further information, see Form Controls

FORM CLEARSEL clears the selected item in a list box control named <control_name>.

FORM CLOSE generates a FORMCLOSE event.

FORM CURSOR{, <parameter>} sets the mouse cursor accordingly. Possible parameters are : WAIT, HAND, UP, CROSS, IBEAM, NODROP, DRAG, HSPLIT, VSPLIT, WE, NWSE, NS, NESW and SIZE. The command FORM CURSOR sets it back to the default cursor. To provide feedback to the user, your script program should show the hourglass (WAIT) cursor or whichever is more suitable in response to a user action.

FORM CREATE function creates a (hidden) form on which controls are added using FORM ADD, and which is then displayed using FORM SHOW. It therefore contains just the title, size, position and any styles that relate to the form. This command has many different parameters which form the description. It is described in more detail

in the topics Creating Forms and Form Controls. You would normally use the Form Designer to generate form code.

FORM DISABLE and FORM ENABLE disable and enable the control named <control_name>.

FORM FOCUS sets the input focus to the control named <control_name>.

FORM HIDE and FORM SHOW hide and show the control named <control_name>. If <control name> is omitted then this command will effect the form itself.

FORM SETPOS lets you alter the size or position of a form element (or the form itself) at run-time. <control_name> is the name of the form element, such as EDIT1. To modify the form itself, use the parameter FORM. The <top>,<left>,<width> and <height> are the values for those respective properties. For each parameter value that is left null, 0 is inserted.

FORM TITLE sets the text in the title bar of the form window to <title>. This is different from setting the title of the script itself, which is done using the TITLE command.

Form styles are:
CLICK: generates a FORMCLICK event if user clicks on the surface of the form.
CENTRESCREEN: the default position of the form is the centre of the desktop.

OK:

Unchanged.

Example:

```
FORM CREATE, "New Application", 139, 356, 264, 90
  FORM ADD, BUTTON, BUTTON1, 14, 11, 97, 25, "Display Contents"
  FORM ADD, EDIT, EDIT1, 16, 124, 121, 21, "Default Text"
FORM SHOW
```

4.1.11 LIST PROGRAMMING

Syntax:

```
LIST <command>, <list>, <parameters>
```

Description:

The LIST command is used to create, manipulate and dispose of string lists. Expression allows you to have a logical unlimited amount of string lists, each identified by a name. List boxes and combo boxes (list controls) which appear in a form can also be treated as string lists. The parameter <list> must be either a string list name or the name of the form list control to which the command will apply.

A list can either be sorted, or will retain the order in which the information was entered.

OK:

Set to true if the command is successful, false if it fails.

Example:

```
LIST CREATE, MYLIST1, SORTED
LIST ADD, MYLIST1, "Fred Jones"
LIST ADD, MYLIST1, "John Smith"
```

4.1.12 LANGUAGE MODIFIERS

Commands:

`OPTION FIELDSEP, <separator_character>`

`OPTION FIELDSEP sets the field separator to be recognised by the PARSE command when it splits a string up into its separate fields. The default is the vertical bar (|).`

`TITLE <string>`

`The title set here is the application title. It appears only in messages and input boxes. The title of the task bar button is the title of the main window.`

OK:

Unchanged.

Example:

```
OPTION FIELDSEP, " "
```

4.1.13 PARSING STRINGS

Syntax:

```
PARSE "<field_list>", <string>
```

Description:

The PARSE command provides an easy way to split up strings which represent records in a database into their constituent data fields.

The <field list> parameter consists of a semicolon-separated list of variables. The <string> contains the data record.

OPTION FIELDSEP sets the field separator used by the PARSE command when it splits a string up into its separate fields. By default this is the vertical bar character: |.

OK:

Unchanged.

Example:

```
AllVolumeInfo = volinfo("C", "FNYTZS")
PARSE "freespace;volname;format;type;total;serialnumber",
AllVolumeInfo

info "All Data:" + tab() + AllVolumeInfo
info "Free Space:" + tab() + freespace
info "Volume Name:" + tab() + volname
info "File Format:" + tab() + format
info "Type:" + tab() + TYPE
info "Total Space:" + tab() + Total
info "Serial number:" + tab() + SERIALNUMBER
```

4.1.14 PLAY

Syntax:

```
PLAY  <file_name>{, WAIT}
```

Description:

Plays an audio wave (*.WAV) file. If the parameter WAIT is specified the script does not proceed to the next command until the sound is finished.

OK:

Leaves unchanged.

Example:

```
PLAY Windir() + "\media\chimes.wav", WAIT
```

4.1.15 USING THE REGISTRY

Syntax:

```
REGISTRY DELETE, <root_key>, <subkey>
REGISTRY WRITE, <root_key>, <subkey>, <name>, <value>
```

Description:

The REGISTRY command modifies the value of keys in the Windows registry.

REGISTRY DELETE will delete <subkey> from the registry, whether it be a folder within the registry (in which case all of its keys will be deleted, and then itself) or just a key.

REGISTRY WRITE will write string data into

<root_key> can be one of:

ROOT	specifies HKEY_CLASSES_ROOT
CURUSER	specifies HKEY_CURRENT_USER
LOCAL	specifies HKEY_LOCAL_MACHINE
USERS	specifies HKEY_USERS
DYNDATA	specified HKEY_CURRENT_CONFIG

<subkey> can be a folder within the reigstry or a key:

"\Software\Microsoft\Windows\CurrentVersion"
"\Software\Microsoft\Windows\CurrentVersion\CommonFilesDir"

<name> is the name of the key when using the REGISTRY WRITE command.

<value> is the string being written to <name>.

WARNING: Modifying the system registry is potentially dangerous, and can render Windows unable to run. Ensure you have a backup of the registry files before experimenting.

OK:

Set to false if the REGISTRY command fails.

Example:

```
REGISTRY WRITE, ROOT, ".wsc", "", "wnscript"
REGISTRY WRITE, LOCAL, "Test", "Name", "31"
REGISTRY DELETE, ROOT, ".tmp"
```

4.1.16 SHELL

Syntax:

```
SHELL  <operation>, <filename>
```

Description:

Performs the Windows shell operation <operation> on the file <filename>.

The available <operation> parameters are:

open - opens a file name with its associated application;
execute - executes an application;

OK:

Unchanged

Example:

```
SHELL open, "mailto:joe@bloggs.com"
SHELL execute, "c:\autoexec.bat"
```

4.1.17 DELAYING SCRIPTS

Syntax:

```
WAIT <interval>
WAIT EVENT
```

Description:

Pauses execution of the script for a period of <interval> seconds If <interval> is not specified then the default period is 1 second. Interval may be a decimal number.

The WAIT EVENT command is used in dialog programming. It is only valid when a dialog is being displayed. It causes the script to wait indefinitely until an event occurs such as a button being pressed.

OK:

Not affected.

Example:

```
INFO "Going to wait for 5 seconds"
WAIT 5
INFO "Done"
```

4.2 Function Reference

4.2.1 INTEGER ARITHMETIC

Syntax:

```
Function(<value1>{, <value2>})
```

Functions:

```
Abs(<value>)             : Returns absolute value of a number
Div(<value1>, <value2>) : Returns value1 \ value2
Mul(<value1>, <value2>) : Returns value1 * value2
Sub(<value1>, <value2>) : Returns value1 – value2
Sum(<value1>, <value2>) : Returns value1 + value2
Diff(<value1>, <value2>): Returns value1 – value2
Prod(<value1>, <value2>): Returns value1 * value2
Pred(<value1>)          : Returns value – 1
Succ(<value1>)          : Returns value + 1
```

Description:

These functions manipulate integer numbers.

OK:

Set to false if any value is none-numeric.

Example:

```
' Returns 10
INFO Abs(-10)
```

4.2.2 FLOATING POINT ARITHMETIC

Syntax:

```
Function(<value1>{, <value2>})
```

Functions:

```
Fabs(<value>)             : Returns absolute value of a number
FAtn(<value>)             : Returns arctangent of a number
FCos(<value>)             : Returns cosine of a number
FDiff(<value1>, <value2>): Returns value1 – value2
FDiv(<value1>, <value2>) : Returns value1 / value2
FExp(<value1>)            : Returns exponential of a number
FInt(<value1>)            : Returns integer portion of a number
FLn(<value1>)             : Returns logarithm of a number
FMul(<value1>, <value2>) : Returns value1 * value2
Frac(<value1>)            : Returns fractional portion of a number
FSin(<value>)             : Returns sine of a number
FSqt(<value>)             : Returns square root of a number
FSub(<value1>, <value2>) : Returns value1 – value2
FSum(<value1>, <value2>) : Returns value1 + value2
```

Description:

These functions manipulate floating point numbers.

OK:

Set to false if any value is none-numeric.

Example:

```
' Returns 10.398
INFO FAbs(-10.398)
```

4.2.3 STRING MANIPULATION FUNCTIONS

Syntax:

```
Function({<value1>, <value2>, <value3>})
```

Functions:

Cr()
Returns Carriage-Return to drop line.

Len(<string>)
Returns length of <string>.

Lower(<string>)
Returns <string> in lower case.

Pos(<source>, <find>)
Returns position of <find> within <source>.

Strdel(<source>, <pos1>, <pos2>)
Deletes characters <pos1> of length <pos2> from <source>.

Strins(<source>, <pos>, <string>)
Inserts <string> into <source> at position <pos>.

SubStr(<source >, <pos1>, <pos2>)
Returns a substring of <source> starting at <pos1> of length <pos2>.

Tab()
Returns Tab charater.

Trim(<string>)
Trims leading/trailing spaces and tabs from <string>.

Upper(<string>)
Returns <string> in upper case.

Description:

These functions manipulate strings.

OK:

Set to false if any function fails

Example:

```
Var1 = "hello"
info SubStr(Var1, 1, 3)
' "hel" is output
```

4.2.4 COMPARISON FUNCTIONS

Syntax:

```
Function(<value1>{, <value2>, <value3>})
```

Functions:

And("<string>", "<string>"{, "<string>" ... "<string>"})
Returns true (1) if all strings contain data.

Equal(<string>, ... ,<string>)
Returns true (1) if all strings are equal.

Greater(<value1>, <value2>)
Returns true (1) if <value1> is greater than <value2>

Less(<value1>, <value2>)
Returns true (1) if <value1> is less than <value2>

Not(<string>)
Returns true (1) if <string> is empty.

Null(<string>)
Returns true (1) if <string> is empty.

Numeric(<string>)
Returns true if <string> can be classed as a valid number.

Zero(<string>)
Returns true (1) if <string> is equal to zero.

Description:

These functions return true or false depending on a comparison.

OK:

Set to true if returns true, false if returns false

Example:

```
' Returns true ("1")
if And(-10, 100, "hello", "world")
  Info "All null/zero"
Endif
```

4.2.5 RETURN DATA FUNCTIONS

Syntax:

Function(<value1>{, <value2>, <value3>})

Functions:

AppPath()
Returns the path of the application itself, minus file name.

Asc("<string>")
This function returns the ASCII value of the first (or only) character in <string>.

CurDir()
This function returns the current directory that has been assigned to the application. By default it is the path from which the script is executed. It can be changed using the DIRECTORY CHANGE command.

Chr()
This function returns the character based on the ASCII <value> passed.

Date()
Returns the current date.

Hex(<value>)
Returns the hexadecimal equivalent of <value>.

Now()
Returns current date/time in a single string.

Ok()
Returns the current value of the OK flag.

Time()
Returns the current time.

WinDir()
Returns the location of the Windows directory.

Description:

These functions return true or false depending on a comparison.

OK:

Set to true if returns true, false if returns false

Example:

```
' Returns true ("1")
if And(-10, 100, "hello", "world")
  Info "All null/zero"
Endif
```

4.2.6 LIST FUNCTIONS

Syntax:

```
Function(<value1>{, <value2>, <value3>})
```

Functions:

Count("<list_name>")
This function returns the number of items in the list <list>. The parameter <list> must be either a list created at run time (a string list) or a list control on a form.

Index("<list>")
This function returns the current index (pointer) value for the string list <list>. The parameter <list> can also be a list control found on a form.
If <list> is a LIST or COMBO element and no list item is selected the value returned by this function is -1.

Item("<list>")
This function returns the text from the current index position in the list. The list may be a string list or a list on a form. The index position can be manipulated using the LIST SEEK command.

Match("<list>", <string>)
This function returns 1 (true) if a string in the string list <list>, starting from the current pointer position, contains text matching <string>. The match is not case-sensitive. The pointer is advanced to the matching item number, so you can obtain the contents of the

```
string using Item() or Next(), rewrite it using LIST PUT and obtain
the index value using Index(). If no match is found, null (false) is
returned and the index value is unchanged.
```

Next("<list>")

```
This function returns the contents of the current item in the string
list <list>, and then advances the pointer by 1. The pointer may be
set using the LIST SEEK command. For LIST and COMBO controls, if no
item is selected the index value is -1 and the value returned by
Next() will be null.
The parameter <list> must be either a string list created at run time
or a list control on a form.
You use the Next() function if you want to read sequentially through
the items in a list.
```

Description:

These functions can be used to modify a list.

OK:

Unchanged

Example:

```
LIST CREATE, MYLIST1, SORTED
LIST LOADFILE, MYLIST1, "NAMES.TXT"
LIST SEEK, MYLIST1, 9
INFO Item("MYLIST1")
LIST CLOSE, 1
```

4.2.7 DIALOG FUNCTIONS

Syntax:

```
Function(<value1>{, <value2>, <value3>})
```

Functions:

Ask("<string>")

```
This function displays a dialog box containing a question mark icon,
the message <string>, and buttons for Yes and No. The value 1 (true)
is returned if the user responds Yes; otherwise the function returns
null (false).
```

Input(<prompt>{, <default_result>})

```
This function displays a dialog box containing a message and a box
where user can enter data. The function returns the default result
if Cancel was pressed. If OK is pressed then the value typed in the
box is returned.
```

OpenDlg("<description>"{, <path>, <filename>})

```
This function shows the Windows Open File common dialog. The
function returns the default file name if the user clicks Cancel and
the chosen file name if the user clicks OK.
<description> must be in the following format:
"file description 1|*.ext1|file description 2|*.ext2"
<path> is optional. If it is not specified then the start directory
will be the current directory.
<filename> is optional. If it is not specified then the default file
name will be blank.
```

Query("<string>")

```
This function displays a dialog box containing a question mark icon,
the message <string>, and buttons for OK and Cancel. The value 1
(true) is returned if the user responds Yes; otherwise the function
returns null (false).
```

SaveDlg("<description>"{, <path>, <filename>})

```
This function shows the Windows Save File common dialog.  The
function returns the default file name if the user clicks Cancel and
the chosen file name if the user clicks OK.
<description> must be in the following format:
"file description 1|*.ext1|file description 2|*.ext2"
<path> is optional.  If it is not specified then the start directory
will be the current directory.
<filename> is optional.  If it is not specified then the default file
name will be blank.
```

Description:

These functions can be used to interact with a user.

OK:

True if OK was pressed, False if Cancel was pressed.

Example:

```
if Ask("Do you wish to continue?")
  info "OK, Continuing..."
else
  info "Never mind..."
endif
```

4.2.8 WINDOWS / FILE FUNCTIONS

Syntax:

```
Function(<value1>{, <value2>, <value3>})
```

Functions:

Event()

```
This function returns the name of the last event to have occurred. It
returns a null string if no event has occurred. After the function
has been called the event is cleared, so it should be stored in a
variable if you need to test it more than once.
```

Ext(<string>)

```
This function returns the extension portion of a file name if it is
possible to do so.  If it is not possible then null will be returned.
```

FormText("<control_name>")

```
This function returns the textual property or numerical value of a
control on an Expression form.
Passing "FORM" as the <control_name> will return the caption of the
form itself.
```

Name(<string>)

```
This function returns the name portion of a file name if it is
possible to do so.  If it is not possible then null will be returned.
```

Path(<string>)

```
This function returns the path up to and including the last backslash
of a string.  If no backslash is found then null is returned.
```

Shortname(<string>)
This function returns the DOS file name equivalent of <string> if string is a valid file name. If the string is not a valid file name then null is returned

Description:

These functions can be used to interact with files and forms.

OK:

Unchanged.

Example:

```
:Evloop
  wait event
  goto event()
```

4.2.9 REGISTRY READ FUNCTION

Syntax:

```
RegRead("<root_key>", <subkey>)
```

Description:

The RegRead function reads string keys from the registry.

<root_key> can be one of:

ROOT	specifies HKEY_CLASSES_ROOT
CURUSER	specifies HKEY_CURRENT_USER
LOCAL	specifies HKEY_LOCAL_MACHINE
USERS	specifies HKEY_USERS
DYNDATA	specified HKEY_CURRENT_CONFIG

<subkey> can be a folder within the reigstry or a key:

"\Software\Microsoft\Windows\CurrentVersion"
"\Software\Microsoft\Windows\CurrentVersion\CommonFilesDir"

<name> is the name of the key to read.

OK:

Set to false if function fails.

Example:

```
info RegRead("LOCAL", "Software\Microsoft\Windows\CurrentVersion\",
"CommonFilesDir")
```

4.2.10 SYSTEM INFORMATION FUNCTION

Syntax:

```
SysInfo("<information_required>")
```

Description:

This function returns information about the system.

The information_required contains parameters which can be:

FREEMEM	- return amount of space memory
BITSPERPIXEL	- return number of bits per pixel
SCREENHEIGHT	- return the screen height
SCREENWIDTH	- return the screen width
CPUTYPE	- returns CPU type
NUMBEROFPROCESSORS	- returns number of processors on the system
PROCESSORLEVEL	- returns the processor level
WINVER	- returns Windows version
WINEXTRA	- returns other Windows version information
COMPNAME	- returns the computer name
USERNAME	- returns the currently logged in username

OK:

Set to false if function fails.

Example:

```
INFO SysInfo("FREEMEM")
INFO SysInfo("BITSPERPIXEL")
INFO SysInfo("SCREENHEIGHT")
INFO SysInfo("SCREENWIDTH")
INFO SysInfo("CPUTYPE")
INFO SysInfo("NUMBEROFPROCESSORS")
INFO SysInfo("PROCESSORLEVEL")
INFO SysInfo("WINVER")
INFO SysInfo("WINEXTRA")
INFO SysInfo("COMPNAME")
INFO SysInfo("USERNAME")
```

4.2.11 VOLUME INFORMATION FUNCTION

Syntax:

```
VolInfo("<drive>", "<information_required>")
```

Description:

This function returns information about the specified volume.

If <drive> is omitted, the default volume "C" is used.

The information_required contains flags, which can be:

F - return amount of space free (Kb)
N - return volume name
S - return total space on drive (Kb)
T - return type of volume:removable, Fixed, Network, CD-ROM or RAM.
Y - returns a text string describing the file system type (for example, FAT32, NTFS.)
Z - returns the hard disk serial number.

If no <information type> is specified, the default N is used.

If more than one flag is specified, each item of information is separated by the field separator character. The data can be stored in separate variables using the PARSE command.

OK:

Set to false if function fails.

Example:

```
VolInfo = volinfo("C", "FNYTZS")
PARSE "freespace;volname;fileformat;type;totalspace;serialnumber",
Volinfo

info "All Data:" + tab() + yer
info "Free Space:" + tab() + freespace
info "Volume Name:" + tab() + volname
info "File Format:" + tab() + fileformat
info "Type:" + tab() + TYPE
info "Total Space:" + tab() + TotalSpace
info "Serial number:" + tab() + SERIALNUMBER
```

5. Troubleshooting

5.1 Error Messages

The following is a list of possible error messages while using Expression:

Error Code / String 101 / "Missing function name"
Error Reason You provided a left parenthesis “(” without a function name.

Error Code / String 201 / "Mismatched brackets"
Error Reason You provided a left or right parenthesis without its opposite bracket.

Error Code / String 202 / "Mismatched quotes"
Error Reason You started a string constant without ending with one.

Error Code / String 301 / "Invalid command"
Error Reason You specified a command that does not exist.

Error Code / String 401 / "Invalid character outside of string"
Error Reason You have string characters outside of quotes.

Error Code / String 501 / "Invalid function"
Error Reason You specified a function that does not exist

Error Code / String 502 / "Invalid parameter to function"
Error Reason You provided an invalid parameter to a function.

Error Code / String 601 / "Until without Repeat"
Error Reason You tried to initiate iteration without a Repeat that starts one.

Error Code / String 701 / "Missing ELSE or ENDIF"
Error Reason An IF statement was started but has no end.

Error Code / String 702 / "ELSE without IF"
Error Reason You tried to use conditional statements without first starting with IF.

Error Code / String 801 / "Label not found"
Error Reason You tried to jump or gosub to a label that does not exist.

Error Code / String 901 / "Missing parameter(s) to command"
Error Reason The command you are using requires more parameters. See help.

Error Code / String 1001 / "Command invalid when no form has been created"
Error Reason You cannot use form-dependant commands when no form exists.

Error Code / String 1002 / "Command invalid when a form already exists"
Error Reason You cannot use this command if a form exists.

Error Code / String 1003 / "Function invalid when no form has been created"
Error Reason You cannot use the function in question when no form exists.

Error Code / String 1004 / "Command invalid when a status bar already exists"

Error Reason	You cannot have more than one status bar on a form at any one time.
Error Code / String	1101 / "Control specified does not exist"
Error Reason	You tried to reference a control name that has not been assigned.
Error Code / String	1102 / "Invalid parameter to command"
Error Reason	You provided an invalid parameter in a command call. See help.
Error Code / String	1201 / "List Index out of bounds"
Error Reason	You tried to access an element in a list that does not exist.
Error Code / String	999999 / "There was a problem while processing this line"
Error Reason	If this error occurs, please see Support.

5.2 Debugging Applications

When a script does not behave as expected it can sometimes be difficult to work out why. Writing programs is never easy, but Expression comes with a set of tools that makes it as easy as possible to work out what is going wrong.

Start by resetting the program (click the Stop button, or press CTRL+F2) so that you start running from the beginning. Then step through your script a line at a time using the Single Step button or the F8 key. Open the Debug Window so that you can see the result of all the variables used by your script after each line has been executed. This is usually enough to work out what the problem is, if you think after each line about what the correct values are supposed to be.

Sometimes a command will not work, and will report that it does not work by setting the OK indicator to false, rather than by halting with an error message. The script language does this to give you a chance to cope with the error within your script, rather than have a user presented with a cryptic error code. However, if you don't use the Ok() function to test the result of OK, but just assume the command or function will work, a script will not work correctly if the command fails. You can check the status of OK at any time during debugging as it is shown in the debug-bar of the IDE.

A common source of problems is failure to get information from files. This is usually caused by path problems. If you only specify a filename and not a full path, the script will look in the current directory for the file. The current directory is not necessarily the directory in which the script program resides. You should always specify a full path when referencing any file. If the file belongs to the script and will always be kept in the same directory use AppPath() to get the directory of the script program.

5.3 Frequently Asked Questions

I want to put some quotes "" in a string, but they are always removed. Why?

Expression uses double-quotes as a method of determining when a constant string starts and ends. Within double-quotes, no variables are be substituted and no functions evaluated. Quotes act as toggles: they can appear anywhere in a string and turn the 'string-constant' effect on and off as the string is parsed from left to right. In the process the quotes are removed. To

get double-quote characters into a string you must use Chr(34) which is converted to the ASCII character with code 34, which is a double-quote.

Some events are missed, why is this?

Expression does not store events to be executed if it is in the middle of processing an event. First wait for the current event to finish, and then continue with the next.

5.4 Support

For further support, email your script and a detailed description of the problem to:

expression@mydan.com