A
C
D
E
F
G
I
L
P
R
S
U

A-Z Function Call Reference

CLOSE

A

FBAddBooleanConstant	Registers a boolean constant with the engne
FBAddConstantPrim	Registers a constant with the engine
FBAddDateConstant	Adds a date constant to the engine
FBAddNumericConstant	Adds a numeric constant to the engine.
FBAddStringConstant	Adds a string constant to the engine
FBAddVariable	Adds a variable to the current expression instance.

C

FBClearExpression	Clears the internal state of the expression
FBCopyValue	Copies a TValueRec structure.
FBCreateString	Creates a FormulaBuilder type string.

D

FBDateToPasString	Converts a FormulaBuilder date to a Pascal type String.

E

FBEnumFunctions	Enumerates all registered functions.
FBEvalExpression	Performs a quick, single statement expression evaluation.
FBEvaluate	Evaluate a specific expression and return its result as a string.
FBEvaluatePrim	Evaluate an expression instance and return the result as a TValueRec.

F

FBFreeConstant	Removes a constant from FormulaBuilders symbol table.
FBFreeConstants	Free all constants registered with FormulaBuilder.
FBFreeExpression	Free an expression instance and all associated memory.
FBFreeValue	Disposes of all memory associated with a TValueRec.
FBFreeVariable	Disposes of a variable associated with an expression instance.
FBFreeVariableList	Free all variables associated with an expression instance.

G

FBGetBooleanResult	Returns the boolean result of an expression.
FBGetBooleanVariable	Returns the value of a boolean variable.
FBGetConstantPrim	Obtains the value of a constant in a TValueRec record.
FBGetConstAsString	Returns the string representation of the value of a constant.
FBGetDateResult	Evaluates an expression instance and returns its date return value.
FBGetDateVariable	Retrieves the value of a date variable.
FBGetErrorString	Returns a description of a FormulaBuilder error code.
FBGetExpression	Returns the text expression assigned to the expression instance with FBSetExpression.
FBGetFloatResult	Evaluates the expression instance and returns its float result.
FBGetFloatVariable	Retrieves the value of a float variable.
FBGetFunctionCount	Determines the number of functions registered with FormulaBuilder.
FBGetFunctionProto	Returns information on a function registered with the engine.
FBGetIntegerResult	Evaluates the expression instance and returns its integer result.
FBGetIntegerVariable	Retrieves the value of an integer variable.
FBGetReturntype	Determines the expected return type of an expression.
FBGetStringResult	Evaluates the expression instance and returns its string result.
FBGetStringVariable	Retrieves the value of a String variable.
FBGetVarAsString	Returns the value of a variable as a string, without regard to its type.
FBGetVariableCount	Counts the number of variables added to an expression instance.
FBGetVariablePrim	Returns the value of a variable as a TValueRec record.
FBGetVarPtr	Returns a pointer to the value of a variable maintained in an expression's variable list.

I

FBInitExpression	Allocate a handle for a new expression.

L

FBlpzToDate	Convert the null-terminated string *source* to a FormulaBuilder Date type

P

FBParseAddConstant	Add a constant to the engine based on the result of a text expression.
FBParseAddVariable	Add a variable to an expression based on the result of a text expression.
FBPasStringToDate	Converts a Pascal style string to a FormulaBuilder Date
FBPeekVariable	Inspect a variable based on its index in an expression's variable list.
FBPeekVarVB	A VB compatible version of the FBPeekVariable function.

R

FBRegisterFunction	Registers a programmer defined function with FormulaBuilder.
FBReparseExpression	Reparses the expression previously set with a call to FBSetExpression.

S

FBSetBooleanVaria	Sets the value of a Boolean variable.

ble	
FBSetDateVariable	Sets the value of a Date variable.
FBSetExpression	Initializes the expression with its infix representation. This triggers the parsing phase of the evaluation process.
FBSetFloatVariable	Sets the value of a Float variable.
FBSetIntegerVariable	Sets the value of an Integer variable.
FBSetStringVariable	Sets the value of a String variable.
FBSetVarFromString	Sets the value of a variable from a valid string representation.
FBSetVariableCallbacks	Register functions to enable external variable processing.
FBSetVariablePrim	Sets the value of a variable from a TValueRec structure.
FBStringToDate	Converts a FormulaBuilder string to a FormulaBuilder date type.
FBStrncpy	Copy a specified number of characters from a FormulaBuilder string to a null-terminated string.

U

FBUnregisterFunction	Unregisters a programmer-defined function registered with the FBRegisterFunction call

ABS Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the absolute (positive) value of its argument

Syntax

ABS(x)

X is any number

See Also

SGN

ACOS Function

[See Also](#) [Math/Trig Functions](#) [A-Z Function Reference](#)

Description

Returns the arc cosine of a number.

Syntax

ACOS(*number*)

number is the cosine of the angle. The cosine can range from 1 to -1.

Remarks

The resulting angle is the angle whose cosine is *number*. The answer is returned in radians (from 0 to Pi). To convert the resulting radians to degrees, use the DEGREES function.

See Also

COS

PI

ACOSH Function

[See Also](#) [Math/Trig Functions](#) [A-Z Function Reference](#)

Description

Returns the hyperbolic arc cosine of x

Syntax

ACOSH(x)

X is any positive number greater than 1.

See Also

ASINH
ATANH
COSH

ACOT Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the inverse cotangent of an angle in radians.

Syntax

ACOT(x)

X is any number. If your *x* value is in degrees, use the RADIANS function to convert it to radians before passing it to this function

See Also

COT

PI

ACOTH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the inverse hyperbolic cotangent of an angle in radians.

Syntax

ACOTH(x)

X is any number between 1 and -1, excluding 1 and -1. If your *x* value is in degrees, use the RADIANS function to convert it to radians before passing it to this function

See Also

ACOT
COT
COTH

ACSC Function

See Also   Math/Trig Functions   A-Z Function Reference

Description

Returns the inverse cosecant of a number.

Syntax

ACSC(x)

X is any number such that |X| < 1. If your *x* value is in degrees, use the RADIANS function to convert it to radians before passing it to this function.

See Also

ACSCH

CSC

ACSCH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the inverse hyperbolic cosecant of a number.

Syntax

ACSCH(x)

X is any number.

See Also

CSC

CSCH

ASC Function

See Also String Functions A-Z Function Reference

Description
Returns a numeric (ANSI) code for the first character in a text string

Syntax
ASC(*text*)

Text is the string for which you want to determine the code.

ASEC Function

See Also   Math/Trig Functions   A-Z Function Reference

Description

Returns the inverse secant of an angle.

Syntax

ASEC(x)

X is the angle in radians, such that |X| < 1. If your *x* value is in degrees, use the RADIANS function to convert it to radians before passing it to this function.

See Also

- ASECH
- SEC

ASECH Function

See Also   Math/Trig Functions   A-Z Function Reference

Description

Returns the inverse hyperbolic secant of an angle.

Syntax

ASECH(x)

X is the angle in radians, such that |x| <= 1

Remarks

If you wish to convert a value expressed in degrees to radians, use the RADIANS function.

See Also

ASEC

SECH

ASIN Function

See Also Math/Trig Functions A-Z Function Reference

Description

ASIN(X) calculates the arc (inverse) sine of of an angle using the sine *x* of the angle.

Syntax

ASIN(x)

X is the sine of the angle, in the range -1 to 1.

Remarks

x is presumed to be in radians as opposed to degrees. To convert an angle from degrees to radians, use the RADIANS function. The result is an angle, in radians, from -Pi through Pi.

See Also

ASINH
PI
SIN

ASINH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the inverse hyperbolic sine of a number.

Syntax

ASINH(*x*)

Remarks

The inverse hyperbolic sine is the value whose hyperbolic sine is , so ASINH(SINH(x)) = x. x is any number floating point or integer value.

See Also

ACOSH
ATANH
SINH

ATAN Function

See Also		Math/Trig Functions		A-Z Function Reference

Description

Returns the arc (inverse) tangent of an angle by using its tangent.

Syntax

ATAN(*x*)

x is a number which represents the tangent of the angle.

Remarks

The result of ATAN is an angle, in radians, between -Pi/2 and Pi/2. To convert the resulting angle from radians to degrees, use the DEGREES function.

See Also

ATAN2
ATANH
Pi
TAN

ATAN2 Function

See Also Math/Trig Functions A-Z Function Reference

Description

Atan2(X,Y) calculates the arc tangent of the angle represented by the point with (x,y) coordinates X and Y.

Syntax

ATAN2(x,y)

x is the x coordinate
y is the y coordinate

Remarks

The arc tangent is the angle, determined by the point described by the coordinates. The result is an angle, in radians, from -Pi through Pi, excluding -Pi.

See Also

- ATAN
- ATANH
- Pi
- TAN

ATANH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the inverse hyperbolic tangent of a number.

Syntax

ATANH(X)

X is any number between -1 and 1 exclusive.

Remarks

The inverse hyperbolic tangent is the value whose hyperbolic tangent is x, i.e. ATANH(TANH(X)) = X.

See Also

ATAN
ATAN2
Pi
TAN

AVG Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the average of a list of numeric values.

Syntax

AVG(*num1* <*, num2,...numn*>)

num1, num2, numN are the numeric values for which you wish to find the mean. Up to MAXPARAMS values may be entered.

See Also

MAX
MIN
PRODUCT
SUM

FormulaBuilder 1.0 ™

YGB Software, Inc

Active Property

See Also

Applies To

TDSFilter

Declaration

```
Property Active : boolean;
```

Description

Determines whether the dataset will be filtered according to the expression set in the Formula or Lines properties.

Note if the Datasource property changes, the TDSFilter checks to ensure that the filter expression is still valid for the new dataset. If an error is detected, Active is automatically set to FALSE. NOTE: Active can be influenced by the LoadActivated property, to put the filter in active state at form startup time.

See Also

LoadActivated Property

AddBooleanConstant Method

See Also

Applies to

All FormulaBuilder Components

Declaration

`Procedure AddBooleanConstant(const name : TVarname;value : Boolean);`

Description

Registers a Boolean constant named *name* with the value *value* with the engine. Note that all constants are system global (visible to all expressions). The Status property will return EXPR_DUPLICATE_IDENT if the name *name* is already in use.

See Also

AddConstantPrim
AddDateConstant
AddNumericConstant
AddStringConstant

AddConstantPrim Method

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Procedure AddConstantPrim(const name : TVarName;Var Value : TValueRec);
```

Description

Registers a constant named *name* with the engine. The engine is responsible for freeing memory associated with *value*. See the definition of TValueRec in the Type Reference Section. Note that all constants in this version of FormulaBuilder are system global.

See Also

[AddBooleanConstant]
[AddDateConstant]
[AddNumericConstant]
[AddStringConstant]

AddDateConstant Method

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Procedure AddDateConstant(const name : TVarName;value : TFBDate);
```

Description

Registers a date constant named *name* with the value *value* with the engine. Note that all constants are system global (visible to all expressions). The Status property will return EXPR_DUPLICATE_IDENT if the name *name* is already in use. Note that TFBDate is a synonym for TDateTime.

See Also

[AddBooleanConstant]
[AddConstant]
[AddNumericConstant]
[AddStringConstant]

AddNumericConstant Method

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Procedure AddNumericConstant(const name : TVarname;value : double);
```

Description

Registers a numeric constant named *name* with the value *value* with the engine. Note that all constants are system global (visible to all expressions). The Status property will return EXPR_DUPLICATE_IDENT if the name *name* is already in use.

See Also

AddConstant
AddDateConstant
AddBooleanConstant
AddStringConstant

AddStringConstant Method

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Procedure  AddStringConstant(const name : TVarname;value : String);
```

Description

Registers a string constant named *name* with the value *value* with the engine. Note that all constants are system global (visible to all expressions). The Status property will return EXPR_DUPLICATE_IDENT if the name *name* is already in use.

See Also

AddBooleanConstant
AddConstant
AddDateConstant
AddNumericConstant

AddVariable Method

See Also Example

Applies to

All FormulaBuilder Components

Declaration

Procedure AddVariable(**const** *name* : string;*vtype* : byte);

Description

Adds a variable of type *vtype* to the engine for the expression object. *name* then becomes available for use in expressions. The initial value will be the NULL representation appropriate to the variable's type.

Note

Both the internally managed variable table and the OnFindVariable event methods are checked to see if the variable exists.

AddVariable Method Example

Example

This code assumes we have an initialized TExpression instance named Expression1

```
Procedure TForm1.AddVariables;
begin
  with Expression1 do
  begin
    { Note that the variables were added before the expression }
    { involving them was assigned to the Formula property }
    AddVariable('Name',vtSTRING);
    AddVariable('BirthDate',vtDATE);
    AddVariable('Married',vtBOOLEAN);
    AddVariable('Children',vtInteger);
    AddVariable('Salary',vtFLOAT);
    AddVariable('PIN',vtFLOAT);
    Formula := 'PIN := Length(Name) + DAY(BirthDate) -
               (Sqrt(Age) * Salary) * IIF(Married,Kids,0)';
  end;
end; { AddVariables }
```

See Also

ParseAddVariable

Adding An Expression Instance To A Form

We can a variable EXPRESSION1 of type TExpression to the form in a few ways. This discussion applies equally to TDSExpression , TDBExpression and TDSFilter

To Add a FormulaBuilder Component to a Form

To use the component version of TExpression, simply select the Expression icon from the 'FBuilder' page of the component palette and drop it onto your form. A `EXPRESSION1 : TExpression` is added to the Delphi-managed portion of the form's declaration. Delphi automatically adds FBCOMP to your USES statement.

Alternately, you may choose to use the component version non-visually. To do so, manually add FBCOMP to your USES statement, and add

```
Expression1 : TExpression;
```

to either the public or private part of the form's declaration.

For the data-aware components, make sure that FBDBCOMP appears in the USES statement of the unit using the expression class. For the RTTI-Aware class, make sure that FBRTCOMP appears in the USES statement.

Adding FormulaBuilder To a Visual Basic Project

To Add FormulaBuilder to a Visual Basic Project

Ensure that the FormulaBuilder DLL is in your search path

From an open Visual Basic project

Select File|Add File

From the file selection dialog box, select the header file **FBCALC.BAS**

Adding New Functions

FormulaBuilder makes it possible for us to define new functions that can be recognized by the parser. These functions have the same status as built-in functions. The means of registering new functions with the engine is the FBRegisterFunction function call.

Programmer defined functions are implemented using a callback procedure of the type TCBKExternalFunc All functions to be added to the FormulaBuilder Engine must adhere to this prototype and the implementation header must declared using the **export** directive.

We will demonstrate how to implement programmer -defined functions.

Example 1
Example 2

Adding Variables

Variables may be added by calls to AddVariable. The variable names may then be used in expressions. If the specified variable name exists, an EXPR_DUPLICATE_IDENT status is returned. Both the internally managed variable table and event methods are checked to see if the variable exists.

Adding Your Own Functions

FormulaBuilder provides over 110 functions in various categories to cover a wide range of problem areas. There are times, however, when specialized functions may be needed to fulfill a particular task. Also, if a particular expression is used frequently in an application, it may be more efficient to convert it into a parameterized function.

Advanced Variable Handling : Callback Example

```
unit Eiscbkfm;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  StdCtrls, Forms, DBCtrls, DB, DBGrids,
  SSheet,FBCOMP,FBDBCOMP,
   FBCALC,
  Grids,DBTables, ExtCtrls, Buttons;

type
  { since SetFieldCallbacks is a protected member of TDSExpression, we }
  { simply declare a dummy descendant to be able to get at the protected }
  { parts of TDSExpression }
  TNewExpression = Class(TDSExpression)
  end;

  TForm2 = class(TForm)
    DBGrid1: TDBGrid;
    DBNavigator: TDBNavigator;
    Panel1: TPanel;
    DataSource1: TDataSource;
    Panel2: TPanel;
    Table1: TTable;
    Panel3: TPanel;
    SSheetGrid: TStringGrid;
    GroupBox1: TGroupBox;
    ResultPanel: TPanel;
    FormulaEdit: TEdit;
    BitBtn1: TBitBtn;
    SpeedButton1: TSpeedButton;
    procedure FormCreate(Sender: TObject);
    procedure SSheetGridGetEditText(Sender: TObject; ACol, ARow: Longint;
      var Value: OpenString);
    procedure SSheetGridSetEditText(Sender: TObject; ACol, ARow: Longint;
      const Value: String);
    procedure FormDestroy(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
  private
    { private declarations }
    Sheet : TSpreadSheet;
  public
    { public declarations }
    Expression : TNewExpression;
  end;

var
  Form2: TForm2;

implementation
{$R *.DFM}

{
The syntax for "spreadsheet" cell access in [RnCn] where n is an integer,
for example :

       "[R1C1] * [R2C2] - [R5C2]"
}


 Function SheetFindVarCBK(vname        : pchar;
                         var vtype    : byte;
```

```
                          var vardata  : longint;
                          CBKData      : longint):integer; export;

 var   r,c      : word;
       theSheet : TSpreadSheet;
 begin
   result := EXPR_SUCCESS;
   if not ParseCellname(strpas(vname),r,c) then
   begin
      vtype := vtNONE;
      exit;
   end;
   theSheet := TSpreadSheet( CBKData ); { Cast CBKData back into spreadsheet }
   { check to see if r and c are within range. If not, return an error }
   if (r > MAXROWS) or (c > MAXCOLS) then
   begin
     Result := EXPR_RANGE_ERROR;
     Exit;
   end;
    { in our spreadsheet, all values are floats }
    vtype := vtFLOAT;
    { typecast vardata to a pointer to our actual value. This speeds }
    { up variable access when the value of the cell needs to be retrieved. }
    { see GetVariable function }
    vardata := longint( @theSheet.sheetData[r,c] );
 end; {}


function SheetGetVarCBK(vname     : pchar;
                        var Value : TValueRec;
                        vardata   : longint;
                        CBKData   : longint) :integer; export;

var theSheet : TSpreadSheet absolute CBKData;
begin
  result := EXPR_SUCCESS;
  { we could retrieve the value this way :

     ParseCellName(varname,r,c);
     value.vFloat := TheSheet.SheetData[r,c];

     but since we set vardata to point directly to the data, all we need to
     do is typecast and dereference the vardata parameter (see above). This
     is a bit faster, since we skip the ParseCellName function call.
     }
     value.vFloat := PDouble(VarData)^;
     { no errors occurred so we dont have to set errcode. Its value is
       EXPR_SUCCESS on entry }
end; { getVariable }



Function SheetSetVarCBK(vname     : pchar;
                        value     : TValueRec;
                        vardata   : longint;
                        CBKData   : longint):integer; export;
begin
  { we could set the value this way :

     ParseCellName(varname,r,c);
     TheSheet.SheetData[r,c] := value.vFloat;

     but since we set vardata to point directly to the data, all we need to
```

```
     do is typecast and dereference the vardata parameter (see above). This
     is a bit faster, since we skip the ParseCellName function call.
     }
     PDouble(VarData)^ := value.vFloat;
     { no errors occurred so we dont have to set errcode. Its value is
       EXPR_SUCCESS on entry }
end; { setVariable }




procedure TForm2.FormCreate(Sender: TObject);
var r, c   : integer;
    tmpstr : String[15];
begin
  Table1.Open;
  Sheet      := TSpreadSheet.Create;
  Expression := TNewExpression.Create;
{ Note the last parameter passed to SetFieldCallbacks. This is the value that }
{ is passed to the CBKData parameter of the callback functions. We use this }
{ fact to pass our instance of the spreadsheet to the callback functions }
  Expression.SetFieldCallbacks(SheetFindVarCBK,
                               SheetGetVarCBK,
                               SheetSetVarCBK,
                               longint(Sheet));
  Expression.Dataset := Table1;
  for r := 0 to MAXROWS do
  for c := 0 to MAXCOLS do
  begin
    if (r + c = 0) then continue;
    if (r = 0) then
    begin
      tmpStr := 'C'+IntToStr(c);
      SSheetGrid.Cells[c,r] := tmpstr;
    end
   else
    if (c = 0) then
    begin
      tmpStr := 'R'+IntToStr(r);
      SSheetGrid.Cells[c,r] := tmpstr;
    end
   else
    begin
       tmpstr := FloatToStrF(Sheet.SheetData[r,c],ffCurrency,10,2);
       SSheetGrid.Cells[c,r] := tmpstr;
    end;
  end;
end;


procedure TForm2.SSheetGridGetEditText(Sender: TObject; ACol,
  ARow: Longint; var Value: OpenString);
begin
   Value := FloatToStrF(Sheet.SheetData[ARow,Acol],ffCurrency,10,2);
end;

procedure TForm2.SSheetGridSetEditText(Sender: TObject; ACol,
  ARow: Longint; const Value: String);
var temp : double;
begin
  Try
    Sheet.SheetData[ARow,ACol] := StrToFloat(value);
```

```
  except
    {}
  end;
end;

procedure TForm2.FormDestroy(Sender: TObject);
begin
  Expression.Free;
end;

procedure TForm2.SpeedButton1Click(Sender: TObject);
var stringExpr : String;
begin
  StringExpr := FormulaEdit.Text;
  if StringExpr <> '' then
  begin
    Expression.Formula := StringExpr;
    if Expression.Status <> EXPR_SUCCESS then
    begin
      MessageBeep( MB_ICONHAND );
      ResultPanel.Caption := Expression.StatusText;
    end
     else
       ResultPanel.Caption := Expression.AsString;
  end;
end;

end.
```

Advanced Variable Handling Examples

Three example programs have been provided to demonstrate the issues discussed here. They implement the simple Stock Market EIS scenario we described above :

The EISBASIC.DPR project demonstrates the possible problems we may encounter if we use default variable processing

The EISCBK.DPR project demonstrates an improvement using callbacks set at the DLL call level.

The EIS.DPR project shows how to use programmer defined variable processing via the events of the TExpression class.

The syntax for "spreadsheet" cell access in [RnCn] where n is an integer, for example :

"[R1C1] * [R2C2] - [R5C2]"

except in EISBASIC.DPR where the square brackets are not used.

Note that the last two projects use the TDSExpression class, which itself uses external variable handling to treat fields of a BDE dataset as variables.

Advanced Variable/Field Handling

The standard methods of handling variables work well in a large number of case (expressions with a small number of variables), but may be inappropriate or inefficient in other instances. Imagine this scenario :

We are designing an EIS project which permits calculations based on Stock Market data in a database as well as a spreadsheet. Suppose also that the number of variables in such a formula are large (for the sake of our examples,well use just a few, but imagine that there are many).

The Usual Methods
Using The Variable/Field Callback Functions

The Variable/Field Handling Events

The TExpression Events fully encapsulate the FormulaBuilder Callbacks in the Onxxx event handlers. Handling variables and fields using these events involves two steps

For programmer defined variable handling, assign methods to the OnFindVariable, OnGetVariable and optionally the OnSetVariable properties.

Set the UseEvents boolean property of the TExpression instance to TRUE. This tells FormulaBuilder that you will implement variable handling in your own code, in addition to the default behavior. Since a field is a variable by another name, no further action is necessary to handle fields in you code.

Examples

Alphabetical Function Reference

See Also

This topic provides an alphabetical reference for the FormulaBuilder functions. Refer to Understanding Functions, for additional information about using these functions. The built-in functions listed in the following sections are:

ABS	FIND	PROPER
ACOS	FIRST	PV
ACOSH	FLOOR	PVAL
ACOT	FRAC	RADIANS
ACOTH	FV	RAND
ACSC	FVAL	RATE
ACSCH	HOUR	REPLACE
ASC	IIF	REPLICATE
ASEC	INSERT	ROUND
ASECH	INT	RTRIM
ASIN	IPAYMT	SEC
ASINH	IRATE	SECH
ATAN	IRR	SECOND
ATAN2	ISEVEN	SGN
ATANH	ISODD	SIN
AVG	LAST	SINH
CEILING	LENGTH	SLN
CHAR	LN	SOUNDEX
CHOOSE	LOG	SOUNDALIKE
CLEAN	LOWER	SQR
COS	LTRIM	SQRT
COSH	MAX	STR
COT	MAXSTR	SUM
COTH	MID	SYD
CSC	MIN	TAN
CSCH	MINSTR	TANH
CTERM	MINUTE	TERM
DATE	MONTH	TIME
DATEDIFF	MONTHNAME	TIMENOW
DATETOSTR	NOW	TIMETOSTR
DATEVALUE	NPER	TIMEVALUE
DAY	NPV	TODAY
DAYNAME	PADCENTER	TRIM
DB	PADLEFT	UPPER
DDB	PADRIGHT	VAL
DEGREES	PAYMT	WEEKDAY
EXP	PMT	WORDCOUNT
EXTRACT	PPAYMT	YEAR
FACT	PRODUCT	

See Also

Date/Time Functions
Financial Functions
Mathematical/Trig Functions
Miscellaneous Functions
String Functions

Arithmetic Operators

Operator Precedence

FormulaBuilder supports the standard arithmetic operators

Binary Arithmetic Operators

Operator(s)	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
div	Performs integer division on the operands.
mod	Performs modulo division.
^, **	Exponentiation (raise a number to a power).
and, &	Performs a bitwise AND of the operands. Floating point values are truncated before the operation is performed.
or, |	Performs a bitwise OR of the operands. Floating point values are truncated before the operation is performed.
not	Performs a unary bitwise negation of an operand. Floating point values are truncated before the operation is performed.
xor	Performs a bitwise exclusive OR of the operands. Floating point values are truncated to integers before the operation is performed.

Unary Arithmetic Operators

Operator	Description
+	Unary plus (sign identity)
-	Unary minus (sign negation)

AsBoolean Property (TInstanceProperty)

See Also

Applies To

TInstanceProperty

Declaration

```
Property AsBoolean : Boolean;
```

Description

Reads and sets the instance property as a boolean. If the underlying property is not of type boolean, a property value error is raised.

AsBoolean Property

See Also

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Property AsBoolean : boolean;
```

Description

Read-only. Evaluates the expression, returning its boolean result. An EXPR_TYPEMISMATCH error will be generated if the expression type is not vtBOOLEAN. The expression result type can be predetermined by using the ReturnType property. To get the result as a string, use the AsString property.

See Also

AsString
ReturnType

AsChar Property (TInstanceProperty)

See Also

Applies To

TInstanceProperty

Declaration

```
Property AsChar : Char;
```

Description

Reads and sets the instance property as a char. If the underlying property is not of type Char, a property value error is raised.

AsDate Property

See Also

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Property AsDate : TDateTime;
```

Description

Evaluates the expression, returning its date result. An EXPR_TYPE_MISMATCH error will be generated if the expression type is not vtDATE. The expression result type can be predetermined by using the ReturnType property. To get the result as a string, use the AsString property.

See Also

AsString
ReturnType

AsFloat (TInstanceProperty)

See Also

Applies To

TInstanceProperty

Declaration

```
Property AsFloat : Extended;
```

Description

Reads and sets the instance property as a Floating point value. If the underlying property is not of type float (I.e. the Kind property is other than tkFloat), a property value error is raised.

AsFloat Property

See Also

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Property AsFloat : Double;
```

Description

Evaluates the expression, returning its real-type result. A EXPR_TYPE_MISMATCH error will be generated if the expression type is not vtFLOAT.or vtINTEGER. The expression result type can be predetermined by using the ReturnType property. To get the result as a string, use the AsString property.

AsInteger (TInstanceProperty)

See Also

Applies To

TInstanceProperty

Declaration

```
Property AsInteger : Longint;
```

Description

Reads and sets the instance property as an integer. Use this property for setting the values of properties whose Kind property reads tkSet, tkEnumeration or tkInteger. If the underlying property is not of one of these types, a property value error is raised.

AsInteger Property

See Also

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Property AsInteger : Longint;
```

Description

Evaluates the expression, returning its *longint* result. A EXPR_TYPE_MISMATCH error will be generated if the expression type is not vtINTEGER. or vtFLOAT. The expression result type can be pre-determined by using the ReturnType property. To get the result as a string, use the AsString property.

AsMethod (TInstanceProperty)

Applies To

TInstanceProperty

Declaration

```
Property AsMethod : TMethod;
```

Description

Reads and sets the instance property as a TMethod. If the underlying property is not a method type, a property value error is raised.

AsObject (TInstanceProperty)

See Also

Applies To

TInstanceProperty

Declaration

```
Property AsObject : TObject;
```

Description

Reads and sets the instance property as an Object instance. If the underlying property is not of type TObject (or a descendant), a property value error is raised. The Kind property may be checked before hand to ensure that it is *tkClass.*

AsString Property

See Also

Applies To

TInstanceProperty

Declaration

```
Property AsString : string;
```

Description

Reads and sets the value of a property of an object instance as a string, regardless of the property type. The string returned from (and expected for) this property is in standard Object Pascal format for a constant of the property's type. For example, the value of the style of a font may be returned as

```
'[fsBold,fsItalic]'
```

To set the style to underline,

```
      fontProp.AsString := '[fsUnderLine]';
```

Note that any enumerated type identifier which appears in a published property (either in an enumerted type or a set type) may be used.

AsString Property

See Also

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Property AsString : string;
```

Description

This readonly property evaluates the expression and returns the string equivalent of the result, regardless of the Returntype.

See Also

AsBoolean	AsInteger
AsDate	ReturnType
AsFloat	StringResult

Assigning The Text To Be Evaluated

Before we can use our expression instance, we need to tell it which text expression we wish to have evaluated. TExpression provides three properties for setting (or querying) the text form of an expression.

The Formula Property Example

Using the Formula property, we have access to the text expression as a Delphi string.

The StrFormula Property Example

For longer strings that exceed the 255 character limit, we can use the StrFormula property.

The Lines Property Example

Even more convenient for memo users is the Lines property. Using this property we have access to the text expression as an indexed set of lines.

NOTE Assigning text to a TExpression does NOT cause the expression to be evaluated. See the topic Getting Expression Results for details on retrieving the results of an expression.

AutoRefresh Property
See Also
Applies To
TDSFilter

Declaration
```
Property AutoRefresh : Boolean
```

Description
Determines whether the Dataset attached to the Datasource property will be automatically refreshed when the Active property changes. If False, you must programatically call Refresh.

See Also

Active Property
Refresh Method

BDE

BDE is an acronym for the Borland Database Engine (also known as **IDAPI**). It is the database engine shipped with Delphi

Basic Concepts

FormulaBuilder makes it easy for you to incorporate run-time expression evaluation into your applications. Expressions are combinations of operators and operands that evaluate to a single value.:

Operators

The operators are divided into the following categories
Arithmetic Operators
Relational Operators
Logical Operators
String Operators
Boolean Operators

FormulaBuilder also supports the assignment operator ":=" (without the quotes). Assignments of the form :
Variable := *Expression*
Field := *Expression*

may be made to variables that have been added to the engine. The assignment expression sets the value of the variable as well as returning the value of the expression. Note that only one assignment is permitted per expression.

Precedence

The meaning of an expression is affected by the Precedence of the operators involved in the expression. Normal precedence order may be overridden by using parentheses.

Operands

Operands are of the following types :
Constants
Variables
Fields
Functions

The Evaluation Process

Click here for a description of the FormulaBuilder evaluation process.

Boolean Constants
The Boolean constants are "TRUE" and "FALSE" (entered without the quotes). Case is not important.

Boolean Operators

The boolean operators take boolean operands and return a boolean. All except **not** are binary operators.

Operator	Description
not	negation
and	logical AND. Returns TRUE if both operands are TRUE.
or	logical OR. Returns TRUE if either operand evaluates to TRUE
xor	logical exclusive or. Returns TRUE if one or the other, but not both operands are TRUE.

Built-In Function Reference

The current release of FormulaBuilder gives the end-user access to over 100 functions in the following areas:

Click here to see an Alphabetical Function Reference

See the chapter "Extending FormulaBuilder" for information on registering programmer-defined functions with the FormulaBuilder engine.

C/C++ External Function Example

Suppose we wanted run time access to a function "myfunc()" . For the sake of our discussion, our function "myfunc()" will include parameters of each type supported by the FormulaBuilder engine. The declaration of our function, in "C" would be as follows :

```
char *myfunc(long l,BOOLEAN b,double d,LPSTR s,TFBDate dt);
```

We could use this in a FormulaBuilder expression as follows :

```
#define MYEXPR  "'myfunc() returns ' + myfunc(12345, true, 10.0245,\
              'myfunc string',today() )"
HEXPR myHandle;
char   result[90];

myHandle = FBInitExpression();
FBSetExpression(myHandle,MYEXPR);
FBEvaluate(myHandle,result,sizeof(result)-1)
```

Implementing The Callback

In order to make myfunc() available, we have to create an exportable callback function with the prototype TCBKExternalFunc. Note that the **CALLBACK** macro expands to FAR PASCAL (see windows.h). Since the callback needs to be exported from the DLL, we need to use the _export directive. Our implementation of the function follows:

```
/* Function with all type parameters */
/* syntax : char *myfunc(long l,BOOLEAN b,double d,LPSTR s,date dt)
void CALLBACK _export myfunc( BYTE           paramcount,
                              LPPARAMLIST    params,
                              LPVALUEREC     retvalue,
                              LPINT          errcode,
                              LONG           lCBKData)
{
    char     result[120];
    char     datestr[20];
    long     intval;
    BOOL     boolval;
    double   floatval;
    char     strval[81];
    TFBDate  dateval;

    intval      = params->[0].vInteger;
    boolval  = params->[1].vBool;
    floatval   = params->[2].vFloat;
    dateval   = params->[4].vDate;

    FBStrncpy(strval,params->[3].vpString,80);
    FBDateToLpz(dateval,datestr,20);
    sprintf(result," int : %ld  bool : %d float : %f str : %s date : %s ",
           intval,boolval,floatval,strval,datestr);

    retvalue->vpString = FBCreateString(result);

    *errcode = EXPR_SUCCESS; /* not really necessary, since this is  its value on entry
*/
};
```

Registering The Function

Now that our callback function is written, we need simply to register the function with the FormulaBuilder parser. We do so by means of the FBRegisterFunction call.

```
int myFnId = FBRegisterFunction("myfunc",vtSTRING,"ibfsd",5,myfunc);
```

The first parameter tells FormulaBuilder the name of your function, the second its type (see the vtXXX constants). The third parameter describes the parameters expected for the function (integer, boolean, float, string and date respectively). FormulaBuilder guarantees that the elements of the *params* parameter passed to *myfunc()* will be exactly of the type and in the order listed. The next parameter instructs the parser to expect a minimum of 5 parameters. This value could have been any value from 0 to the length of the previous parameter. The *paramcount* parameter of the callback routine, upon entry, contains the number of parameters the user entered. The final parameter, of course, is a pointer to the function which implements *myfunc()*.

FBRegisterFunction returns EXPR_INVALID_FUNCTION if the call is unsuccessful, otherwise it returns a positive integer > 400 which uniquely identifies your function. You may use the return value from the registration call to unregister the function.

Thats It ! Youve successfully added a function to FormulaBuilder. *myfunc()* will be treated like any of FormulaBuilder’s other functions. As you can see, practically any function can be added, including wrapper functions for the Windows API.

CEILING Function

See Also Math/Trig Functions A-Z Function Reference

Description

Rounds a number up to the nearest whole number

Syntax

CEILING(*x*)

x is any number

See Also

FLOOR
FRAC
INT
ROUND

CHAR Function

See Also String Functions A-Z Function Reference

Description

Returns the ANSI character corresponding to a number.

Syntax

CHAR(*number*)

Remarks

Number is a number between 1 and 255. For example, CHR(32) returns the space character.

See Also

CODE

CHOOSE Function

See Also String Functions A-Z Function Reference

Description

Uses an numeric expression index to select a value from a list.

Syntax

CHOOSE(*choice,value1, value2,...valueN*)

choice is the number which is used as the index. If choice is 1, CHOOSE returns *value1*. If choice is 2, *value2* is returned, and so on.
value*1,value2, ...valueN* are the values from which the choice is made. Up to MAXPARAMS values of any type may be included in the list.

If choice is less than 1 or greater than the number of elements in the list, an EXPR_RANGE_ERROR is returned.

Example

CHOOSE(2,{10/10/95},"hello",TRUE,Cos(pi * 2)) evaluates to "hello"

CHOOSE(3,"Mon","Tue","Wed","Thu","Fri","Sat","Sun") equals "Wed"

See Also

IIF

CLEAN Function

[See Also](#) [String Functions](#) [A-Z Function Reference](#)

Description

Cleans a string of all unprintable characters.

Syntax

CLEAN(*st*)

St is any string or string expression.

See Also

LTRIM
RTRIM
TRIM

CODE Function

[See Also] [String Functions] [A-Z Function Reference]

Description

Returns a numeric (ANSI) code for the first character in a text string

Syntax

CODE(*text*)

Text is the string for which you want to determine the code.

See Also

CHAR

COS Function

See Also Math/Trig Functions A-Z Function Reference

Description

Calculates the cosine of angle x, expressed in radians, returning a value between -1 and 1.

Syntax

COS(x)

X is the angle. To convert an angle expressed in degrees to radians, use the RADIANS function.

See Also

ACOS
COSH
Pi

COSH Function

See Also   Math/Trig Functions   A-Z Function Reference

Description

Returns the hyperbolic cosine of its argument.

Syntax

COSH(*x*)

Remarks

x is any number floating point or integer value. The value returned is in radians. Use the DEGREES function if you would like to convert the answer to degrees.

See Also

ACOSH
SINH
TANH

COT Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the cotangent of an angle in radians.

Syntax

COT(x)

X is the angle in radians.

Remarks

If you wish to convert a value expressed in degrees to radians, use the RADIANS function.
COT(X) is equivalent to 1/TAN(X)

See Also

ACOT
ACOTH

COTH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the hyperbolic cotangent of an angle.

Syntax

COTH(x)

X is the angle in radians. If your *x* value is in degrees, use the RADIANS function to convert it to radians before passing it to this function

See Also

ACOTH

COT

CSC Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the cosecant of an angle.

Syntax

CSC(x)

X is the angle in radians. To convert an angle expressed in degrees to degrees, use the RADIANS function.

See Also

- ACSC
- ACSCH
- CSCH

CSCH Function

See Also   Math/Trig Functions   A-Z Function Reference

Description

Returns the hyperbolic cosecant of an angle.

Syntax

CSCH(x)

X is the angle in radians.

Remarks

If your wish to convert a value expressed in degrees to radians, use the RADIANS function.

See Also

ACSC
ACSCH
CSC

CTERM Function

See Also | Financial Functions | A-Z Function Reference

Description

Calculates the number of compounding periods it takes for the present value of an investment to grow to a future value at a fixed rate of interest per period.

Syntax

CTERM(*rate,fv,pv,nper*)

Parameter	**Description**
rate	the periodic rate of interest, greater than -1
fv	future value. The value the investment is expected to attain after the last payment.
pv	present value. The current value of the investment
nper	the number of payment periods for the investment

See Also

RATE

CheckLoadFB Function

See Also

Unit

FBCALC

Declaration

```
Function CheckLoadFB : boolean;
```

Description

CheckLoadFB checks to see if the FormulaBuilder DLL is loaded. If it is, the function returns TRUE, otherwise it attempts to load the DLL, and returns true if successful.

See Also

FBLoaded
FreeFBuilder
InitFBuilder

ClassAssignmentCompatible Function

Unit

FB_Rtti

Declaration

```
Function ClassAssignmentCompatible(Class1 , Class2 : TObject):boolean;
```

Description

Returns true is class2 can be assigned to class1.

Clear Method

Applies to
All FormulaBuilder Components

Declaration
```
Procedure Clear;
```

Description
Clears all internal variables. Returns the TExpression to the state it would be in after a Create call.

Clearing An Expression

The Clear method sets the text and tokenized versions of an expression to NULL, and returns an expression instance to the state it would be in after a call to the Create constructor.

Note It is not necessary to clear an expression before changing the expression text. For instance, there is no need for a Clear in the following code :

```
Expression.Formula := 'Sin(X^2) * Abs(X * COS(Y))';
Panel1.Caption := Expression.AsString;
Expression.Formula := 'IIF( WeekDay(Today()) = 2, TRUE, FALSE )';
Panel2.Caption := Expression.AsString;
```

Constant Handling Functions

This section details the FormulaBuilder functions relating to user-defined constants. Note that constants added with the FBAddxxx functions are DLL global (visible to all calling apps), but unlike user-defined functions, are maintained internally by FormulaBuilder. It is not strictly necessary, except as a matter of good programming practice, for a task to remove the constants it added. The most frequent error returned from these functions is EXPR_DUPLICATE_IDENT, indicating that another identifier (variable, constant, function or operator) was registered with the same name.

Adding Constants

FBAddBooleanConstant
FBAddConstantPrim
FBAddDateConstant
FBAddNumericConstant
FBAddStringConstant
FBParseAddConstant

Getting Constant Values

FBGetConstAsString
FBGetConstantPrim

Disposing Of Constants

FBFreeConstant
FBFreeConstants

CONSTANTS

Constants (also referred to as Literals) are values that do not change. Constants of the following type are permitted in expressions

Numeric
Strings
Boolean
Date/Time

Constants Property

Applies to
All FormulaBuilder Components

Declaration
```
Property Constants[Const cname : TvarName]:TValueRec;
```

Description
This array property provides read/write access to the constant values by a name. If you attempt to assign a value to a constant that does not exist, a constant with name *cname* is created and given the value of the right side of the expression. If you attempt to modify an existing *constant*, you will get an error EXPR_DUPLICATE_IDENT.

Note
After assignment, the expression instance owns the memory of the TValueRec assigned to this property, so DO NOT dispose of it with FBFreeValue .

Example
```
      PiValue := RotationExpr.Constants['Pi'];
```

FormulaBuilder 1.0 ™ YGB Software, Inc.

An Advanced Expression Evaluation Engine.

About Copyright Order Form

FormulaBuilder 1.0 ™ YGB Software, Inc.

An Advanced Expression Evaluation Engine.

FormulaBuilder is published by

YGB Software, Inc
161 Pearl Street
Paterson, NJ, 07501
USA

Trademarks

Create Constructor

See Also

Applies To

TInstanceProperty

Declaration

```
Constructor TInstanceProperty.Create;
```

Description

Creates an instance of a TInstanceProperty. Note that the Propname and Instance properties must be set for the object to be useable.

Create Constructor

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Constructor Create; Virtual;
```

Description

Creates an instance of the expression evaluator class.

See Also

Destroy Destructor

CreateFromPath Constructor

See Also

Applies To

TInstanceProperty

Declaration

```
Constructor CreateFromPath(root : TObject; PropPath : String);
```

Description

Creates a TInstanceProperty object based on the property path PropPath. Root is the starting point of PropPath. The Instance property is automatically set to the actual object instance which contains the property.

Example

For instance, if Root is set to an instance of a TForm, valid property paths would be

```
'Caption'
'Font.Name'
```

Note also that you also have (recursive) access to the properties of named components contained in the Components array of components. For instance, given the same form which contains a TDataSource named CustomerSource, we could use the following property path:

```
'CustomerSource.Dataset.Tablename'
```

If the Root property were set to *Application*, and our form were named *CustomerForm*, we would write the properties as follows :

```
'CustomerForm.Caption'
'CustomerForm.Font.Name'
'CustomerForm.CustomerSource.Dataset.Tablename'
```

See Also

Create
CreateFull
CreateFromSearch

CreateFromSearch Constructor

See Also

Applies To

TInstanceProperty

Declaration

```
Constructor CreateFromSearch(root : TObject;Const Propname : string;kinds :
TTypeKinds);
```

Description

Searches recursively downward from root, looking for the first instance of a published property with the name Propname, of a type in the set kinds. If found, it creates a TInstanceProperty object to encapsulate the located property. The Instance property is automatically set to the object instance in which the property was found.

Remarks

The search involves not only properties of root proper, but recursively all named components contained by root (in the Components array property).

The *Kinds* parameter, which limits the search to specific types of properties, is of type *TTypeKinds*, which is defined in TYPINFO.INT as follows :

```
type
  TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
             tkString, tkSet, tkClass, tkMethod);
  TTypeKinds = set of TTypeKind;
```

See Also

- Create
- CreateFull
- CreateFromPath
- Instance
- Propname

CreateFull Constructor

see also

Applies To

TInstanceProperty

Declaration

```
Constructor CreateFull(anInstance : TObject;APropInfo : PPropInfo);
```

Description

Creates an instance of a TInstanceProperty and initializes it with both the Instance value and a pointer to the RTTI record which describes the instance property the object is to encapsulate.

For the declaration of PPropInfo, check the TYPINFO.INT file in the \DELPHI\DOC directory.

see also

- Create
- CreateFromPath
- CreateFromSearch
- Instance
- Propname

DATE Function

See Also Date/Time Functions A-Z Function Reference

Description

The Date() function returns a date serial number from the values specified as the Year, Month, and Day parameters.

Syntax

Date(*year,month,day*)

The *year* must be between 1 and 9999.
Valid *Month* values are 1 through 12.
Valid *Day* values are 1 through 28, 29, 30, or 31, depending on the *Month* value. For example, the possible *Day* values for month 2 (February) are 1 through 28 or 1 through 29, depending on whether or not the Year value specifies a leap year.

Remarks

If the specified values are not within range, an EXPR_CONVERT_ERROR error is raised. The resulting value is one plus the number of days between 1/1/0001 and the given date.

See Also

DATEVALUE	NOW
DAY	TODAY
MONTH	YEAR

DATEDIFF Function

Date/Time Functions A-Z Function Reference

Description
Returns the number of days between two dates.

Syntax
DATEDIFF(*date1*,*date2*)

date1 and *date2* are dateserial numbers. *date2* is presumed to be the later date.

DATETOSTR Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the date string representation of a date/time serial number.

Syntax

DATETOSTR(*date_serial*)

date_serial. is a text string which contains a valid date.

Remarks

This function performs the reverse of the DateValue function.

See Also

DATEVALUE
STR
TIMETOSTR

DATEVALUE Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the date serial number equivalent of the string expression dateStr.

Syntax

DATEVALUE(*date_text*)

date_text. is a text string which contains a valid date.

Remarks

Valid values for *date_text* are determined by the International settings in the Windows Control Panel.

See Also

- DATETOSTR
- NOW
- TIMEVALUE
- TODAY

DAY Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns an integer (1 - 31) representing the day component of a date serial number.

Syntax

DAY(*date_serial*)

date_serial is the date value.

See Also

- DAYNAME
- MONTH
- WEEKDAY
- YEAR

DAYNAME Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the name (for example "Monday", "Friday") of the day on which a date falls.

Syntax

DAYNAME(*datenumber*)

datenumber is the date serial number for which you wish to find the dayname.

See Also

DAY

WEEKDAY

DB Function

See Also   Financial Functions   A-Z Function Reference

Description

Calculates the depreciation allowance for an asset using the fixed-declining balance method.

Syntax

DB(*Cost,Salvage,Life,Period*)

Parameter	**Description**
Cost	the initial amount paid for the asset. Can be any positive value or 0.
Salvage	value of the asset at the end of its life. Can be 0 or any positive value.
Period	the period , >= 1, for which depreciation is to be calculated
Life	number of periods in the life of the asset.

Remarks

Life and *Period* must be expressed the same units.

See Also

DDB
SYD
SLN

DDB Function

See Also Financial Functions A-Z Function Reference

Description

Calculates the accelerated depreciation expense for an asset using the Double Declining Balance Method.

Syntax

DDB(*Cost,Salvage,Life,Period*)

Parameter	Description
Cost	the amount paid for the asset
Salvage	the amount expected for the asset at the end of the asset's useful life
Life	the expected useful life of the asset
Period	the time period for which the depreciation expense is to be calculated

Remarks

All arguments are assumed to be numeric values.The following relationships must hold :

Life >= *Period* >= 1
Cost >= *Salvage* >= 0

See Also

DB
SLN
SYD

DEGREES Function

See Also Math/Trig Functions A-Z Function Reference

Description

Converts an angle in radians to its equivalent in degrees.

Syntax

DEGREES(*x*)

Remarks

X is any number floating point or integer value. The resulting value is *X* * (180/PI)

DLL Function Reference

A-Z Function Call Reference

This section documents the functions exported by the FormulaBuilder DLL according to functional category. Most functions return one of the EXPR_XXX constants. For an explanation of these, see the error code reference in the appendix.

Expression Initialization And Disposal
Expression Evaluation
Variable Handling
Constant Handling
Function Handling
Utility Routines
Error Reporting

Note

In addition to the above topics, Delphi/Pascal users should read the topic Preliminary Issues For Delphi Users before attempting to call any DLL functions.

Data-Aware Classes : Setting The Data Source

Before assigning expression text to the data-aware components, we must specify where the expression will be deriving its variable data from.

TDSExpression

TDSExpression has a Dataset property which specifies the TTable or TQuery whose fields will act as variables. After creating an instance of TDSExpression, set the Database property to an open dataset before assigning a value to the Formula, Lines or StrFormula properties. This may be done via the Object Inspector, or programmatically as follows : :

Example

```
Procedure TForm1.FormCreate(Sender : TObject);
begin
    LineItemsTable.Open;
    exprCost := TDSExpression.Create(self);
    exprCost.Dataset  := LineItemsTable;
    exprCost.Formula := 'QUANTITY * UNIT_PRICE';
end;
```

Notice that the fields of LineItemsTable are now treated as variables.

TDSFilter

TDSFilter has a Datasource property which specifies the BDE datasource whose dataset will be filtered. After creating an instance of TDSFilter, set the Datasource property to a datasource before assigning a value to the Formula, Lines or StrFormula properties. This may be done via the Object Inspector, or programmatically as follows : :

Example

```
Procedure TForm1.FormCreate(Sender : TObject);
begin
    InvoiceTable.Open;
    Datasource1.Dataset := InvoiceTable;
    Table1Filter.Datasource := Datasource1;
    Table1Filter.Formula   := '(TOTAL * (1 + Tax_Rate)) > 3500';
end;
```

Invoicetable will now be filtered such that only invoices whose post-tax amount is greater than $3500 will be visible.

TDBExpression

TDBExpression allows us to have expressions based on any open Dataset in a Database. The Database property specifies which Database to use.

The field syntax is

'[' ***TableName*** '->' ***FieldName*** ']'

Example

Database Property

Applies To
TDBExpression

Declaration
```
Property Database : TDatabase;
```

Description
Read/write. Database specifies the database (TDatabase) component associated with the TDBExpression instance. Once set, the TDBExpression will have access to all the fields defined on open datasets in Database. **Note** - changing the Database property causes an automatic trigger of the Reparse method.

Example
```
TaxExpression.Database := TaxTable.Database;
```

Dataset Property

Applies To
TDSExpression

Declaration
```
Property Dataset : TDataset;
```

Description
Read/write. Dataset specifies the dataset (TTable or TQuery) component associated with the TDSExpression instance. Once set, the TDSExpression will have access to all the fields defined on the dataset. **Note** - changing the Dataset property causes an automatic trigger of the Reparse method.

Example
```
TaxExpression.Dataset := Form1.Table1;
```

Datasource Property

Applies To
TDSFilter

Declaration
```
Property Datasource : TDatasource
```

Description
Read/Write. Determines the Datasource whose dataset will be filtered. Note that the TDSFilter may fail to run on SQL-servers, as it   was designed for LOCAL databases

Date/Time Functions

For calculation purposes, date/time values (*serial numbers*) are stored internally as a double, where the integer portion represents the number of days that have passed since 1/1/0001. Time is stored as the floating-point part of the value. The floating-point part represents the fractional part of the day, ranging from 0.0 to 0.9999999, representing the times from 0:00:00 (12 midnight) to 23:59:59 (11:59:59 P.M.) For example, 0.5 represents noon, 0.75 represents 6:00 P.M.

DATE	HOUR	TIME
DATEDIFF	MINUTE	TIMEVALUE
DATETOSTR	MONTH	TIMETOSTR
DATEVALUE	MONTHNAME	TODAY
DAY	NOW	WEEKDAY
DAYNAME	SECOND	YEAR

Date/Time Serial Numbers

Date/Time serial numbers are used to represent a date and/or time. Internally they are stored as floating point values (double) where the integer portion represents the date (the number of days elapsed since 1/1/0001). The floating point portion represents the fractional portion of the day. For example 0.5 represents noon (12:00 PM), 0.75 represents 6 PM, and 0 represents midnight.

Date/Time Constants

Date Constants

The format for a date constant is {mm/dd/yy}, with the curly braces being delimiters. The format of the date entered depends on the Shortdate format settings established in international section of the Windows Control panel. Note that if a year ranging from 0 to 99 is entered, it is assumed to the year starting at 1900.

Numeric constants may be added to a date, returning a date

Example:

{10/10/95} + 365	= {10/10/96}
DAYNAME(20 + Today())	returns the name of the day 20 days from today

Time Constants

The format for a time constant is {hh:mm:ss}. Specifying AM or PM is optional, as are the seconds. Military (24 hour) time should be used if the AM/PM designator is ommitted.

Examples

HH:MM:SS AM/PM e.g. 10:12:19 PM
HH:MM AM/PM e.g. 12:13 AM
The Long International Time Format chosen as a system default, one of which {rw} is
HH:MM:SS eg 15:45:30

The Short International Time Format chosen as a system default, one of which is HH:MM eg. 10:35

Date/Time Constants

The date and time may be combined as follows :

{mm/dd/yy hh:mm:ss}

Example

{10/12/95 10:25:30 am}
{10/10/1885 21:30}

Please refer to the *Date Functions* section of the Function Reference for additional details of the date type.

See Also

RADIANS Function

Callback Error Reporting Example

Suppose we want to limit the range of values the user can enter as arguments to the ROMAN function from Example 1. The ROMAN function, for example, does not handle negative numbers. Also remember from our previous discussion that FormulaBuilder does automatic type conversions between compatible types to ensure that the correct parameter type is passed to a function. This would allow the user of the ROMAN function to type 'ROMAN(15.43)', which would be evaluated as ''ROMAN(15)'. We will disallow the use of floating point numbers in our function .

```
{ RomanFunc with range checking }
Procedure RomanProc( paramcount     : byte;
                     const params   : TActParamList;
                     var   retvalue : TValueRec;
                     var   errcode  : integer;
                           Exprdata : longint); export;
var number : longint;
    roman  : string[40];
begin
  number := params[0].vFloat;
  { complain if there is a fractional part }
  if (Frac(params[0].vFloat) - 1E6) > 0 then
     Errcode := EXPR_TYPE_MISMATCH
 else
  if number < 0 then
     errcode := EXPR_DOMAIN_ERROR; { param is out of domain of function  }
 else                            { definition   }
  begin
    roman  := Romanize(number)+#0;
    retvalue.vpString := FBCreateString(@Roman[1]);
  end;
end;
```

If a negative or floating point value were passed into the function (for example `Expression1.formula = 'Roman(-1)'`) then evaluation of the expression would terminate with the Status Property of the TExpression being set to EXPR_DOMAIN_ERROR.

We will have to modify our registration slightly to change the single parameter to a float rather than an integer :

```
RomanFnId := FBRegisterFunction('ROMAN',vtSTRING,'f',1,RomanProc);
```

FormulaBuilder Delphi Component Reference

The FormulaBuilder package includes Delphi components that simplify the use of the FormulaBuilder DLL engine,including two components to handle expressions based on BDE Datasets.

"Standard" Components

TExpression

TRTTIExpression

Data-Aware Components

TDSExpression

TDBExpression

TDSFilter

For the most part the Tasks involved in using these classes is common to all of them.

Error Handling

You may select how errors are handled in TExpression and TDBExpression by setting the UseExceptions property. If *UseExceptions* is set to false, errors are returned in the *Status* property, otherwise an exception of type EFBError is raised.

EFBError
EFBDBError

Demoware Version
The Demoware Version of FormulaBuilder displays a registration reminder for each task which calls the DLL. It is otherwise fully functional. The registered version does not contain this reminder.

DescendsFrom Function

Unit

FB_RTTI

Declaration

```
Function DescendsFrom(Ancestor : TObject;Test : TObject):Boolean;
```

Description

Returns true if Test is of a type which decends from the type of Ancestor.

Example

`DescendsFrom( AComponent, Form1 )     is true for a component AComponent and a form Form1`

Destroy Destructor

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Destructor Destroy; Override;
```

Description

Disposes of the component and disposes of all associated memory. If your Delphi application is the only application using the engine, and the last expression frees itself, the DLL will automatically unload.

See Also

Clear Method
Create Constructor

Determining If Expression Text has been Assigned

We can determine whether text has been assigned to our TExpression instance by querying the IsNull property. A value of TRUE indicates that text has been assigned to one of the Formula, StrFormula or Lines properties. IsNull also becomes true after a call to the TExpression.Clear method.

Determining an Expression's Return Type

As soon as the text expression is assigned to the Formula, StrFormula or Lines properties of a TExpression or descendant class, the engine "compiles" the text expression into a tokenized form. A benefit of this process is that the result type of the expression may then be determined without evaluating the expression. For example, if we had set the Formula property to each of the following strings, the ReturnType property would reflect the type of result that would be expected :

Text Expression	Return Type
'Sin(X) / LN(X^2)'	vtFLOAT
'TODAY() - 365'	vtDATE
'WEEKDAY(TODAY()) > 5'	vtBOOLEAN

You may use the ReturnType property to restrict expression types to those that fit your particular application domain.

Note There are certain built-in functions (CHOOSE and IIF for example) which may return any of the standard FormulaBuilder types. If these functions are used in a text expression, FormulaBuilder will try to determine the return type based on the other operators and operands used in the expression. In certain cases it is impossible for the engine to figure out the return type beforehand. In these instances a vtANY is returned.

Disclaimer

License Agreement

THIS SOFTWARE AND THE ACCOMPANYING FILES ARE SOLD "AS IS". THE AUTHOR DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING (WITHOUT LIMITATION), THE WARRANTIES AS TO PERFORMANCE OF MERCHANTABILITY, OR ANY OTHER WARRANTIES. Because of the various hardware and software environments in which FormulaBuilder may be used, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. The author assumes no liability for damages, direct or consequential, which may result from the use of FORMULABUILDER.

Good data processing procedure dictates that any program be thoroughly tested with non-critical data before relying on it.   The user must assume the entire risk of using the program.   ANY LIABILITY OF THE SELLER WILL BE LIMITED EXCLUSIVELY TO PRODUCT REPLACEMENT OR REFUND OF PURCHASE PRICE.

Although this documentation covers C/C++ as well as Visual Basic and Delphi, the author makes no claim or guarantee that the functionality of FormulaBuilder version 1.0 is available to C/C++. Testing with other Windows programming languages is ongoing, and they should be fully supported in an upcoming release.

Distribution

Please Read the License Agreement for important information. You are bound by the licensing restrictions contained in that document.

Demoware Version

Provided that you verify that you are distributing the Demoware Version you are hereby granted permission to make and distribute unlimited copies of the Demoware version of this software and documentation, provided that such copies are complete and unmodified duplicates of the original. There is no charge for any of the above.

You are specifically prohibited from charging, or requesting donations, for any such copies, however made; and from distributing the software and/or documentation with other products (commercial or otherwise) without prior written permission, with one exception: Disk Vendors approved by, and in good standing with the Association of Shareware Professionals are permitted to redistribute FormulaBuilder, subject to the conditions in this license, without specific written permission. However, under NO CONDITION are ***Software Of The Month Club (SOMC) inc*** or ***Digital Impact*** permitted to copy or distribute any portion of the software or its documentation.

Registered Version

UNDER NO CIRCUMSTANCE SHOULD ANY PORTION OF A REGISTERED VERSION OF FORMULABUILDER BE DISTRIBUTED except as outlined below :

Distributing FormulaBuilder Applications

Redistributing Files

You can redistribute the run time version of the software according to the terms of the license agreement. you can ship the following files with your application:

File	Description
FBCALC.DLL	FormulaBuilder Calculation Engine DLL
USERDOCS.RTF	RTF file containing documentation on the FormulaBuilder built-in functions and their use. You may modify and distribute this as appropriate to your end users.

EFBError, EFBDBError Class

See Also

Exception classes for the FormulaBuilder class wrappers. When the UseExceptions property of any instance of the FormulaBuilder Component Classes is set to true, FormulaBuilder will raise an exception of type EFBError and EFBDBError respectively, with its ErrorCode property set to the EXPR_XXX constant describing the error. In addition to the properties and methods inherited from TException, EFBError provides a constructor CreateEcode and a property ErrorCode.

Create

```
Constructor CreateEcode(const ecode : integer)
```

Create an instance of EFBError with the offending error passed in Ecode.

Errcode

```
Property ErrorCode : integer;
```

Returns the *EXPR_XXX* constant describing the error. This is the same value passed in to the constructor.

EXP Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the value of the mathematical constant *e* raised to the power x.

Syntax

EXP(X)

Remarks

This function is the inverse of the LN function, i.e. EXP(LN(X)) = x

See Also

LN

LOG

EXPR_CONVERT_ERROR = 32

EXPR_DOMAIN_ERROR = 20

EXPR_DUPLICATE_IDENT = 30

EXPR_FP_OVERFLOW = 22

EXPR_FP_UNDERFLOW = 23

EXPR_INT_OVERFLOW = 24

EXPR_INVALID_CALLBACK = 51

EXPR_INVALID_FUNCTION = 13

EXPR_RANGE_ERROR = 19

EXPR_SUCCESS = 1

EXPR_TYPE_MISMATCH = 10

EXPR_UNKNOWN_IDENT = 5

EXPR_XXX Constants

The EXPR_XXX constants are passed back from FormulaBuilder functions to indicate the status of an operation.

CONSTANT	VALUE	DESCRIPTION
EXPR_SUCCESS	1	SUCCESS
EXPR_MISSING_PAREN	2	Parenthesis expected
EXPR_BAD_EXPRESSION	3	Invalid Expression
EXPR_BAD_ASSIGNMENT	4	Bad assignment syntax
EXPR_UNKNOWN_IDENT	5	Unknown Identifier
EXPR_LINE_TOO_LONG	6	Line Too Long. String constants and identifiers cannot exceed 255 characters in length..
EXPR_INVALID_TOKEN	7	Invalid Token
EXPR_INVALID_CHAR	8	Invalid Character
EXPR_MISSING_PARAM	9	Parameter Expected
EXPR_TYPE_MISMATCH	10	The operand types in an operation were incompatible.
EXPR_INVALID_NUMBER	11	Invalid numeric format.
EXPR_MISSING_VARIABLE	12	Variable missing. Avariable encountered in the parsing phase is missing in the evaluation phase.
EXPR_INVALID_FUNCTION	13	Invalid function
EXPR_ZERO_DIVISION	14	Division by zero
EXPR_STACK_OVERFLOW	15	Evaluation stack overflow
EXPR_UNEXPECTED_EOS	16	Unexpected end of stream
EXPR_INVALID_DATE	17	Invalid Date format
EXPR_IDENTIFIER_EXPECTED	18	Identifier expected
EXPR_RANGE_ERROR	19	Value out of range
EXPR_DOMAIN_ERROR	20	the parameter for a function exceeds the function's domain of definition.
EXPR_MATH_ERROR	21	Math Error
EXPR_FP_OVERFLOW	22	Floating point overflow
EXPR_FP_UNDERFLOW	23	Floating point underflow
EXPR_INT_OVERFLOW	24	Integer overflow
EXPR_INVALID_OP	25	Invalid operation
EXPR_VARIABLE_EXPECTED	26	Variable expected
EXPR_MISSING_OPERATOR	27	Missing operator
EXPR_MISSING_OPERAND	28	Missing operand
EXPR_CONSTANT_EXPECTED	29	Constant expression expected
EXPR_DUPLICATE_IDENT	30	Duplicate identifier
EXPR_SYNTAX_ERROR	31	Syntax error
EXPR_CONVERT_ERROR	32	An error occurred during type conversion
EXPR_INVALID_TYPE	33	An invalid type was passed to a function or an operation between two incompatible operands was attempted
EXPR_INVALID_HANDLE	51	An invalid expression handle was passed to a DLL function
EXPR_INVALID_CALLBACK	53	An invalid or NULL callback was passed to a DLL function
EXPR_FORMULA_TOO_COMPLEX	54	The function is too complex to be evaluated

The following errors are generated by the data-aware Delphi classes TDSExpression , TDBExpression and TDSFilter.

CONSTANT	VALUE	DESCRIPTION
EXPR_INVALID_DATABASE	100	An invalid or null database was passed to the Database property, or the property was not set.
EXPR_INVALID_TABLENAME	111	An invalid table name exists in the input expression. The table does not exist in the database.
EXPR_INVALID_FIELDNAME	113	An invalid fieldname was specified. The fieldname does not exist for the specified table.
EXPR_INVALID_TABLE	115	A problem occurred with a field referenced by the expression during the evaluation phase.
EXPR_INVALID_FIELD	117	A problem occurred with a table referenced by the expression during the evaluation phase.

EXTRACT Function

See Also String Functions A-Z Function Reference

Description

Returns the nth delimited word from a string.

Syntax

Extract(*N,Source,Delims*)

N is the number of the word you wish to extract.
Source is the string from which to extract a word.
Delims is a string of delimiters which defines a word.

See Also

WORDCOUNT

See Also

- TExpression
- TDBExpression
- TDSExpression
- TDSFilter
- TRTTIExpression

Error Reporting Functions

Most FormulaBuilder DLL functions return one of the EXPR_XXX constants to indicate the status of an operation. The FBGetErrorString function returns a text message giving the explanation of an error code.

EvaluatePrim Method

See Also

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Procedure EvaluatePrim(var value : TVALUEREC); virtual;
```

Description

Evaluates the expression and returns the result in *value.* See the declaration of TValueRec for details. Use this function if you need access to the results of a calculation in native (as opposed to string) format. The tag field of value (vtype) gives the result type, and the appropriate field of the union/variant contains the resulting value. Don't forget to to call FBFreevalue to dispose of any memory associated with *value* .

See Also

AsString AsFloat
AsBoolean AsInteger
AsDate

StringResult

ExprData Data Passing : Example 2

The following code provides an example of accessing the calling TDSExpression instance from a callback. It duplicates existing BDE functionality and is not especially efficient, but it should give you an idea of the possibilities of using the ExprData parameter.

The Function

This example implements the DBCOUNT function, which returns the number of records in a dataset which matches a certain criterion.

DBCOUNT(<Filter>)

If Filter is ommitted, a count of all records in the dataset is returned.

The Code

```
(*********************************************************************************)

Uses Sysutils,DB,DBTables,FBDBCOMP,FBCALC;

...

Var
     fnidDBCOUNT : integer;


Procedure PrepareDataset( dataset : TDataset; var bookmark : TBookmark );
begin
  {Disable any components that reference the dataset.  Don't
   want those updating while we traverse the table.}
   dataset.DisableControls;
   BookMark := dataset.GetBookMark;
end;


Procedure RestoreDataset( dataset : TDataset; var bookmark : TBookmark );
begin
   With dataset do
   begin
     GotoBookmark(BookMark);
     FreeBookmark(BookMark);
     EnableControls;
   end;
end;


{ DBCOUNT(<Criteria>) }

Procedure DBCOUNT( nParamcount     : byte;
                   const params    : TActParamList;
                   var ReturnVal   : TVALUEREC;
                   var nErrCode    : Integer;
                   ExprData        : longint); export;
var
  exprFilter    : TDSExpression;
  tblDBCOUNT    : TDataset;
  lCount        : longint;
  BookMark      : TBookMark;
  ntype         : byte;

begin
  TRY
```

```
      {Cast ExprData back to its original type. Works only if this proc is called }
      {from a TDSExpression or Descendant }
      tblDBCOUNT := TDSExpression(ExprData).DataSet;
    EXCEPT
      nErrCode := EXPR_INVALID_DATASET;  { Invalid_Expression }
      exit;
    END;
   if (nParamcount = 0) or (params[0].vpString^ = '') then
   begin
     ReturnVal.vInteger := tblDBCOUNT.RecordCount;
     exit;
   end;
   exprFilter := TDSExpression.Create(NIL);
   with exprFilter do
   begin
     UseExceptions := False;
     Dataset       := tblDBCOUNT;
     Formula       := params[0].vpString^;
     nErrCode      := Status;
     if nErrCode <> EXPR_SUCCESS then
     begin
       Free;
       Exit;
     end;
     if not (ReturnType = vtBOOLEAN) then
     begin
       nErrCode := EXPR_TYPE_MISMATCH;    { EXPR_INVALID_FILTER }
       free;
       exit;
     end;
   end; {with }
   TRY
     PrepareDataset(tblDBCOUNT,BookMark);
     lCount := 0;
     TRY
       tblDBCOUNT.First;
       while not tblDBCOUNT.EOF do
       begin
         inc(lcount,ord(exprFilter.AsBoolean));
         if exprFilter.Status <> EXPR_SUCCESS then
         begin
           nErrCode := exprFilter.Status;
           RestoreDataset(tblDBCOUNT,BookMark);
           exit;
          end;
          tblDBCOUNT.Next;
        end;
        ReturnVal.vInteger := lCount;
      FINALLY
        RestoreDataset(tblDBCOUNT,BookMark);
      END;
     FINALLY
       exprFilter.Free;
     END;
end; { DBCount }



Procedure RegisterFunction;
begin
   InitFbuilder;
   fnIdDBCOUNT := FBRegisterFunction('DBSUM',vtINTEGER,'s',0,DBCOUNT);
end;
```

```
Procedure UnRegisterFunctions; far;
begin
  If not FBLoaded then exit;
  FBUnregisterFunction(fnIdDBCOUNT);
  FreeFBuilder;
end;


INITIALIZATION
  RegisterFunction;
  AddExitProc(UnRegisterFunctions);
END.
```

ExprData Data Passing Example

Observe the following code which implements the function WHOCALLED. The implicit typecast works only if WHOCALLED is called from a TExpression or descendant class:

```
Procedure ReturnCallerProc( paramcount     : byte;
                            const params   :  TActParamList;
                            var   retvalue : TValueRec;
                            var   errcode  : integer;
                                  exprdata : longint); export;
var i      : integer;
    expr   : TExpression absolute exprdata;   {implicit typecast}
    tmpstr : string;

begin
  try  {verify we are indeed being called from a TExpression }
   tmpstr := 'Called from '+Expr.ClassName+'.  Formula  = '+
             Expr.Formula + #0;
  Except
    on EGPFault do tmpstr := 'NOT called from a TExpression !'#0;
  end;
  retvalue.vpString := FBCreateString(@tmpstr[1]);
end;
```

Register the function as follows :

```
FBRegisterFunction('WHOCALLED',vtSTRING,NIL,0,ReturnCallerProc);
```

```
then use 'WHOCALLED()' in an expression.
```

This can be especially useful for subclasses of TExpression which add additional methods and properties. Using this method, we have access to the **public** and **published** methods and properties of the TExpression instance.

Expression Evaluation Functions

The following functions relate to determining the type, and finding the result of an expression previously set with FBSetExpression.

Function	Description
FBEvaluate	Evaluate the expression, returning the result as a string, regardless of the return type.
FBEvaluatePrim	Evaluate the expression, returning the result as a TValueRec
FBEvalExpression	Performs a one function expression evaluation given a string containing a valid expression.
FBFreeValue	Dispose of all memory associated with a TValueRec structure.
FBGetBooleanResult	Evaluate the expression, returning its boolean result.
FBGetDateResult	Evaluate an expression, returning its date/time result
FBGetFloatResult	Evaluate an expression, returning its floating point result
FBGetIntegerResult	Evaluate an expression, returning its integer (longint) result
FBGetStringResult	Evaluate an expression, returning its string result
FBReturnType	Determine the return type of the expression

Expression Initialization And Disposal

The following functions deal with initializing an expression, setting and clearing its text (formula), and disposing of expressions and their associated memory.

FBInitExpression
FBFreeExpression
FBSetExpression
FBReparseExpression
FBGetExpression
FBClearExpression

Extending FormulaBuilder

FormulaBuilder provides added flexibility by allowing the programmer to supplement the built-in functions and handle variable processing external to the core calculation engine.

Installing New Functions
External Variable/Field Handling

External Functions : Example 1

We will start with a simple, one parameter function ROMAN, which takes an integer value and returns the equivalent Roman numeral string.

```
 Procedure RomanProc( paramcount     : byte;
                      const params   : TActParamList;
                      var   retvalue : TValueRec;
                      var   errcode  : integer;
                            Exprdata : longint); export;
var number : longint;
    roman  : string[40];
begin
  number := params[0].vInteger;
  roman  := Romanize(number)+#0; { code for Romanize is in FBMisc.PAS }
  retvalue.vpString := FBCreateString(@Roman[1]);
end;
```

We must register the function to make it available to expressions. This is accompished with the FBRegisterFunction function call.

```
      RomanFnId := FBRegisterFunction('ROMAN',vtSTRING,'i',1,RomanProc);
```

The first argument is, of course, our new function name. The second is the return type. Since our function takes a single integer argument, the *params* argument is set to 'i'. The fourth parameter to FBRegisterFunction tells the parser the minimum number of parameters the parser should expect. The final parameter, of course, is a pointer to the function which implements "ROMAN".

Thats It ! Youve successfully added a function to FormulaBuilder. "ROMAN" will be treated like any of FormulaBuilder's other functions.   This can be verified by the code similar to the following :

```
        Expression.Formula := 'ROMAN(1996)';
     Edit1.Text := Expression.AsString;
```

External Functions : Example 2

Consider the Compound Interest Formula

A = P * (1 + i)^*n*

where *A* is the accumulated value, *P* is the principal, *I* is the rate of interest and *n* is the number of periods. Here is how the function could be implemented :

```
Procedure CompoundInterestProc(paramcount : byte;
                              const params : TActParamList;
                              var Retvalue : TValueRec ;
                              var errcode  : integer;
                                  ExprData : longint); export;
var p, I , n : double;
      A    : extended;
begin
    p := params[0].vFloat;
    I := params[1].vFloat;
    N := params[2].vFloat;
    A := P * power(1 + i,n);
    retvalue.vFloat  := A;
end;
```

To register the function, we would do the following :

```
MyFuncId := FBRegisterFunction('CompInterest',vtFLOAT,
                                'fff',3,CompoundInterestProc);
```

As we can see, our function name is 'CompInterest', it returns a float, has 3 float parameters, requires all three parameters, and is implemented by the callback CompoundInterestProc.

External Functions and the vtANY (Variant) type

Occasionally we need to use functions whose parameter or return types are not known before we execute the function. If we consider spreadsheets, for example, we can use aggregation functions (SUM, AVG, etc) on ranges of cells which may contain text, formulas, floating point values, boolean values, and so on. The built-in function STR also takes any expression type and returns its text equivalent. In order to allow this flexibility,   FormulaBuilder uses the vtANY (variant) type.

How the Parser Treats vtANY

When a   parameter is described as variant, the parser suspends type checking for the parameter when the expression is tokenized. The callback function will have to use the *vtype* field of the appropriate parameter to determine the actual type passed to the callback procedure. It should be noted that a TValueRec passed by FormulaBuilder NEVER has its *vtype* field set to vtANY. The vtANY constant simply lets the parser know that any value type may be entered as a parameter or returned from a function. Similarly, the value type field of the *Retvalue* parameter should be set to one of the other vtXXX constants describing the function's return type, otherwise error EXPR_TYPE_MISMATCH will occur.

The following are examples of how to use the variant types in external functions :

Example 1
Example 2
Example 3

External Variable/Field Handling

Callback routines extend the flexibility of FormulaBuilder by adding the capability of handling variables and fields in programmer written code. Variable data can then be extracted directly from the data source without reparsing the expression. Variables and fields are handled identically, so the discussion on variable handling applies equally to fields.

Establishing Callbacks
To register callbacks to handle variables and fields, use the FBSetVariableCallbacks function.

Callback Function Types
TCBKFindVariable
TCBKGetVariable
TCBKSetVariable

The Callback Handling Process

For the sake of clarity in the following discussion, you should revisit the FormulaBuilder evaluation process as it relates to variable and field callbacks.

In the Parsing Phase, the string formula is decomposed into its constituent parts - variables, fields, operators and functions. When the FormulaBuilder encounters an identifier it does not recognize, it calls the callback routine of type TCBKFindVariable to allow the programmer to determine whether the identifier represents a valid variable. During the evaluation phase the variable callback routine of type TCBKGetVariable is called to furnish the engine with the current value of the variable. If the original expression contained an assignment statement which updated the variable (the variable was the LValue of the expression) the TCBKSetVariable routine would be called with the result of the evaluation to allow the programmer to update the variable or field.

Delphi Users
For a more thorough discussion of this subject, see the section Using FormulaBuilder with Delphi.

FACT Function

[See Also](#)	[Math/Trig Functions](#)	[A-Z Function Reference](#)

Description
Returns the factorial of a positive number.

Syntax
FACT(*number*)

number is any positive number

Remarks
The factorial of a number X is equal to 1*2*3*...X. If *number* is a floating point value, it will be truncated to an integer before the calculation is performed..

See Also

PRODUCT

FBAddBooleanConstant Function

See Also Constant Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBAddBooleanConstant(name : Pchar;value : BOOL):integer;
```

C/C++

```
FBERROR FBAPI FBAddBooleanConstant(LPSTR name,BOOL value);
```

VB

```
Function FBAddBooleanConstant% LIB "FBCALC.DLL" (ByVal name$,ByVal value%)
```

Description

Adds a boolean constant to FormulaBuilder's global symbol table.

Example

```
FBAddBooleanConstant('Approved', -1)
```

See Also

- FBAddConstantPrim
- FBAddDateConstant
- FBAddNumericConstant
- FBAddStringConstant
- FBFreeConstant

FBAddConstantPrim Function

See Also   Constant Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBAddConstantPrim(name : pchar;var value : TValueRec):Integer;
```

C/C++

```
FBERROR FBAPI FBAddConstantPrim(LPSTR name,LPVALUEREC value);
```

Description

Add a constant in the form of a TValueRec. Do **NOT** dispose of value with the FBFreeValue call.

See Also

FBAddBooleanConstant
FBAddDateConstant
FBAddNumericConstant
FBAddStringConstant
FBFreeConstant

FBAddDateConstant Function

see also example Constant Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBAddDateConstant(name : Pchar;value : TFBDate):integer;
```

C/C++

```
FBERROR FBAPI FBAddDateConstant(LPSTR name,TFBDate value);
```

VB

```
Function FBAddDateConstant% Lib "FBCALC.DLL" (ByVal name$,ByVal value#)
```

Description

Add a date constant *name* to the engine with value *value*.

FBAddDateConstant Example

```
FBlpzToFBDate('10/10/32',MomsBDate)
FBAddDateConstant(‘MomsBDay’,MomsBDate);
FBSetExpression(expr,'iif(Month(Today()) = Month(MomsBDay)) &(Day(Today()) =
Day(MomsBDay)),”Aren't you forgetting something ?”, “Safe for the moment”)’);
```

See Also

FBAddBooleanConstant
FBAddConstantPrim
FBAddNumericConstant
FBAddStringConstant
FBFreeConstant

FBAddNumericConstant Function

See Also   Constant Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBAddNumericConstant(name : Pchar;value : double):integer;
```

C/C++

```
FBERROR FBAPI FBAddNumericConstant(LPSTR name,double value);
```

VB

```
Function FBAddNumericConstant% Lib "FBCALC.DLL" (ByVal name$,ByVal value#)
```

Description

Adds a numeric constant to the engine.

Example

Pascal

```
FBAddNumericConstant('E',2.718282)
```

C/C++ and VB

```
FBAddNumericConstant("e",2.718282);
```

See Also

FBAddBooleanConstant
FBAddConstantPrim
FBAddDateConstant
FBAddStringConstant
FBFreeConstant

FBAddStringConstant Function

See Also   example   Constant Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBAddStringConstant(name : Pchar;value : pchar):integer;
```

C/C++

```
FBERROR FBAPI FBAddStringConstant(LPSTR name,LPSTR value);
```

VB

```
Function FBAddStringConstant% Lib "FBCALC.DLL" (ByVal name$,ByVal value$)
```

Description

Adds a string constant to named *name* with value *value* FormulaBuilder’s global symbol table.

FBAddStringConstant Example

```
FBAddStringConstant("company","YGB Software")
```

See Also

FBAddBooleanConstant
FBAddConstantPrim
FBAddDateConstant
FBAddNumericConstant
FBFreeConstant

FBAddVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBAddVariable(handle : HEXPR;name : pchar;vtype : byte):integer;
```

C/C++

```
FBERROR FBAPI FBAddVariable(HEXPR handle,LPSTR name,BYTE vtype);
```

VB

```
Declare Function FBAddVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
name$,ByVal vtype%)
```

Description

Adds a variable of type *vtype* to the engine for the expression with handle *handle*. *name* then becomes available for use in expressions. The initial value will be the NULL representation appropriate to the variable’s type.

Remarks

A single expression can own up to 16,000 variables, memory permitting. Parsing may be slower for a large number of variables, but there is no performance penalty in the actual evaluation process.

See Also

FBParseAddVariable
FBFreeVariable

The FBCALC Import Unit

The FBCALC unit is a dynamic import unit to interface with the FormulaBuilder DLL. Since the calculation engine is explicitly loaded, we must ensure that the DLL is loaded before attempts are made to call its exported functions. If you intend to make calls to DLL functions outside of those encapsulated in components, please take a minute to read the following topics :

Routines
CheckLoadFB
FBLoaded
FreeFBuilder
InitFBuilder

All other routines that access the functionality of the engine are documented in the DLL Reference.

FBClearExpression Function

See Also   Expression Initialization and Disposal   DLL Reference   A-Z Function Reference

Pascal

```
Function FBClearExpression(handle : HEXPR):integer;
```

C/C++

```
FBERROR FBAPI FBClearExpression(HEXPR handle);
```

VB

```
Declare Function FBClearExpression% LIB "FBCALC.DLL" (ByVal handle&)
```

Description

Clears the internal state of the expression. All defined variables are freed and the expression *handle* is returned to the state it would be in after a FBInitExpression call.

See Also

FBFreeExpression

FBInitExpression

FBComp Unit

The FBComp unit contains the declarations for TExpression and its associated objects, types, and routines.The FBComp unit is automatically added to the uses clause whenever you add a TExpression component to your form.The following items are declared in the FBComp unit:

Components
TExpression

Objects
EFBError

Types
TFindVariableEvent
TGetVariableEvent
TSetVariableEvent
TVariable

Routines
Register
GetFunctionPrototypes
ValueAsString

FBCopyValue Function

See Also Utility Routines DLL Reference A-Z Function Reference

Pascal

```
Function FBCopyValue(value : TVALUEREC):TValueRec;
```

C/C++

```
TVALUEREC FBCopyValue(TVALUEREC value);
```

Description

Returns a copy of the *value* structure. You should use this method instead of manually copying structure items to protect yourself against changes in the implementation of TValueRec or its field types.

See Also

FBFreeValue

FBCreateString Function

See Also	Utility Routines	DLL Reference	A-Z Function Reference

Pascal

```
Function FBCreateString(str : pchar):TFBString;
```

C/C++

```
TFBSTRING FBAPI FBCreateString(LPSTR str);
```

Description

Creates a FormulaBuilder string (a Borland Delphi/Pascal Pstring) from a null-terminated string. This may be used when setting the vpString component of the TValueRec union/variant. This routine is included for the convenience of C/C++ programmers who wish to use the more advanced features of FormulaBuilder, hence needing access to the TValueRec type.

See Also

FBStrncpy

FBDBComp Unit

The FBDBComp unit contains the declarations for the FormulaBuilder data-aware components. This unit is automatically added to the uses clause whenever you create a add a data-aware component to your form.The following items are declared in the FBDBComp unit:

Components
TDBExpression
TDSExpression
TDSFilter

Objects
EFBDBError

Routines
Register
FieldDataType
GetErrorString
IsValidDBExpression

FBDateToPasString Function

[Utility Routines] [DLL Reference] [A-Z Function Reference]

Declaration

```
Function FBDateToPasString(date : TFBDate):String;
```

Description

Converts a FormulaBuilder date type to a Pascal String.

FBEnumFunctions Function

[Example](#) [Function Handling Routines](#) [DLL Reference](#) [A-Z Function Reference](#)

Pascal

```
Function FBEnumFunctions(fncbk : TCBKEnumFunctions ; Enumdata :
longint):integer;
```

C/C++

```
FBERROR FBAPI FBEnumFunctions(TCBKEnumFunctions fncbk,LONG Enumdata);
```

Description

Enumerate all registered functions, calling *fnCBK* for each. The parameter *Enumdata* is passed to the callback *fnCBK* on each iteration. Note that since Visual Basic does not support callbacks, this function is unavailable for that environment.

FBEnumFunctions Example

Delphi

The following example shows how to use the FBEnumFunctions call to obtain a list of the names of all registered functions. Notice the use of typecasting with the *EnumData* parameter.

```
Function getFuncNames(name       : pchar;
                      vtype      : byte;
                      parms      : pchar;
                      minPrms    : byte;
                      EnumData   : longint):integer; export;

var  List : TStringList absolute EnumData; { implicit typecast }

begin
  if not Assigned(List) then
     List := TStringList.Create;
  List.Add( strpas(vname) );
end;

Function getFunctionNames : TStringList;
begin
  Result  := TStringList.Create;
  FBEnumFunctions(getFuncNames,Longint(Result));
end;
```

C/C++

```
typedef char *StringList[120], *LPStringList;
static int iCount = 0;

FBERROR FBAPI EXPORT getFuncnames(LPCSTR name,BYTE vtype,LPCSTR parms,BYTE
minPrms,LONG EnumData)
{
  *(LPStringList(EnumData))[iCount++] = name;
};


StringList getFunctionNames(LPINT count)
{
  iCount = 0;
  StringList List;

  FBEnumFunctions(getFuncNames,LONG(&List));

  *count = iCount;
  return List ;
}
```

FBEvalExpression Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBEvalExpression(expr : pchar;var retType : datatypes;buf : pchar;
buflen : word):integer;
```

C/C++

```
FBERROR FBAPI FBEvalExpression(LPSTR expr,LPBYTE retType,LPSTR buf,WORD
buflen);
```

VB

```
Declare Function FBEvalExpression% Lib "FBCALC.DLL (ByVal expr$,retType%,ByVal
buf$,ByVal buflen%)
```

Description

Perform a single operation expression evaluation. Evaluates the formula *expr*, returning its return type in *retType*, and up to *buflen* characters of the string representation of its result in *buf.*

See Also

FBEvaluate
FBEvaluatePrim

FBEvaluate Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBEvaluate(handle : HEXPR;buf : pchar;buflen : word):integer;
```

C/C++

```
FBERROR FBAPI FBEvaluate(HEXPR handle,LPSTR buf,WORD buflen);
```

VB

```
Declare Function FBEvaluate% lib "FBCALC.DLL" (ByVal handle&,ByVal buf$,ByVal
buflen%)
```

Description

Evaluates the expression previously set by FBSetExpression and copies the null-terminated string result to the buffer pointed to by *buf*. The programmer should set *buflen* to the maximum number of characters that can be copied to *buf*. This is the most efficient way to recalculate an expression that has not changed since a call to FBSetExpression. This is because the FBEvaluate performs its calculations without having to reparse the input expression. This is especially beneficial in loops where only the value of variables change.

See Also

FBEvaluatePrim
FBGetFloatResult
FBGetIntegerResult
FBGetBooleanResult
FBGetDateResult
FBGetStringResult

FBEvaluatePrim Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBEvaluatePrim(handle:HEXPR;var value : TVALUEREC):integer;
```

C/C++

```
FBERROR FBAPI FBEvaluatePrim(HEXPR handle,LPVALUEREC value);
```

Description

Evaluates the expression previously set by FBSetExpression and returns the result in *value.* See the declaration of TValueRec for details. Use this function is you need access to the results of a calculation in native (as opposed to string) format. The tag field of value (vtype) gives the result type, and the appropriate field of the union/variant contains the resulting value. Don't forget to to call FBFreevalue to dispose of any memory associated with *value*.

See Also

FBEvaluate
FBGetBooleanResult
FBGetDateResult
FBGetFloatResult
FBGetIntegerResult
FBGetStringResult

FBFreeConstant Function

See Also Constant Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBFreeConstant(name : pchar):integer;
```

C/C++

```
FBERROR FBAPI FBFreeConstant(LPSTR name);
```

VB

```
Declare Function FBFreeConstant% Lib "FBCALC.DLL" (ByVal name$)
```

Description

Removes the constant named *name* from FormulaBuilder's symbol table and frees all associated memory.

See Also

FBFreeConstants
FBFreeVariable

FBFreeConstants Function

[See Also] [Constant Handling Functions] [DLL Reference] [A-Z Function Reference]

Pascal

```
Function FBFreeConstants :integer;
```

C/C++

```
FBERROR FBAPI FBFreeConstants();
```

Syntax(VB)

```
Declare Function FBFreeConstants% Lib "FBCALC.DLL" ()
```

Description

Removes ALL constants from FormulaBuilder’s symbol table. Since constants are system global, this should only be called with great caution !

See Also

[FBFreeConstant]

[FBFreeVariableList]

FBFreeExpression Function

See Also Expression Initialization and Disposal DLL Reference A-Z Function Reference

Pascal

```
Function FBFreeExpression(handle : HEXPR) : integer;
```

C/C++

```
FBERROR FBAPI FBFreeExpression(HEXPR handle);
```

VB

```
Declare Function FBFreeExpression% Lib "FBCALC.DLL" (ByVal handle&)
```

Description

Frees all memory associated with an expression. Handle is the same as was returned with the FbInitExpression call.

See Also

FBClearExpression
FBInitExpression

FBFreeValue Function

See Also Utility Routines DLL Reference A-Z Function Reference

Pascal

```
Procedure FBFreeValue(var value  : TValueRec);
```

C/C++

```
void FBAPI FBFreeValue(LPVALUEREC value);
```

Description

FBFreeValue disposes of any memory associated with *value*. This is only strictly necessary when value.vtype is vtSTRING.

See Also

FBCopyValue

FBFreeVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBFreeVariable(handle : HEXPR;name : pchar):integer;
```

C/C++

```
FBERROR FBAPI FBFreeVariable(HEXPR handle,LPSTR name);
```

VB

```
Declare Function FBFreeVariable% Lib "FBCALC.DLL"(ByVal handle&,ByVal name$)
```

Description

Free all memory associated with the variable *name* and remove it from the expressions variable list.

See Also

FBAddVariable
FBFreeVariableList

FBFreeVariableList Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBFreeVariableList(handle : HEXPR) : Integer;
```

C/C++

```
FBERROR FBAPI FBFreeVariableList(HEXPR handle);
```

VB

```
Declare Function FBFreeVariableList% lib "FBCALC.DLL" (ByVal handle&)
```

Description

Dispose of all declared variable for the given expression. This is done implicitly on a FBFreeExpression call.

Note. This >>>will << cause a problem, and possibly a GPF, if any of the variables were referenced in the expression referenced by handle, and an attempt is made to evaluate the expression.

See Also

FBAddVariable
FBFreeVariable

FBGetBooleanResult Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBGetBooleanResult(handle : HEXPR;var value : BOOL):integer;
```

C/C++

```
FBERROR FBAPI FBGetBooleanResult(HEXPR handle,LPBOOL value)
```

VB

```
Declare Function FBGetBooleanResult% LIB "FBCALC.DLL" (ByVal handle&,value%)
```

Description

Evaluates the expression with handle *handle*, and returns the boolean result in *value*. If the return type of the expression is not vtBOOLEAN, FBGetBooleanResult returns EXPR_TYPE_MISMATCH. The return type of an expression can be determined with the FBGetReturnType call.

See Also

[FBEvaluate]
[FBEvaluatePrim]
[FBGetReturnType]

FBGetBooleanVariable Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetBooleanVariable(handle : HEXPR;vname : pchar;var value : BOOL);
```

C/C++

```
FBERROR FBAPI FBGetBooleanVariable(HEXPR handle,LPSTR vname,LPBOOL value);
```

VB

```
Declare Function FBGetBooleanVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,value%)
```

Description

Assigns the value of the boolean variable *vname* to *value*. FBGetBooleanVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a boolean. Otherwise this function returns EXPR_SUCCESS.

See Also

FBGetVarAsString
FBGetVariablePrim
FBGetVarPtr
FBSetBooleanVariable

FBGetConstAsString Function

See Also   Constant Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetConstAsString(name : pchar;buf : pchar;buflen : word);
```

C/C++

```
FBERROR FBAPI FBGetConstAsString(LPSTR name,LPSTR buf,WORD buflen);
```

VB

```
Declare Function FBGetConstAsString% Lib "FBCALC.DLL" (ByVal name$,ByVal
buf$,ByVal bufLen)
```

Description

Copies up to *bufLen* of the string representation of the value of the constant named *name* into the string pointed to by *buf*.

See Also

FBGetConstantPrim

FBGetConstantPrim Function

See Also Constant Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBGetConstantPrim(name : pchar;var value : TValueRec):integer;
```

C/C++

```
FBERROR FBAPI FBGetConstantPrim(LPSTR name,LPVALUEREC value);
```

Description

Retrieves a copy of the value of the constant named *name* into *value.* See the Type And Constant Reference for details on the TValueRec type. Memory allocated for this structure should be freed by calling FBFreeValue after *value* is no longer needed (this is only strictly necessary if *value.vtype* is vtSTRING).

See Also

FBGetConstAsString

FBGetDateResult Function

[See Also](#) [Expression Evaluation Functionsl](#) [DLL Reference](#) [A-Z Function Reference](#)

Pascal

```
Function FBGetDateResult(handle : HEXPR;var value : TFBDate):integer;
```

C/C+

```
FBERROR FBAPI FBGetDateResult(HEXPR handle,LPFBDATE value)
```

VB

```
Declare Function FBGetDateResult% LIB "FBCALC.DLL" (ByVal handle&,value#)
```

Description

Evaluates the expression with handle *handle*, and returns the date result in *value*. If the return type of the expression is not vtDATE, FBGetDateResult returns EXPR_TYPE_MISMATCH. The return type of an expression can be determined with the FBGetReturnType call.

See Also

FBEvaluate
FBEvaluatePrim
FBGetReturnType

FBGetDateVariable Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetDateVariable(handle : HEXPR;vname : pchar;var value :
TFBDate):integer;
```

C/C++

```
FBERROR FBAPI FBGetDateVariable(HEXPR handle,LPSTR vname,LPFBDATE value);
```

VB

```
Declare Function FBGetDateVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,value#)
```

Description

Assigns the value of the date variable *vname* to *value*. FBGetDateVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a date. Otherwise this function returns EXPR_SUCCESS.

See Also

FBGetVarAsString
FBGetVariablePrim
FBGetVarPtr
FBSetDateVariable

FBGetEnumValue Function

Unit

FB_RTTI

Declaration

```
Function FBGetEnumValue(Root : TObject; Const EnumName : String; var Instance
: TObject;  var EnumTypeInfo : PTypeInfo):integer;
```

Description

Performs a recursive search from root downward to see if *EnumName* exists as one of the members of a published enumeration or set type. If root is a component, its component list is also recursively searched. Returns the ordinal value of *EnumName*, or -1 if it is not found. The type information record for the enumeration type in which *EnumName* occurs is returned in *EnumTypeInfo*.

FBGetErrorString Function

See Also DLL Reference A-Z Function Reference

Pascal

```
Procedure FBGetErrorString(ecode : integer;buf : pchar;buflen : word);
```

C/C++

```
void FBAPI FBGetErrorString(int ecode,LPSTR buf,WORD bufLen);
```

VB

```
Declare Sub FBGetErrorString Lib "FBCALC.DLL" (ByVal ecode%,ByVal buf$,ByVal
bufLen%)
```

Description

Returns, in *buf*, the null-terminated string description of the error *ecode*, up to a maximum of *bufLen* characters. *ecode* is one of the EXPR_XXX constants.

See Also

EXPR_XXX Constants

FBGetExpression Function

see also Expression Initialization and Disposal DLL Reference A-Z Function Reference

Pascal

```
Function FBGetExpression(handle: HEXPR;expr :pchar;buflen:word):integer;
```

C/C++

```
int FBAPI FBGetExpression(HEXPR handle,LPSTR expr,WORD buflen);
```

VB

```
Declare Function FBGetExpression% Lib "FBCALC.DLL" (ByVal handle&,ByVal
expr$,ByVal maxlen%)
```

Description

Returns the last infix expression successfully added by a FBSetExpression call. The string is copied to *expr*, up to *buflen* characters.

see also

FBClearExpression Function
FBSetExpression Function

FBGetFloatResult Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBGetFloatResult(handle : HEXPR;var value : double):integer;
```

C/C++

```
FBERROR FBAPI FBGetFloatResult(HEXPR handle,LPDOUBLE value)
```

VB

```
Declare Function FBGetFloatResult% LIB "FBCALC.DLL" (ByVal handle&,value#)
```

Description

Evaluates the expression with handle *handle*, and returns the floating point result in *value*. If the return type of the expression is not vtFLOAT, FBGetFloatResult returns EXPR_TYPE_MISMATCH. The return type of an expression can be determined with the FBGetReturnType call.

See Also

FBEvaluate
FBEvaluatePrim
FBGetReturnType

FBGetFloatVariable Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetFloatVariable(handle : HEXPR;vname : pchar;var value :
double):integer;
```

C/C++

```
FBERROR FBAPI FBGetFloatVariable(HEXPR handle,LPSTR vname,LPDOUBLE value);
```

VB

```
Declare Function FBGetFloatVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,value#)
```

Description

Assigns the value of the float variable *vname* to *value*. FBGetFloatVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a float. Otherwise this function returns EXPR_SUCCESS.

See Also

FBGetVarAsString
FBGetVariablePrim
FBGetVarPtr
FBSetFloatVariable

FBGetFunctionCount Function

See Also Function Handling Routines DLL Reference A-Z Function Reference

Pascal

```
Function FBGetFunctionCount : word;
```

C/C++

```
WORD FBAPI FBGetFunctionCount();
```

VB

```
Declare Function FBGetFunctionCount% Lib "FBCALC.DLL" ()
```

Description

Returns the number of functions (internal and programmer-defined) registered with FormulaBuilder.

See Also

FBGetFunctionProto
FBEnumFunctions
FBRegisterFunction
FBUnregisterFunction

FBGetFunctionProto Function

see also   Function Handling Routines   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetFunctionProto(funcname : pchar;var vtype : byte;params :
pchar;var minprms : byte):integer;
```

C/C++

```
FBERROR FBAPI FBGetFunctionProto(LPCSTR funcname,LPBYTE vtype,LPSTR
params,LPBYTE minprms);
```

VB

```
Declare Function FBGetFunctionProto% LIB "FBCALC.DLL" (ByVal funcname$,vtype
%,ByVal params$,minprms%)
```

Description

Returns information on the single function named *funcname*, whether it is internal to FormulaBuilder or programmer-installed.

Parameter	Description
func*name*	the name of the function.
vtype	function return type. See the vtXXX constants
parms	a pointer to a null-terminated string in which each character represents the type of parameter for that position. There string is no longer than MAXPARAMS+1 characters long.You should copy this string to a buffer in your program. DO NOT ATTEMPT TO MODIFY IT.
minPrms	the minimum allowable number of parameters, for functions with variable parameter lists

see also

FBRegisterFunction

FBUnregisterFunction

FBGetIntegerResult Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBGetIntegerResult(handle : HEXPR;var value : longint):integer;
```

C/C++

```
FBERROR FBAPI FBGetIntegerResult(HEXPR handle,LPLONG value)
```

VB

```
Declare Function FBGetIntegerResult% LIB "FBCALC.DLL" (ByVal handle&,value&)
```

Description

Evaluates the expression with handle *handle*, and returns the integer (longint) result in *value*. If the return type of the expression is not vtINTEGER, FBGetIntegerResult returns EXPR_TYPE_MISMATCH. The return type of an expression can be determined with the FBGetReturnType call.

See Also

FBEvaluate
FBEvaluatePrim
FBGetReturnType

FBGetIntegerVariable Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetIntegerVariable(handle : HEXPR;vname : pchar;var value :
longint):integer;
```

C/C++

```
FBERROR FBAPI FBGetIntegerVariable(HEXPR handle,LPSTR vname,LPLONG value);
```

VB

```
Declare Function FBGetIntegerVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,value&)
```

Description

Copies the value of the integer variable *vname* into the parameter *value*. FBGetIntegerVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a integer.   If no errors occur, this function returns EXPR_SUCCESS.

See Also

[FBGetVarAsString]
[FBGetVariablePrim]
[FBGetVarPtr]
[FBSetIntegerVariable]

FBGetReturnType Function

[Expression Evaluation Functionsl](#) [DLL Reference](#) [A-Z Function Reference](#)

Pascal

```
Function FBGetReturnType(handle : HEXPR) : integer;
```

C/C++

```
int FBAPI FBGetReturnType(HEXPR handle);
```

VB

```
Declare Function FBGetReturnType% Lib "FBCALC.DLL" (ByVal handle&)
```

Description

Gets the return type of the expression. Valid values are the vtXXX constants.

If the expression is invalid or empty, the function returns vtTYPEMISMATCH.

FBGetStringResult Function

See Also Expression Evaluation Functionsl DLL Reference A-Z Function Reference

Pascal

```
Function FBGetStringResult(handle : HEXPR;value : pchar;maxlen :
word):integer;
```

C/C++

```
FBERROR FBAPI FBGetStringResult(HEXPR handle,LPSTR value,WORD maxlen);
```

VB

```
Declare Function FBGetStringResult% Lib "FBCALC.DLL" (ByVal handle&,ByVal
value$,ByVal maxlen%)
```

Description

Evaluates the expression with handle *handle*, and returns the string result in *value*. If the return type of the expression is not vtSTRING, FBGetStringResult returns EXPR_TYPE_MISMATCH. The return type of an expression can be determined with the FBGetReturnType call.

See Also

FBEvaluate
FBEvaluatePrim
FBGetReturnType

FBGetStringVariable Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetStringVariable(handle : HEXPR;vname : pchar;value : pchar;maxlen
: word):integer;
```

C/C++

```
FBERROR FBAPI FBGetStringVariable(HEXPR handle,LPSTR vname,LPSTR value,WORD
maxlen);
```

VB

```
Declare Function FBGetStringVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,ByVal value$,ByVal maxlen%)
```

Description

Copies up to *maxlen* characters of the value of the string variable *vname* into the parameter *value*. FBGetStringVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a string. Otherwise this function returns EXPR_SUCCESS.

See Also

[FBGetVarAsString]
[FBGetVariablePrim]
[FBGetVarPtr]
[FBSetStringVariable]

FBGetVarAsString Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBGetVarAsString(handle : HEXPR;name : pchar;value : pchar;buflen :
word):integer;
```

C/C++

```
FBERROR FBAPI FBGetVarAsString(HEXPR handle,LPSTR name,LPSTR value,WORD
buflen);
```

VB

```
Declare Function FBGetVarAsString% lib "FBCALC.DLL" (ByVal handle&,ByVal
name$,ByVal value$,ByVal buflen%)
```

Description

Returns the string representation of the value of the variable *name* in value, up to *buflen* characters.

See Also

FBGetBooleanVariable
FBGetDateVariable
FBGetFloatVariable
FBGetIntegerVariable
FBGetStringVariable
FBSetVarFromString

FBGetVarPtr Function

See Also example Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

Function FBGetVarPtr(*name* : pchar;**var** *vtype* : byte;**var** *value* : pointer):integer;

C/C++

FBERROR FBAPI FBGetVarPtr(LPSTR *name*,LPBYTE *vtype*,LPVOID **value*);

Description

Returns a pointer to the value of a variable maintained in the expression's variable list. This may be beneficial in instances where the expression needs to be evaluated through a large number of iterations based on the value of the variable.

Parameter	Description
name	the name of the variable
vtype	the variable type. It is one of the vtXXX constants
value	the pointer to the variable. Note that for string variables (vtype = vtSTRING), the value returned is the actual string pointer itself, not a pointer to the pointer.

FBGetVarPtr Example

Pascal

```
var
   handle : HEXPR;
   xtype,ytype : byte;
   xptr : ^longint;
   yptr : ^double;
   total : extended;
   v   : TValueRec;

begin
     handle = FBInitExpression;
     FBAddVariable(handle,'X',vtFloat);
     FBAddVariable(handle,'Y',vtFloat);
     FBSetExpression(handle,'y := rand(x)');
     FBGetVarPtr(handle,'X',xtype,longint(xptr));
     FBGetVarPtr(handle,'Y',ytype,longint(yptr));
     total := 0;
    for xptr^ := 1 to 2500 do
    begin
         FBEvaluatePrim(handle,v);
         total := total + yptr^
    end;
end;
```

C/C++

```
double *xptr,*yptr;
HEXPR handle;
unsigned char xtype,ytype;
TValueRec v;

handle = FBInitExpression;
FBAddVariable(handle,'X',vtInteger);
FBAddVariable(handle,'Y',vtFloat);
FBSetExpression(handle,'y := rand(x)');
FBGetVarPtr(handle,'X',xtype,xptr);
FBGetVarPtr(handle,'Y',ytype,yptr);
double total = 0;
for (int *xptr = 1;*xptr <= 2500; *xptr++) {
     FBEvaluatePrim(handle,v);
     total = total + *yptr;
};
/* total now contains sum of random numbers */
```

See Also

FBGetBooleanVariable
FBGetDateVariable
FBGetFloatVariable
FBGetIntegerVariable
FBGetStringVariable
FBGetVariablePrim
FBPeekVariable

FBGetVariableCount Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBGetVariableCount(handle : HEXPR ) : integer;
```

C/C++

```
int FBAPI FBGetVariableCount(HEXPR handle);
```

VB

```
Declare Function FBGetVariableCount% Lib "FBCALC.DLL" (ByVal handle&)
```

Description

Returns the number of variables successfully added with FBAddVariable for the expression referenced by *handle.*

See Also

- FBAddVariable
- FBFreeVariable
- FBFreeVariableList
- FBPeekVariable
- FBPeekVarVB

FBGetVariablePrim Function

[See Also](#)   [Variable Handling Functions](#)   [DLL Reference](#)   [A-Z Function Reference](#)

Pascal

```
Function FBGetVariablePrim(handle : HEXPR;name : pchar;Var value :
TValueRec):integer;
```

C/C++

```
FBERROR FBAPI FBGetVariablePrim(HEXPR handle,LPSTR name,LPVALUEREC value);
```

Description

Returns the value of the variable *name* in *value*. *value*.vtype contains the variable type, with the appropriate variant field containing its value. FBFreeValue should be called to dispose of value when it is no longer needed.

See Also

- FBGetBooleanVariable
- FBGetDateVariable
- FBGetFloatVariable
- FBGetStringVariable
- FBPeekVariable
- FBSetVariablePrim

FBGetXXXResult functions

Function	Description
FBGetBooleanResult	Evaluate the expression, returning its boolean result.
FBGetDateResult	Evaluate an expression, returning its date/time result
FBGetFloatResult	Evaluate an expression, returning its floating point result
FBGetIntegerResult	Evaluate an expression, returning its integer (longint) result
FBGetStringResult	Evaluate an expression, returning its string result

FBGetXXXVariable Functions

- FBGetBooleanVariable
- FBGetDateVariable
- FBGetFloatVariable
- FBGetIntegerVariable
- FBGetStringVariable

FBInitExpression Function

See Also	Expression Initialization and Disposal	DLL Reference	A-Z Function Reference

Pascal

```
Function FBInitExpression(exprData : longint) : HEXPR;
```

C/C++

```
HEXPR FBAPI FBInitExpression(LONG exprData);
```

VB

```
Declare Function FBInitExpression& Lib "FBCALC.DLL" (ByVal exprData&)
```

Description

Allocate a handle for a new expression. Returns the expression handle, or a negative integer value if the initialization fails. The *exprData* parameter (unused in VB) is a programmer definable value provided to allow you to pass data to programmer defined functions. The value used for *exprData* is passed in the *exprData* field of external functions. See TCBKExternalFunc for details. If you are not making use of add-in functions, this may be set to any value, but by convention it should be set to NULL (C/C++ 0L) or 0 (zero).

See Also

FBFreeExpression

FBLoaded Function

See Also

Unit

Fbcalc

Declaration

```
Procedure FBLoaded : boolean;
```

Description

Returns True if the FormulaBuilder DLL (FBCALC.DLL) is loaded in memory. Since the FBCalc import unit links dynamically to the DLL, it may be necessary to see if the DLL is loaded before calls are made to its routines.

See Also

CheckLoadFB
FreeFBuilder
InitFBuilder

FBParseAddConstant Examples

Pascal

```
FBParseAddConstant('sqrt_pi','SQRT(pi)')
FBParseAddConstant('PastDue','DAY( Today() ) > 15')
```

C/C++

```
FBParseAddConstant("sqrt_pi","SQRT(pi)")
FBParseAddConstant("PastDue","DAY( Today() ) > 15")
```

FBParseAddConstant Function

[examples](#) [Constant Handling Functions](#) [DLL Reference](#) [A-Z Function Reference](#)

Pascal

```
Function FBParseAddConstant(handle : HEXPR;name : PChar;expr : PChar);
```

C/C++

```
FBERROR FBAPI FBParseAddConstant(HEXPR handle,LPSTR name,LPSTR expr);
```

VB

```
Declare Function FBParseAddConstant% Lib "FBCALC.DLL" (ByVal handle&,ByVal
name$,ByVal expr$)
```

Description

Create a variable with the name *name*, setting its initial value to the result of the expression *expr*. The new constant takes the type of *expr.*

FBParseAddVariable Function

See Also Example Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBParseAddVariable(handle : HEXPR;name : PChar;expr : PChar);
```

C/C++

```
FBERROR FBAPI FBParseAddVariable(HEXPR handle,LPSTR name,LPSTR expr);
```

VB

```
Declare Function FBParseAddVariable% Lib "FBCALC.DLL" (ByVal handle&,ByVal
name$,ByVal expr$)
```

Description

Create a variable with the name *name*, setting its initial value to the result of the expression *expr*. The new variable takes the type of *expr.*

FBParseAddVariable Examples

```
FBParseAddVariable(handle,'next_week','today() + 7')
FBParseAddVariable(handle,'fullname','Proper(lastname + char(32) +Firstname)')
```

See Also

FBAddVariable

FBPasStringToDate Function

[Utility Functions](#) [DLL Reference](#) [A-Z Function Reference](#)

Pascal

```
Function FBPasStringToDate(const s : string):TFBDate;
```

Description

Converts the Pascal string *s* to a FormulaBuilder date type.

FBPeekVarVB Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

```
Function FBPeekVarVB(handle : HEXPR;vno : integer;vname : pchar;maxlen :
word;Var vtype : integer;Var value : pchar;vallen : word):integer;
```

C/C++

```
FBERROR FBAPI FBPeekVarVB(HEXPR handle,int vno,LPSTR vname,WORD maxlen,LPINT
vtype,LPSTR value,WORD vallen);
```

VB

```
Declare Function FBPeekVarVB% Lib "FBCALC.DLL" (ByVal handle&,ByVal vno%,ByVal
vname$,ByVal maxlen%,ByRef vtype%,ByVal value$,ByVal vallen%)
```

Description

Visual Basic version of FBPeekVariable, since VB does not support unions (variant records in Pascal). It allows access to the name and value of variables by index. *vno* is the index number of the desired variable. The name of the variable is copied to *Vname* , up to *maxlen* characters. up to *vallen* characters of the string representation of variable's value is copied to *value*. vtype is the vtXXX constant describing the type of the variable.

Remarks

Variables accessed through this call are indexed from at 0 to Variablecount - 1

This function returns only those variable handled internally to the engine. Variables handled in programmer code is not visible to this function.

See Also

FBGetVarAsString
FBGetVariableCount
FBPeekVariable

FBPeekVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBPeekVariable(handle : HEXPR;vno : integer;name : pchar;buflen :
word;Var value : TValueRec):integer;
```

C/C++

```
FBERROR FBAPI FBPeekVariable(HEXPR handle,int vno,LPSTR name,WORD
buflen,LPVALUEREC value);
```

Description

Inspect the *vno*th variable in the variable list. The name of the variable is copied to *name*, up to *buflen* characters. *value* is a TValueRec representing the value of the variable.

Remarks

Variables accessed through this call are indexed from at 0 to Variablecount - 1
This function returns only those variable handled internally to the engine. Variables handled in programmer code is not visible to this function.

See Also FBGetVariableCount

See Also

FBGetVariableCount
FBPeekVarVB

FBRTCOMP Unit

The FBRTCOMP unit contains the declarations for TRTTIExpression, the FormulaBuilder class which interacts directly with Delphi's Runtime Type Information (RTTI) system. The following items are declared in the FBRTCOMP unit:

Components

TRTTIExpression

Notes

FormulaBuilder includes a unit FB_RTTI to provide a higher level interface to Delphi's RTTI. This unit depends on routines in FB_RTTI.

FBRegisterFunction Function

See Also Function Handling Routines DLL Reference A-Z Function Reference

Pascal

```
Function FBRegisterFunction(  fname  : pchar;
                              returntype  : byte;
                              params  : pchar;
                              minparms : integer;
                              func   : TCBKExternalFunc):integer;
```

C/C++

```
FBERROR FBAPI FBRegisterFunction(LPSTR fname,BYTE returntype,LPSTR params,int
minparms,TCBKExternalFunc func);
```

Description

Register a function with the FormulaBuilder engine. If the call is successful, the return value is an integer > 100 which the engine uses as a unique ID for the function. This value should be stored for used with the FBUnregisterFunction call. The call will return

EXPR_DUPLICATE_IDENT if the function name is not unique
EXPR_INVALID_FUNCTION if either the function name or function pointer is NULL

Parameter **Description**

fname the name of the function. Note that case is unimportant.

returntype the return type of the function. This must be one of the vtXXX constants.

params a null-terminated string containing a character for each parameter expected by the function according to the following table :

Type	Character
Integer	'I'
String	'S'
Date	'D'
Float	'F'
Boolean	'B'
Any	'A'

FormulaBuilder guarantees that each parameter passed to a callback procedure will be exactly of the type and in the order listed. It uses automatic type conversions for assignment compatible parameters.

minprms The *minparms* parameter tells the parser the minimum number of arguments the function expects. This value can be any value from 0 to the length of the params parameter. The parser will expect no less than *minparms* and no more than *strlen(params)* parameters. If the number of parameters entered by the user are not in this range, the parser will report an error.

This is our means of telling the parser that our function supports a variable number of parameters.

func the callback function. See TCBKExternalFunc for the prototype.

See Also

- FBGetFunctionCount
- FBGetFunctionProto
- FBUnregisterFunction

FBReparseExpression Function

[Expression Initialization and Disposal](#) [DLL Reference](#) [A-Z Function Reference](#)

Pascal

```
Function FBReparseExpression(handle : HEXPR):integer;
```

C/C++

```
FBERROR FBAPI FBReparseExpression(HEXPR handle);
```

VB

```
Declare Function FBReparseExpression% Lib "FBCALC.DLL" (ByVal handle&)
```

Description

Reparses the expression previously set with a call to FBSetExpression. This function is useful in cases where variables/fields are handled externally, and the external data source changes.

FBSetBooleanVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBSetBooleanVariable(handle : HEXPR;vname : pchar;value :
BOOL):integer;
```

C/C++

```
FBERROR FBAPI FBSetBooleanVariable(HEXPR handle,LPSTR vname,BOOL value);
```

VB

```
Declare Function FBSetBooleanVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,ByVal value%)
```

Description

Sets the value of a boolean variable *vname* with the boolean *value*. FBSetBooleanVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a boolean.

See Also

FBSetVarFromString
FBSetVariablePrim

FBSetDateVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBSetDateVariable(handle : HEXPR;vname : pchar;value :
pchar):integer;
```

C/C++

```
FBERROR FBAPI FBSetDateVariable(HEXPR handle,LPSTR vname,LPSTR value);
```

VB

```
Declare Function FBSetDateVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,ByVal value$)
```

Description

Sets the value of a date variable *vname* with the date *value*. FBSetDateVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a date.

See Also

- FBGetDateVariable
- FBGetVariablePrim
- FBSetVarFromString

FBSetExpression Function Call

See Also Expression Initialization and Disposal DLL Reference A-Z Function Reference

Pascal

```
Function FBSetExpression(handle : HEXPR;expr : pchar):integer;
```

C/C++

```
FBERROR FBAPI FBSetExpression(HEXPR handle,LPSTR expr);
```

VB

```
Declare Function FBSetExpression Lib "FBCALC.DLL" (ByVal handle&,ByVal expr$)
```

Description

Initializes the expression with its infix representation. This triggers the parsing phase of the evaluation process.

Example

```
FBSetExpression(hCommission,"[sales->total] * [employee->comrate]");
```

see also

FBClearExpression
FBGetExpression

FBSetFloatVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBSetFloatVariable(handle : HEXPR;vname : pchar;value :
double):float;
```

C/C++

```
FBERROR FBAPI FBSetFloatVariable(HEXPR handle,LPSTR vname,DOUBLE value);
```

VB

```
Declare Function FBSetFloatVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,ByVal value#)
```

Description

Sets the value of a float variable *vname* with the float *value*. FBSetFloatVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not an float.

See Also

[FBGetFloatVariable]
[FBGetVariablePrim]
[FBSetVarFromString]

FBSetIntegerVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBSetIntegerVariable(handle : HEXPR;vname : pchar;value :
longint):integer;
```

C/C++

```
FBERROR FBAPI FBSetIntegerVariable(HEXPR handle,LPSTR vname,LONG value);
```

VB

```
Declare Function FBSetIntegerVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,ByVal value&)
```

Description

Sets the value of a integer variable *vname* with the long integer *value*. FBSetIntegerVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not an integer.

See Also

FBGetIntegerVariable
FBGetVariablePrim
FBSetVarFromString

FBSetStringVariable Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBSetStringVariable(handle : HEXPR;vname : pchar;value :
pchar):integer;
```

C/C++

```
FBERROR FBAPI FBSetStringVariable(HEXPR handle,LPSTR vname,LPSTR value);
```

VB

```
Declare Function FBSetStringVariable% LIB "FBCALC.DLL" (ByVal handle&,ByVal
vname$,ByVal value$)
```

Description

Sets the value of a string variable *vname* with the string *value*. FBSetStringVariable returns EXPR_UNKNOWN_IDENT if the variable does not exist, and EXPR_TYPE_MISMATCH if the variable is not a string.

See Also

FBGetStringVariable
FBGetVariablePrim
FBSetVarFromString

FBSetVarFromString Function

See Also Variable Handling Functions DLL Reference A-Z Function Reference

Pascal

```
Function FBSetVarFromString(handle : HEXPR;name : pchar;value :
pchar):integer;
```

C/C++

```
FBERROR FBAPI FBSetVarFromString(HEXPR handle,LPSTR name,LPSTR value);
```

VB

```
Declare Function FBSetVarFromString% Lib "FBCALC.DLL" (ByVal handle&,ByVal
name$,ByVal value$)
```

Description

Set the value of variable *name* from the null-terminated string *value*. *value* should be the string representation of a **constant** of the same or compatible type as the variable *name*, otherwise a EXPR_TYPE_MISMATCH error is returned.

See Also

FBAddVariable
FBGetVarAsString
FBParseAddVariable

FBSetVariableCallbacks Function

Pascal

```
Function  FBSetVariableCallbacks(handle    : HEXPR;
                             CBKVFind  : TCBKFindVariable;
                             CBKVGetval : TCBKGetVariable;
                             CBKVSetVal :TCBKSetVariable;
                             CBKData    : longint) :integer;
```

C/C++

```
FBERROR FBAPI FBSetVariableCallbacks(HEXPR handle,
                             TCBKFindVariable CBKVFind,
                             TCBKGetVariable  CBKVGetVal,
                             TCBKSetVariable  CBKVSetVal,
                             LONG  CBKData);
```

Description

Register functions to enable external programmer-defined variable processing. Setting callbacks overrides the internal variable handling routines. All variables must be handled externally. An explanation of the parameters follows in the section "External Variable/Field Handling".

FBSetVariablePrim Function

See Also   Variable Handling Functions   DLL Reference   A-Z Function Reference

Pascal

Function FBSetVariablePrim(*handle* : HEXPR;*name* : pchar;*value* : TValueRec):integer;

C/C++

FBERROR FBAPI FBGetVariablePrim(HEXPR *handle*,LPSTR *name*,TVALUEREC *value*);

Description

Sets the value of the variable *name* to *value*. *value*.vtype contains the variable type, with the appropriate variant field containing its value. If the value.vtype field does not match the variable's type, EXPR_TYPE_MISMATCH is returned.

See Also

FBGetVariablePrim

FBSetVarFromString

FBSetXXXVariable Functions

FBStringToDate Function

Utility Routines DLL Reference A-Z Function Reference

Pascal

```
Procedure FBStringToDate(source : TFBString;var date : TFBDate);
```

C/C++

```
void FBAPI FBStringToDate(TFBString source,LPFBDATE date);
```

Description

Converts the FormulaBuilder string *source* to a FB date .

FBStrncpy Function

Utility Routines DLL Reference A-Z Function Reference

Pascal

```
Procedure FBStrncpy(dest : pchar;source : TFBString;maxlen : word);
```

C/C++

```
void FBAPI FBStrncpy(LPSTR dest,TFBString source,WORD maxlen);
```

Description

Copy up to *maxlen* characters from the FormulaBuilder string source to the null-terminated string *dest.*

FBUnregisterFunction Function

See Also Function Handling Routines DLL Reference A-Z Function Reference

Pascal

```
Function FBUnregisterFunction(fnId : integer):integer;
```

C/C++

```
FBERROR FBAPI FBUnregisterFunction(int fnID);
```

Description

Unregisters a programmer-defined function registered with the FBRegisterFunction call. this call is necessary when multiple clients use the FormulaBuilder DLL and at least one application calls FBRegisterFunction with a function implemented in the application itself (as opposed to a DLL). The explanation follows :

The internal function table is a global DLL resource. All functions registered with the DLL are visible to all other DLL clients. If the process containing the actual callback implementation exits without Unregistering, the function table still maintains a stale pointer to the callback. If the FormulaBuilder attempts to call this routine, a GPF will most likely occur.

For this reason, **it is best to place all programmer defined functions in a separate DLL** which registers its functions with FormulaBuilder at load-time. This way the functions remain available independent of the applications using the FormulaBuilder DLL.

NOTE: this information pertains primarily to registered functions located in an application, as opposed to a DLL.

see also

FBGetFunctionProto
FBRegisterFunction

FB_RTTI Unit

The FB_RTTI unit contains the declarations for TInstanceProperty and associated routines to ease the task of interfacing with Delphi Runtime Type Information (RTTI) system. The following items are declared in the FB_RTTI unit:

Objects

TInstanceProperty
RTTIError

Routines

ClassAssignmentCompatible
DescendsFrom
FBGetEnumValue
FindPropInfo
FreePropertyData
GetComponentProperties
GetProperties
GetPropFromPath
GetRTTIErrorText
StringSetToInt

The following were provided for the use of the FBRTCOMP unit as convenient ways of generating appropriate RTTI Errors :

```
procedure PropValueError;
procedure PropertyNotFound;
procedure PropPathError;
Procedure InvalidPropertyError;
Procedure PropReadOnlyError;
```

See the entry on RTTIError for more information.

FBlpzToDate Function

See Also Utility Routines DLL Reference A-Z Function Reference

Pascal

```
Procedure FBlpzToDate(source : pchar;var date : TFBDate);
```

C/C++

```
void FBAPI FBlpzToDate(LPSTR source,LPFBDATE date);
```

Description

Convert the null-terminated string *source* to a FormulaBuilder Date type. *Source* must be valid date string. Note that the curly braces denoting date/time constants are not needed.

See Also

FBStringToDate

FIND Function

See Also String Functions A-Z Function Reference

Description

Returns the position of a substring within another string.

Syntax

FIND(*search,source*<,*start*>)

Remarks

the position of the string *search* within *source. Start* optionally specifies where in *source* to begin the search.

See Also

LENGTH
MID

FIRST Function

See Also String Functions A-Z Function Reference

Description

Returns a specified number of characters from the beginning of a string.

Syntax

FIRST(*count*, s)

Returns the first *count* characters of string expression *s*. If *count* is greater than the length of s, the value of *s* is returned.

See Also

LAST
LENGTH
MID

FLOOR Function

See Also Math/Trig Functions A-Z Function Reference

Description

Rounds a number down to the nearest whole number.

Syntax

FLOOR(*x*)

x is any number

See Also

CEILING
INT
ROUND

FRAC Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the fractional portion of a number.

Syntax

FRAC(x)

X is any number.

See Also

CEILING
FLOOR
INT
ROUND

FV Function

See Also Financial Functions A-Z Function Reference

Description

Calculates the future value of an investment with a specified present value based on a series of equal payments *pmt*, earning interest rate *Rate* over *Nper* payment periods

Syntax

FV(*Pmt,Rate,NPer*<,*Type*>)

Parameter	Description
Pmt	a numeric value representing the amount of the periodic payment.
Rate	a numeric value greater than -1, representing the periodic interest rate.
Nper	the number of payment periods
Type	an optional number denoting the type of annuity. 0 (zero) for ordinary annuity, 1 for an annuity due. The default is 0.

Remarks

Rate and *NPer* should be expressed in the same increment. For example, if you are calculating a weekly payment, enter the *Rate* and *NPer* in weekly increments.

See Also

FVAL	PMT
NPER	PV
PAYMT	RATE

FVAL Function

See Also Financial Functions A-Z Function Reference

Description

Calculates the future value of an investment with a specified present value based on a series of equal payments *pmt*, earning interest rate *Rate* over *Nper* payment periods

Syntax

FVAL(*Rate, Nper, Pmt*<, *Pv, Ptype*>)

Parameter	Description
Rate	a numeric value greater than -1, representing the periodic interest rate.
NPer	the number of payment periods
Pmt	a numeric value representing the amount of the periodic payment.
PV	the present value of the annuity
Ptype	0 if the investment is an ordinary annuity or 1 if it is an annuity due

Remarks

The *Ptype* and *PV* arguments are optional. If they are ommitted, their values are taken to be zero. *Rate* and *NPer* should be expressed in the same increment. For example, if you are calculating a weekly payment, enter the *Rate* and *NPer* in weekly increments.

See Also

FV
NPER
PAYMT
PMT
PV
RATE

FieldDataType Function

Unit

FBDBComp

Declaration

```
Function FieldDatatype(const f : TField):datatypes;
```

Description

The FieldDatatype function converts a dataset field type to its equivalent FormulaBuilder type (see the vtXXX constants).

Fields
A field in FB terminology is simply a variable delimited by square brackets. When a field is encountered, the text between the brackets is passed to the CBKFindVariable callback routine (DLL) or the OnFindVariable event of the Delphi component classes. This allows the flexibility of dealing with variables that do not fit the standard naming convention, e.g. database field names, spreadsheet cell definitions

example

[parts->partno]   [$A1]   [Aug95:R1C10]

The programmer has the responsibility of identifying and providing values for fields when needed by the parser. Otherwise, fields are handled in the same manner as variables.

FilterHandle Property

Applies To

TDSFilter

Declaration

```
Property FilterHandle : hDbiFilter;
```

Description

The FilterHandle property returns the read only BDE filter handle corresponding to the TDSFilter instance. This FilterHandle is provided should you need to use BDE level calls. See the BDE documentation for further details.

Financial Functions

FormulaBuilder provides the following financial functions to deal with the areas of annuities, depreciation, capital budgeting and depreciation. The annuity functions assume by default that all investments are ordinary annuities. An annuity is an investment in which a series of equal payments are made. An ordinary annuity is an annuity in which a payment is made at the end of each time period. An annuity due is an annuity in which a payment is made at the beginning of each time period.

The standard arguments for the annuity functions are as follows :

Parameter	Description
	Rate interest rate, greater than -1, representing the periodic interest rate. An interest rate of 15 percent would be represented by (15/100 = .15) for example.
Nper	number of payment periods. Should be an integer greater than 1.
Pv	the present value of the annuity. When this argument is optional, its assumed default value is 0.
Pmt	a numeric value representing the amount of periodic payment
Fv	the future value of the annuity. When this argument is optional, its assumed default value is 0.
Type	0 (zero) for ordinary annuity, 1 for an annuity due. In most functions this parameter is optional and the default is 0, meaning that payments are assumed to be at the end of the period.

Please Note ! Make sure that you are consistent about the units used in specifying Rate and NPer. For example, for a 12% annual rate loan with monthly payments (Nper = 12 per year) and 3 years duration, Rate would be 1% (0.01) and Nper would be 12 * 3 = 36.

Function Categories

Annuities

FV	NPER	PV
FVAL	PAYMT	PVAL
IPAYMTPMT	TERM	
IRATE	PPAYMT	

Capital Budgeting

IRR Calculates the Internal Rate of Return on an investment.
NPV Determines the Net Present Value of a series of cash flows.

Depreciation

DB Calculates the depreciation allowance for an asset using the fixed-declining balance method.
DDB Calculates the depreciation allowance for an asset using the double-declining balance method.
SLN Uses the Straight Line method to calculate the depreciation of an asset.
SYD Uses the Sum-of the-Years-Digits depreciation method to calculate the amount of depreciation in one period.

Single-sum Compounding

CTERM Calculates the number of compounding periods it takes for the present value of an investment to grow to a future value at a fixed rate of interest per period.
RATE Returns the interest rate per period of an annuity

FindPropInfo Function

Unit

FB_RTTI

Declaration

```
Function FindPropInfo(Root                  : TObject;
                      Const PropName        : String;
                      Const Kinds           : TTypeKinds;
                      var   Instance        : TObject;
                      var   APropInfo       : PPropInfo):boolean;
```

Description

Performs a recursive search from ROOT downward to see if PropName exists as one of the published properties of an object which is a property of root, or if root is a component, contained in its component list. Returns true if the property was found.

Parameter	Description
Root	The starting point of the search
PropName	The name of property were interested in finding

Kinds	A set of the types of properties were interested in. TTypeKinds is defined as follows :
	
	`type`
	`  TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,tkString, tkSet, tkClass, tkMethod);`
	`  TTypeKinds = set of TTypeKind;`

Instance	The actual object in which the property was found
APropInfo	A pointer to the RTTI property information record for the located property. See TYPINFO.INT for more details.

See Also

AsString

ReturnType

Flow of Operation

Using FormulaBuilder expressions in applications follow a simple procedure

1. Add optional constants that will be available to all expressions (constants are global)
2. Initialize the expression
3. Add the variables and constants that will be needed for the formula(s) that will be assigned to the expression instance. Note that variables and constants to be used in expressions MUST be added before the formula using them is assigned to the expression instance.
4. Set the expression text
5. Set the value of variables
6. Evaluate the expression
7. While variables change goto step 5
8. Free the expression instance. Variables are local to expressions and will be freed automatically.

Formula Property

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Property Formula : String;
```

Description

Reads and sets the string expression to be evaluated. Setting this property will automatically invoke the parsing process. An error will be generated if syntactical or other errors are detected in the expression. Reading this property will return the original string expression.

Example

```
PiFunc.Formula := '22/7';
Panel1.Caption := PiFunc.AsString;{Panel1.Caption = '3.142.....' }
Panel2.Caption := PiFunc.Formula; {Panel2.Caption = '22/7' }
```

Formula Property Example

We can initialize the expression instance with the string to be evaluated with code such as the following ::

```
Expression1 := TExpression.Create(NIL);
Expression1.Formula := 'PAYMT(0.15,12,15000,35000,1)';
Panel1.Caption := Expression1.AsString;
```

See Also

Lines Property
StrFormula Property

FreeFBuilder Function

see also

Unit

FBCALC

Declaration

```
Procedure FreeFBuilder;
```

Description

The FreeFBuilder function decreases the FBCalc import unit's internal reference count. If the count reaches 0 (zero), the FormulaBuilder DLL is unloaded from memory. This has no effect if the DLL is not already loaded.

Although the FormulaBuilder Delphi components call this automatically in their destructors, this call may still be necessary if direct DLL calls are made outside of component (for example FBRegisterFunction, FBCreateString, and FBGetFunctionCount)

Note

As a matter of good housekeeping, please ensure that a call to InitFBuilder is matched to a call to FreeFBuilder.

see also

CheckLoadFB
FBLoaded
InitFBuilder

FreePropertyData Procedure

Unit
FB_RTTI

Declaration
```
Procedure FreePropertyData( AList : TStrings );
```

Description
Disposes of the data allocated and assigned to the Objects array property of AList in the GetProperties or GetComponentProperties call.

FreeVariable Method

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Procedure FreeVariable(Const name : string);
```

Description

Dispose of the variable *name*.Free all memory associated with the variable *name* and removes it from the expression's variable list. The Reparse method is automatically called to ensure that the expression remains valid.

See Also

Clear Method
FreeVariableList Property

FreeVariableList Method

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Procedure FreeVariableList;
```

Description

Disposes of all variables associated with the current instance.

See Also Clear

See Also

Clear Method

FreeVariable Method

Freeing The Expression

If you have used the non-component version of TExpression, or have added the component version manually, code similar to the following should appear in the FormDestroy method :

```
 Procedure TFORM1.FormDestroy(Sender: TObject)
Begin
  { Cleanup code }
   EXPRESSION.Free
  { Other cleanup code }
End;
```

VB : Freeing the Expression

Expressions that have been initialized with a call to FBInitExpression, must be dispose of with a call to FBFreeExpression

Example

```
Sub Form_Unload (Cancel As Integer)
    status% = FBFreeExpression%(handle&)
End Sub
```

Function Handling Routines

FormulaBuilder provides calls to register, unregister and query installed formula functions.

FunctionCount Property

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Property FunctionCount : word;
```

Description

Read Only. Returns the number of functions (both internal and programmer-defined) registered with FormulaBuilder

See Also

FBGetFunctionProto
FBRegisterFunction
FBUnregisterFunction

Functions

Functions take input values (*arguments or parameters*) and return a result, which may be string, numeric, date or boolean. Function names follow the naming convention for identifiers

The format of a function is as follows

FUNCTION(*argument1,argument2*,...)

> ***FUNCTION*** is the function name. Function names are not case-sensitive
> Functions arguments are enclosed in parentheses, even for functions with no arguments.
> Multiple parameters are separated by commas.
> FormulaBuilder supports functions with optional arguments. Elements surrounded by angle brackets in the function listings (< and >) are optional.

```
NPER(Pmt,Rate,Fv<,Type,Pv>)
```

If you omit an optional argument, a default value is assumed for the argument.

> Functions may be nested arbitrarily. For example :

```
AVG( SUM(SIN(Pi),ABS(10 * COS(X)),10, e), ASEC(X), .215, 10)
```

GetComponentProperties Procedure

See Also

Unit

FB_RTTI

Declaration

```
Procedure GetComponentProperties(AObject      : TComponent;
                                 TypeKinds    : TTypeKinds;
                                 AList        : TStrings;
                                 iIndentLevel : integer);
```

Description

Retrieves a recursive list of all named published properties and contained components of AObject. The list is specifically formatted for outline use, and the AList.Objects[n] item contains a TInstanceProperty object which encapsulates the property corresponding to the AList.Strings[n] string. This procedure is similar to GetProperties, but is specifically for components.

Parameter	**Description**
AObject	The top-level object in the heirarchy. The names of all published properties and properties of contained objects will be loaded into AList

TypeKinds	Restricts the types of properties included in AList. TypeKinds is defined in TYPINFO.INT as follows :
	`type`
	`  TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,tkString, tkSet, tkClass, tkMethod);`
	`TTypeKinds = set of TTypeKind;`

AList	The list containing the names of the properties. This may be assigned to the Lines property of a TOutline to present a tree-view of the property heirarchy rooted at AObject. The Objects[] array property contains a TInstanceProperty object for the corresponding property named in the Strings[] property.
iIndentLevel	The beginning indent level for the property name list.

Remarks

Collecting the data for AList may involve quite a bit of recursion, and since Delphi Classes are references, which may contain published references to other classes, this routine may cause some delay in processing. It does yield processing periodically, but delays may still be noticeable.

Because of the levels of recursion involved and the possible amount of data collected, it is advisable to set AObject to only simple or moderately complex objects/components

Since this routine allocates TInstanceProperty instances for each matching property it encounters, you should ensure that FreePropertyData is called on AList (or the outline Lines property to which it was assigned) to free the allocated memory.

See Also

GetProperties

GetErrorString Function

Unit

FBDBComp

Declaration

```
Function GetErrorString(const ecode : integer):String;
```

Description

Returns a string describing the FormulaBuilder error code. This routine works for the data-aware component related errors as well. Refer to the EXPR_XXX constants for additional information.

GetFunctionPrototypes Function

Unit
FBComp

Declaration

```
Function getFunctionPrototypes(useResultType : boolean):TStringList;
```

Description
The GetFunctionPrototypes function returns a stringlist containing a string prototype for each function registered with FormulaBuilder. The format of the string is as follows :

FUNCNAME(type1,type2<,...typeN)<:returntype>

FUNCNAME	the name of the function
type1..typeN	characters describing the type of parameter required

Type	**Character**
Integer	'I'
String	'S'
Date	'D'
Float	'F'
Boolean	'B'
Any	'A'

returntype	a string describing the return type of the function. Whether or not this appears depends on the value of the *useResultType* parameter.

GetPropFromPath Function

Unit

FB_RTTI

Declaration

```
Function GetPropFromPath(Root : TObject; PropPath : string;var Instance :
TObject) : PPRopInfo;
```

Description

Returns Property information for a property given its Path from *Root*. The object instance to which the property belongs is returned in *Instance.*

Property Paths

If Root is set to an instance of a TForm, valid property paths would be

```
'Caption'
'Font.Name'
```

Note also that you also have (recursive) access to the properties of named components contained in the Components array of components. For instance, given the same form which contains a TDataSource named CustomerSource, we could use the following property path:

```
'CustomerSource.Dataset.Tablename'
```

If the Root property were set to *Application*, and our form were named *CustomerForm*, we would write the properties as follows :

```
'CustomerForm.Caption'
'CustomerForm.Font.Name'
'CustomerForm.CustomerSource.Dataset.Tablename'
```

GetProperties Procedure

See Also

Unit

FB_RTTI

Declaration

```
Procedure GetProperties(AObj   : TObject; TypeKinds    : TTypeKinds;
                        AList : TStrings;iIndentLevel : Integer);
```

Description

Please see the description for GetComponentProperties.

See Also

GetComponentProperties

GetRTTIErrorText Function

Unit

FB_RTTI

Declaration

```
Function GetRTTIErrorText(ecode : integer):string;
```

Description

Returns a text string corresponding to a FB generated RTTI error. See the RTTIError topic for more information.

GetVarPtr Examples

This code assumes we have an initialized TExpression instance named Expression1, and a TForm1 with the method AddVariables :

```
Procedure TForm1.AddVariables;
begin
  with Expression1 do
  begin
    { Note that the variables were added before the expression }
    { involving them was assigned to the Formula property }
    AddVariable('Name',vtSTRING);
    AddVariable('BirthDate',vtDATE);
    AddVariable('Married',vtBOOLEAN);
    AddVariable('Children',vtInteger);
    AddVariable('Salary',vtFLOAT);
    AddVariable('PIN',vtFLOAT);
    Formula := 'PIN := Length(Name) + DAY(BirthDate) -
               (Sqrt(Age) * Salary) * IIF(Married,Kids,0)';
  end;
end; { AddVariables }
```

Example 1

```
 Procedure TForm1.GetVarPtr_SetVars;
var Salary  : ^Double;
   {name    : PString;  string vars should not be accessed directly }
    DOB     : ^TDateTime;
    married : ^Boolean;
    kids    : ^longint;
    vtype   : byte;

begin
  With expression do
  begin
    GetVarPtr('Name',vtype,pointer(namePtr));
    GetVarPtr('BirthDate',vtype,pointer(DOBptr));
    GetVarPtr('Married',vtype,pointer(MarriedPtr));
    GetVarPtr('Children',vtype,pointer(childrenPtr));
    GetVarPtr('Salary',vtype,Pointer(SalaryPtr));
  end;
  SalaryPtr^   := Person.Salary;
  ChildrenPtr^ := Person.Children;
  MarriedPtr^  := Person.Married;
  DOBPtr       := Person.BirthDate;
  { Name should not be directly accessed, so well use the
    stringValue property}
  StringValues['Name'] := Person.Name;
end;
```

Example 2

Beyond convenience, variable access using GetVarPtr is beneficial in expressions that need to be calculated over many iterations. Here’s an example using a contrived expression :

```
Procedure TForm1.GetvarPtrLoop;
var lcv : longint;
      xptr, yptr, zptr : ^Double;
```

```
begin
     With Expression do
     begin
       AddVariable('X',vtFLOAT);
       AddVariable('Y',vtFLOAT);
       AddVariable('Z',vtFLOAT);
       GetVarPtr('X',vtype,pointer(xptr));
       GetVarPtr('Y',vtype,pointer(yptr));
       GetVarPtr('Z',vtype,pointer(zptr));
       Formula := ' Sin(X + Y) -  Cos(X - Y)  * Z'; { or any complex formula }
       for lcv := 1 to 2500 do
       begin
            x^ := Sqrt(Z^) + (LCV mod 10);
            y^ := random(X^);
            z^ := AsFloat;
       end;
   end;
end;
```

For large iterations the GetVarPtr method may save considerable processing overhead as compared to the other variable access methods.

GetVarPtr Method

See Also

Applies to

All FormulaBuilder Components

Declaration

Procedure GetVarPtr(**Const** *name* : string;**var** *vtype* : byte;**var** *value* : pointer);

Description

Returns a pointer to the data for the variable *name* that was added with a call to the AddVariable or ParseAddVarible method. This procedure is a wrapper around the FBGetVarPtr function call. This method is valuable in cases where you may need to recalculate the formula for numerous values of a variable. It avoids the overhead of a call to the StringValues, Variables, and VariableList properties for each iteration. The *vtype* parameter is the vtXXX constant describing the type of the variable. Note that for variables of type vtSTRING, the actual string pointer is returned, not a pointer to the pointer.

VERY IMPORTANT

if the variable is of type vtSTRING, it is very important not to alter the length of the string pointed to by the value pointer.

See Also

StringValues
Variables
VariableList

Getting And Setting Variable Values

To get values from our formula for various inputs, we must have access to the values of our variables. We do so using the various variable handling methods of the TExpression class.

The Variables Property Example

The Variables Array property gives us read/write access to variables as TValueRec types

The StringValues Property Example

The StringValues array property provides read/write access to the string representation of variable values. It is indexed by the name of the desired variable.

The VariableList Property Example

Variables can also be accessed by the VariableList Property.

The GetVarPtr Method Examples

GetVarPtr retrieves a pointer to the data for the variable name that was added with a call to the AddVariable method. It provides the most efficient means of accessing a variable.

Getting Expression Results

Setting the text expression of a TExpression does not automatically cause the expression to be evaluated. The following method and properties are provided to obtain the results of an expression for the current variable set.

The EvaluatePrim Method evaluates the text expression set with the Formula, StrFormula and Lines properties and returns the result in a TValueRec structure.

The following Properties call EvaluatePrim directly, so reading the values of these properties causes the expression to be recalculated.

Property	Returns
AsString	the result of the expression as a string, regardless of its return type
AsBoolean	the boolean result of the expression
AsDate	the date result of an expression
AsFloat	the floating point result of the expression
AsInteger	the integer (longint) result of the expression
StringResult	the string result of the expression

Getting the Variable Count

The VariableCount property returns a count of all variables added to the TExpression instance. In our example, VariableCount is equal to 5. A single TExpression or descendant instance can own up to 16,000 variables, memory permitting. Parsing may be slower for a large number of variables, but there is no performance penalty in the actual evaluation process.

HEXPR Type

Pascal

Type HEXPR = longint;

C/C++

typedef LONG HEXPR;

Description

HEXPR is the signed 32 bit integer handle type returned by FBInitExpression and used to uniquely identify each expression for subsequent calls to the engine.

HOUR Function

See Also		Date/Time Functions		A-Z Function Reference

Description

Returns the hour component of a date/time serial number.

Syntax

HOUR(*datetime_serial*)

datetime_serial is the date/time value from which to derive the hour. The fractional portion represents the fraction of the day.

Remarks

The hour is returned in military (24 hour) format, from 0 (representing 12:00 am) to 23 (representing 11:00 p.m.)

See Also

MINUTE
SECOND

Handle Property

see also

Applies to

All FormulaBuilder Components

Declaration

`**Property** Handle : HEXPR;`

Description

Returns the handle associated with the expression. This is returned when an expression is initialized by a call to FBInitExpression. This property is provided to allow direct calls to the FormulaBuilder DLL.

see also

FBInitExpression

Handling Expression Errors

When you pass the text form of an expression to an instance of TExpression, the text is parsed and translated into a tokenized intermediate representation of that expression. The expression is stored in both its text form and in its tokenized form.There are many errors which may occur either in this phase or when the expression is finally evaluated.

The UseExceptions Property

You can determine what happens in the event of an error by setting the UseExceptions Property. For instance, if you would like all errors encountered in either parsing or evaluating the expression to be returned as exceptions, set the UseExceptions Property to TRUE after constructing your TExpression instance :

```
Expression := TExpression.Create(NIL);
Expression.UseExceptions := TRUE;
Try
  Try
      Expression.Formula := '"First & "+10';
  Except
      on e: EFBError do
      begin
             MessageDlg('FB Error #'+Inttostr(E.Errcode),
                          mtError,[mbOk],0);
      end;
  End;
Finally
  Expression.Free;
end;
```

Status And StatusText

Note that the default state of the UseExceptions property is FALSE. In the FALSE state, all errors are returned as an integer in the Status property. Refer to the entry under EXPR_XXX Constants for a list and explanations of possible error codes.

```
Expression := TExpression.Create(NIL);
Expression.Formula := ' "First and " + 10 ';
if Expression.Status <> EXPR_SUCCESS then
MessageDlg(' Error code +
               IntToStr(Expression.Status),mtError,[mbOk],0);
```

For your convenience, the StatusText Method can be used to get a text representation of the error which occurred.

```
if Expression.Status <> EXPR_SUCCESS then
    MessageDlg(Expression.StatusText,mtError,[mbOk],0);
```

In the event that the parser detects errors in the text expression, the text and intermediate representation are cleared. You can verify this by testing the value of the IsNull property. You will have to enter a valid expression before any evaluation can occur.

Handling Function Callback Errors

In case an error occurs in the callback procedure, the *errcode* parameter of the TCBKExternalFunc may be used to notify the expression of its occurance. Upon entry to a function implementation callback, the *errcode* parameter is set to EXPR_SUCCESS, therefore it only needs to be modified in the event of an error. If it is set to any value other than EXPR_SUCCESS, the evaluation process halts and the value of *errcode* is returned. This value may be accessed using the Status property of TExpression.

Example

NOTE - The programmer should try to trap all exceptions which may occur in the callback, and return an error code to describe the function.

IIF Function

See Also   String Functions   A-Z Function Reference

Description

Returns one of two values based on a true/false condition.

Syntax

IIF(*condition,value1,value2*)

Condition must be a value or expression which evaluates to a boolean (TRUE/FALSE) value. if *condition* evaluates to TRUE, *value1* is returned otherwise return *value2* is returned. *Value1* and *Value2* may be of any type.

Example

```
IIF( (BALANCE > 0) AND (TODAY() - LASTPAYMENTDATE > 30),"Delinquent","Uptodate")
IIF([SHIFT->HOURS] > 40,1.5,1.0) * [EMPLOYEE->RATE]
```

See Also

CHOOSE

INSERT Function

[String Functions] [A-Z Function Reference]

Description

Insert a string into another at a specified position.

Syntax

INSERT(*str,source,p*)

Returns the string *source* with the string *str* inserted at position *p.*

INT Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the integer portion of a number.

Syntax

INT(*number*)

number is any number.

See Also

- CEILING
- FLOOR
- FRAC
- ROUND

IPAYMT Function

See Also Financial Functions A-Z Function Reference

Description

For a given period and a fixed interest rate, IPAYMT calculates the portion of a payment amount that is interest.

Syntax

IPAYMT(*Rate,Per,Nper,Pv*<,*FV,Type*>)

Parameter	Description
Rate	the fixed periodic interest rate. *Rate* must be greater than -1.
Per	the period for which you wish to find the interest payment. This value must be between 1 and *Nper*.
NPer	the total number of payment periods for the annuity.
PV	a number representing the amount borrowed.
FV	a numeric value representing the future value of the investment.
Type	is a number specifying when payments are due. 0 (zero) indicates an ordinary annuity, whereas 1 specifies an annuity due

Both *FV* and *Type* are assumed to be 0 if ommitted.

See Also

PAYMT
PMT
PPAYMT

IRATE Function

See Also Financial Functions A-Z Function Reference

Description

Calculates the periodic interest rate of an annuity. You can use it to determine the interest rate necessary for an investment to grow to a future value over a specified number of periods.

Syntax

IRATE(*Nper,Pmt,Pv,<FV,Type>*)

Argument	Description
NPer	a numeric value > 0 representing the number of periods of the investment.
Pmt	a numeric value representing the amount of the periodic payment.
PV	the current value of the investment.
FV	the future value of the investment
Type	a numeric value equal to 0 (zero) if the annuity is an ordinary annuity.

See Also

RATE

IRR Function

[See Also](#) [Financial Functions](#) [A-Z Function Reference](#)

Description

Returns the internal rate of return on an investment. The internal rate of return is the percentage at which the present value of an expected series of cash flows equals the the present value of the initial investment

Syntax

IRR(*Guess,List*)

Guess is your estimate of what the answer will be
List is a list of up to 15 numeric values

Generally the first number in list is negative, indicating the initial payment or investment. The amounts in *List* are assumed to have been received at regular intervals, with negative amounts being considered as outflows and positive values being considered as inflows.

See Also

NPV

RATE

ISEVEN Function

See Also Math/Trig Function A-Z Function Reference

Description

Returns TRUE if number is even, or FALSE if number is ODD.

Syntax

ISEVEN(number)

Number is the number to test. If *Number* is not an integer, it is truncated before it is tested

See Also

ISODD

ISODD Function

See Also Math/Trig Function A-Z Function Reference

Description

Returns TRUE if number is odd, or FALSE if number is even.

Syntax

ISODD(*Number*)

Number is the number to test. If *Number* is not an integer, it is truncated before it is tested

See Also

ISEVEN

Implementing Functions With Variable Parameter Lists

Occasionally, it is useful to have functions where the number of parameters is not fixed. Statistical functions (SUM, AVG, etc) for example take a varying number of parameters. This feature is especially useful for functions which have default parameter values.

Example 1
Example 2

Important Preliminary Issues For Delphi Users

Issues with Dynamic Linking

FormulaBuilder is based on a dynamically linked DLL, which is loaded and unloaded on demand. Since loading is explicit, it is necessary to ensure that the engine is loaded before attempts are made to access its exported functions. Components handle this process transparently, but it is the programmer's responsibility to ensure that the DLL is loaded for calls not made in the scope of a component method. Please refer to the topics listed in the section on the FBCALC unit.

External Functions, Callbacks and Exceptions

All FormulaBuilder callbacks (and the corresponding component events) have an error code parameter which the programmer may use to signal an abnormal condition. While it is standard practice in Delphi to use exceptions to handle out-of-the-ordinary conditions, you should ensure that all exceptions which may occur within FormulaBuilder callbacks and events are trapped and returned in the error code parameter. Components whose UseExceptions property is set to TRUE will in turn generate an appropriate exception once the callback/event returns.

Because objects (including exceptions) cannot be passed across the EXE-DLL boundary, the DLL will have no knowledge of the occurence of the EXE generated exception, and this may lead to an inconsistent expression state in the DLL.

InitFBuilder Function

see also

Unit

FBCALC

Declaration

```
Procedure InitFBuilder;
```

Description

Since the FBCalc import unit dynamically links to the FormulaBuilder DLL, it is necessary to ensure that the DLL is loaded before we access any of its routines. The InitFBuilder function checks to see if the DLL is already loaded. If it is, an internal reference count is incremented. If the engine is not loaded, it is dynamically loaded and the reference count set to 1.

Although the FormulaBuilder Delphi components call this automatically, this call may still be necessary if direct DLL calls are made outside of component (for example FBRegisterFunction, FBCreateString, and FBGetFunctionCount)

Note

As a matter of good housekeeping, please ensure that a call to InitFBuilder is matched to a call to FreeFBuilder.

see also

CheckLoadFB
FBLoaded
FreeFBuilder

Initializing The Expression

see also

If you have added the TExpression instance to your form from the component palette, the object is initialized automatically. If you chose to work non-visually, use the following code to initialize the FORM1.EXPRESSION1 instance of TExpression. Any other instance of TExpression must be initialized in a similar fashion.

```
Procedure TFORM1.FormCreate(Sender: TObject)
 Begin
    {Some initialization code ...}
    EXPRESSION := TExpression.Create(Self);
    {Other Initialization code ...}
 End;
```

VB : Initializing The Expression

Expressions are intialized with a call to FBInitExpression. Any number of expressions (limited by memory) may be allocated.

To use FormulaBuilder expressions

declare a variable (handle& in our examples) of type long with the scope appropriate to your project

Initialize the expression with a call to FBInitExpression

Example

```
Sub Form_Load ()
 handle& = FBInitExpression&(0)
End Sub
```

See Also

Freeing The Expression

Installation

INSTALLATION

There are no special restrictions as to the installation of Formula Builder, except that the DLL must reside in the Windows path.

Installing The Components

To install the FormulaBuilder components to the Delphi Component Palette :

Copy the following files to the LIB directory of your main Delphi directory.

Filename	**Description**
FBCALC.PAS	The FomulaBuilder DLL import unit for Delphi
FBCOMP.DCU	The unit defining the TExpression component
FBDBCOMP.DCU	The unit defining the FormulaBuilder Data-Aware components
FBRTCOMP.DCU	FormulaBuilder RTTI-Aware component
FBREG.PAS	Delphi Registration unit for FormulaBuilder
FBREG.DCR	Delphi Palette bitmaps

Alternately, you may leave these in the FormulaBuilder directory and add this to you Library search path

Copy the file FBCALC.DLL to your \WINDOWS\SYSTEM directory or a directory on the Windows search path.

Copy the file FBUILDER.KWF to the HELP subdirectory of your main Delphi directory.

Copy the file FBUILDER.HLP to the BIN subdirectory of your main Delphi directory.

Run the HELPINST program, located in the Delphi Program Group to integrate thel FBUILDER.KWF keyword file into the DELPHI multihelp system.

Start DELPHI and choose 'Install Components...' from the options menu.

Press the 'Add...' Button. to open The Add Module dialog box.

Click the Browse button to open the Add Module file selection dialog box. Select the full drive and directory path of the FBREG.PAS file as copied in step 2.

Press OK. The unit name FBREG will occur in the Installed Components list box of the Install Components dialog box.

Press the OK button of the Install Components dialog box to install the components.   The components will appear on a Component Palette page labeled FBuilder.

Installing New Functions

One of FormulaBuilder's greatest features is the ability to extend the engine by dynamically adding functions which become available to the end-user at runtime. All external function are implemented via callbacks of type TCBKExternalFunc.

We will show how to install functions by example.
Delphi Example
C/C++ Example

IMPORTANT ! If the registered function is defined in the application (as opposed to a DLL) and more than 1 client is likely to use the DLL concurrently, the value returned from FBRegisterFunction should be stored and used with the corresponding FBUnregisterFunction call when the function is no longer needed:
This is necessary because the internal function table is a global DLL resource. All functions registered with the DLL are visible to all other DLL clients. If the process containing the actual callback implementation exits without Unregistering, the function table still maintains a stale pointer to the callback. If the DLL attempts to call this procedure, an error will occur.

Its is therefore highly recommended that external functions be implemented in DLLs.

Delphi Users

For a more thorough discussion of this subject, see the section Using FormulaBuilder with Delphi.

Instance Property

Applies To

TInstanceProperty

Declaration

```
Property Instance : TObject;
```

Description

Read/write the object instance to which the property named Propname applies. If a published property named Propname does not exist for Instance, an exception is raised.

See Also

[CreateFull](#)
[CreateFromPath](#)
[CreateFromSearch](#)
[Instance](#)
[Propname](#)

See Also

AsString Property
EvaluatePrim Method

International Issues

Date/Time Values

Valid date/time values are dependent on the International Settings in the Windows Control Panel. FormulaBuilder respects the installed language drivers.

Message Strings

Certain FormulaBuilder functions (date/time functions in particular) use text which in general is human-language dependent. In the current release, the DLL returns only English text for such strings. All text strings used by FormulaBuilder (function names, return values and messages) reside in String Table Resources in the DLL. Those wishing to translate FormulaBuilder resources to another language should do so by using a resource editor. Most Windows development tools bundle a resource editor as a part of the package.

The string tables contains the following text:

- Error Message Text
- Function Names. If a function name is made blank or deleted, it is not added to the engine's symbol table, and will not be available to users.
- Short strings for each month of the year (Jan, Feb, Mar through Dec)
- Long strings for the months of the year (January, February, through December)
- Short strings for the days of the week (Mon, Tue, Wed, through Sun)
- Long string containing seven letters corresponding to the week days (Sunday through Saturday)

Changing these strings to the appropriate language will cause FormulaBuilder to return the language specific text for the functions whose values reside in the string resources.

Please make sure that you retain a backup copy of the DLL in case of difficulties before attempting to modify its resources.

Introduction

Welcome to **FormulaBuilder**, the most powerful expression evaluation engine available for any Windows development tool capable of calling a Dynamic Link Library (DLL).   FormulaBuilder (FB) has the versatility and power to deal with the even the most complex expressions.

Expressions of arbitrary complexity

Expression text may be up to 32K in length (subject to memory constraints), with unlimited nesting of functions and parentheses.

Mixed expression parsing.

Boolean, String, Longint, Date/Time as well as floating-point expressions are supported..

Multi-parameter functions.

Most other expression parsers restrict functions to a single floating point parameter. FormulaBuilder functions may contain as many as 16 parameters, each being of any type supported by the engine. Each parameter is typechecked for validity during the parsing process. An "Any" type is supported for parameters whose types cannot be pre-determined.

Functions with variable parameters lists.

This allows the construction of expressions with functions such as, for example

```
MID('Test',2,1)    =  'e'
MID('Test',3)      =  'tes'
MAX(1,2,3,4,5,AVG(4,Cos(PI)),7) = 7
MAX(1,2,3)                      = 3
CHOOSE(3,"String",10 * Rand(10),TRUE,Today())    = TRUE
```

Over 100 built-in functions

Mathematical/Trig (including hyperbolic trig), Financial, String, Date/Time and Miscellaneous functions are included.

Programmer installable functions.

Functions can be easily registered with the DLL engine. They simply need to follow a prototype and be registered. The parser will ensure that the parameters expected by the function are of the correct type and in the correct order. Once functions are installed in this manner, they become a part of the FB environment and act like any other FB built in function. This allows practically any function imaginable to be easily added to the system.

Variable and Constant support.

Variables and constants may be dynamically added or removed. By default, variables are stored in an expression managed symbol table. For even greater flexibility callbacks may be installed to be fired whenever the engine needs information on a variable or needs to set its value. Variables, therefore, may be implemented in any fashion the programmer desires - from items in a list to fields in a database table.

Efficiency.

Expressions are parsed once, tokenized and stored in an intermediate form for quicker evaluation. There is no need to re-parse when the value of a variable in the expression changes. This is especially beneficial where expressions need to be recalculated in loops.

Delphi Integration

The FormulaBuilder package includes five components which simplify the use and extends the functionality of the calculation engine. They integrate tightly into Delphi's design environment, allowing you to greatly decrease application development time.

IsBoolean Property

See Also

Applies To

TInstanceProperty

Declaration

```
Property IsBoolean : Boolean;
```

Description

Returns true is the instance property is of type boolean. This is necessary since boolean values are treated internally by the RTTI manager as enumerated types.In fact the TTypeKind enumerated type (which is the type of the Kind property) has no entry for boolean. This property determines whether or not the encapsulated property was declared as a boolean in the Object Pascal source code.

See Also

AsBoolean
Kind
Typename

IsDefault Property

Applies To
TInstanceProperty

Declaration
```
Property IsDefault : Boolean;
```

Description
The IsDefault property returns true if the value of the instance property is the default value for that property.

IsNull Property

see also

Applies to

All FormulaBuilder Components

Declaration

```
Property IsNull : boolean;
```

Description

Read Only. Returns true if no infix expression test has been assigned, or if the expression set by the Formula, StrFormula or Lines properties were the empty string "" or NIL.

see also

Clear Method
Formula Property
Lines Property
StrFormula Property

IsReadOnly Property

Applies To

TInstanceProperty

Declaration

```
Property IsReadOnly : Boolean;
```

Description

Returns true if the instance property is readonly, false otherwise.

IsStored Property

Applies To

TInstanceProperty

Declaration

```
Property IsStored : Boolean;
```

Description

Returns true if the property is a stored property.

IsValidDBExpression Function

Unit

FBDBComp

Declaration

```
Function IsValidDBExpression(theDB : TDatabase;expr : pchar):boolean;
```

Description

Determines whether the text expression expr is a valid database expression. Please refer to TDBExpression for additional information.

Kind Property

See Also

Applies To

TInstanceProperty

Declaration

```
Property Kind : TTypeKind;
```

Description

Returns the base type of the property. TTypekind is defined in TYPINFO.INT as follows :

```
type
  TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
             tkString, tkSet, tkClass, tkMethod);
```

See Also

Propinfo
Typedata
Typename

LAST Function

See Also String Functions A-Z Function Reference

Description
Returns the last *count* characters from a string.

Syntax
LAST(*count, source*)

count is the number of characters you wish to extract
source is the string from which to extract the characters

Remarks
If *count* is greater than the length of *source*, the entire string *source* is returned.

See Also

EXTRACT
FIRST

LENGTH Function

See Also String Functions A-Z Function Reference

Description

the length of a string.

Syntax

LENGTH(*St*)

St is any string value or expression

See Also

WORDCOUNT

LN Function

[See Also](#) [Math/Trig Functions](#) [A-Z Function Reference](#)

Description

Returns the natural logarithm base (base e) of x

Syntax

LN(X)

X is the positive real number for which you want the natural logarithm.

Remarks

LN is the inverse of the EXP function, i.e.   LN(EXP(X)) = x

See Also

e
EXP
LOG

LOG Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the logarithm of a number to a given base.

Syntax

LOG(*number<, base>*)

number is the positive real number for which you want the logarithm.
base is the base of the logarithm, it is assumed to be 10 if ommitted, that is LOG(*number*) returns the base 10 Logarithm of *number*.

See Also

EXP

LN

LOWER Function

[See Also](#)   [String Functions](#)   [A-Z Function Reference](#)

Description

Converts a string to all lowercase characters.

Syntax

LOWER(*Source*)

Source is the string you wish to convert to lowercase.

See Also

PROPER

UPPER

LTRIM Function

See Also String Functions A-Z Function Reference

Description

Returns the a string left trimmed of a specified character.

Syntax

LTRIM(*source* <, *trimchar*>)

Remarks

Source is left trimmed of the first character in *trimchar*. If *trimchar* is not specified, the space character is assumed.

See Also

RTRIM

TRIM

License Agreement

Copyright   Ordering Information

FormulaBuilder 1.0

YGB Software, Inc.

READ CAREFULLY BEFORE INSTALLING AND/OR DISTRIBUTING THE SOFTWARE.
You should carefully read the following terms and conditions before using this software. Unless you have a different license agreement signed by YGB Software Inc, or the author Clayton Collie,.your use of this software indicates your acceptance of this license agreement and warranty.

Demoware Version

Use of the Demoware version of this software is limited to a 30 day evaluation period. Use of the Demoware software beyond the evaluation period is a license violation without payment of a license fee. If you decide after the evaluation period that   See the Order Form for additional information.

The remainder of this agreement applies to such persons who have purchased the Software or have otherwise been granted licenses by YGB Software, Inc. or Clayton Collie.

License Agreement

This is a legal agreement between you (either an individual or an entity) and YGB Software, Inc. By Installing and/or distributing the software you are agreeing to be bound by the terms of this agreement.

This License agreement grants you the non-exclusive and non-transferrable right to unlimited use of one copy of the enclosed software program on a single computer. The software is in "use" on a computer when it is loaded into temporary memory (i.e. RAM) or installed into permanent memory (e.g. hard disk, or other storage device) of that computer.

Each license is for a single copy of FORMULABUILDER. A license is required for each developer using FORMULABUILDER. This includes access of FormulaBuilder through a network. All workstations that will access the software through the network for development purposes must have its own FormulaBuilder license, regardless of whether they use FormulaBuilder at different times or concurrently. Please contact YGB Software for information on ***site-licensing*** arrangements.

You may use all the files accompanying this product for development of an application. the included RTF file is provided for your convenience in creating end-user documentation, and may be used and distributed without restriction.

You have a royalty-free right to distribute only the "run-time modules " with the executable files created in any other vendor product (Language or Development Tool) limited as set forth in paragraph **a** through **d**. YGB Software, Inc. grants you a royalty-free distribution if :

(**a**) you distribute the "run time" modules only in conjunction with the executable files that make use of them as a part of your software product;

(**b**) you do not use the YGB Software, Inc. name, logo or trademark to market your software product;

(**c**) The end users do not use the "run time" modules or any other included components for development purposes. and,

(**d**) you agree to indemnify, hold harmless, and defend YGB Software, Inc. and its suppliers from and against any and all claims or lawsuits including attorney's fees, that arise or result from the use or distribution of your software product.

Any violation of the conditions outlined in this agreement constitutes unlawful use of the software. The "run time modules" are those files included in the software package that are required during execution of your software program.

Governing Law

This agreement shall be governed by the laws of the State of New Jersey.

Exclusions or modifications of this agreement can only be made by obtaining written consent from the author, Clayton Collie, or YGB Software, Inc.

Line Property Example

The Lines Property makes it simple and convenient for TMEMO users to set the text form of an an expression :

```
Procedure Tform1.SetLinesFormula;
begin
     Expression1.Lines := Memo1.Lines;
end;
```

Lines Property

see also example

Applies to

All FormulaBuilder Components

Declaration

```
Property Lines : TStrings;
```

Description

Allows read/write access to the original text expression as a TStrings object. This is especially convenient for use with TMemo components;

see also

Formula Property
StrFormula Property

LoadActivated Property

See Also

Applies To

TDSFilter

Declaration

```
Property LoadActivated : boolean;
```

Description

If the Active property is set to true in Design Mode, *LoadActivated* determines whether the filter will be active when the form loads. If TRUE, the attached dataset will be filtered at startup. If false, the Active property must be programmatically set for filtering to occur.

See Also

Active Property

Logical Operators

The following operators work with numeric operands and return an integer. Floating point operands will be truncated to integers before the operation is performed. All except the ***not*** operator are binary operators.
.

Operator	Description
not	performs unary bitwise negation on its operand
and	bitwise and
or	bitwise or
xor	bitwise exclusive or

Mathematical/Trigonometric Functions

FormulaBuilder contains a full complement of math, trigonometric and hyperbolic trigonometric functions. The mathematical functions take numeric values as arguments and return a numeric result. If *list* is specified in the parameter list, it indicates that a list of numeric values is expected. The trigonometric functions expect and return angles in radians as opposed to degrees. To convert between radians and degrees, use the functions RADIANS (degrees to radians) and DEGREES(radians to degrees). In addition to the functions listed below, FormulaBuilder recognizes the predefined constants Pi and e
.

Function	Returns
ABS(*x*)	the absolute value of *x*.
ACOS(x)	the arc cosine of the argument x. x is presumed to be in radians, not degrees.
ACOSH(x)	the hyperbolic arccosine of x.
ACOT(x)	the arccotangent of x.
ACOTH(x)	the hyperbolic arccotangent of x.
ACSC(x)	the hyperbolic arccosecant of x.
ACSCH(x)	the inverse hyperbolic arccosecant of x
ASEC(x)	the inverse secant of x.
ASECH(x)	the inverse hyperbolic secant of x.
ASIN(X)	the inverse sine of x
ASINH(x)	the hyperbolic inverse sine of x
ATAN(x)	the arc tangent of the argument x.
ATAN2(x,y)	the arctangent of an angle defined by the *x*- and *y-coordinates*.
ATANH(x)	the hyperbolic tangent of x
CEILING(x)	x rounded up to the nearest whole number.
COS(x)	the cosine of the argument x.
COSH(x)	the hyperbolic cosine of x.
COT(x)	the arc cotangent of the argument x.
COTH(x)	the hyperbolic cotangent of x.
CSC(x)	the cosecant of the argument x.
CSCH(x)	the hyperbolic cosecant of x.
DEGREES(X)	the value of *x* converted to degrees. *X* is presumed to be in radians.
EXP(x)	the mathematical constant e, raised to the xth power.
FACT(x)	the factorial of *x*. If *x* is a floating point number, it will be truncated to an integer before the calculation occurs.
FLOOR(x)	the argument x rounded down to the nearest whole number.
FRAC(x)	the fraction part of the float expression *x*.
INT(f)	the integer portion of the float expression *f.*
ISEVEN(n)	TRUE is the numeric argument *n* argument is even, false if not
ISODD(x)	TRUE if the argument is odd, false otherwise
LN(x)	the natural logarithm of x.
LOG(x<,n>)	the base *n* logarithm of the number *x.* If n is not specified, the base 10 logarithm is returned.
MAX(*list*)	the largest number in the list of numbers *list*.
MIN(*list*)	the smallest number in the list of numbers *list*.
PRODUCT(*list*)	the product of the list of floating point values
RADIANS(x)	x converted from degrees to radians.
RAND(<n1,n2>)	returns a pseudo-random floating point number.
ROUND()	The Round function rounds a Real-type value to an Integer-type value.
SEC(x)	the secant of x.
SECH(x)	the hyperbolic secant of x.
SIN(x)	the sine of the argument x.
SINH(x)	the hyperbolic sine of x.
SGN(x)	the sign of the number x
SQR(f)	the square of f, i.e. f*f
SQRT(f)	the square root of f

SUM(*list*)	the sum of the list of floating point values, *list*.
TAN(x)	Returns the tangent of the argument x.
TANH(x)	Returns the hyperbolic tangent of x.

MAX Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the maximum value of a list of numbers.

Syntax

MAX(*number1, number2,...*)

number1, number2,... are 1 to MAXPARAMS values for which you want the maximum value.

See Also

- AVG
- MIN
- PRODUCT
- SUM

MAXPARAMS Constant
FormulaBuilder 1.0 supports a maximum of 16 parameters

MAXSTR Function

See Also String Functions A-Z Function Reference

Description

Finds the largest string in a list. That is, it returns the value which would appear first if the list were sorted in descending order.

Syntax

MAXSTR(*string1, string2,* <,...>)

String1, String2 ... are the values from which the largest string is determined. Up to MAXPARAMS parameters are allowed.

See Also

MINSTR

MID Function

[See Also](#) [String Functions](#) [A-Z Function Reference](#)

Description

Returns a specified number of characters from a string, starting at specified position in the string. If the length parameter is not included, the function returns the first start characters of source.

Syntax

MID(*source, start* <*,len* >)

source is the string from which to return characters.
start is the position of the first character to return from text.

If start is 1, the first character in text is returned.
If start is greater than the number of characters in text, an empty string ("") is returned.
If start is less than 1, a null string is returned.
len is the number of characters to return.

Remarks

If start + len exceeds the length of text, the characters from start to the end of source are returned.

See Also

EXTRACT
LENGTH

MIN Function

See Also	Math/Trig Functions	A-Z Function Reference

Description

Returns the smallest number in a list of numbers.

Syntax

MIN(*number1, number2,...*)

number1, number2,... are 1 to MAXPARAMS values for which you want the minimum value.

See Also

- AVG
- MAX
- PRODUCT
- SUM

MINSTR Function

See Also String Functions A-Z Function Reference

Description

Finds the smallest string in a list. That is, it returns the value which would appear first if the list were sorted in ascending order.

Syntax

MINSTR(*string1, string2,* <,...>)

String1, String2 ... are the values from which the smallest string is determined. Up to MAXPARAMS parameters are allowed.

See Also

MAXSTR

MINUTE Function

[See Also](#)   [Date/Time Functions](#)   [A-Z Function Reference](#)

Description

Returns the integer value in the range 0 to 59 corresponding to the minute portion of a date/time serial number.

Syntax

MINUTE(*datetime_serial*)

datetime_serial is the date/time value from which to derive the minute.

See Also

[HOUR]
[SECOND]

MONTH Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns an integer (1 - 12) representing the month component of a date serial number.

Syntax

Month(*date_serial*)

date_serial is the date value.

See Also

[DAY]
[YEAR]

MONTHNAME Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the month name of the month of a date value

Syntax

MONTHNAME(*date1*)

date1 is the date serial number for which you want to find the month name

See Also

MONTH

Miscellaneous Functions

CHOOSE IIF

NOW Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns today's date and time as a date/time serial number value

Syntax

NOW()

Remarks

The date value is stored in the integer portion of the value. The fractional portion represents the fraction of the day.

See Also

TIME
TIMENOW
TODAY

NPER Function

See Also Financial Functions A-Z Function Reference

Description

Calculates the number of periods required for an annuity with regular fixed payments and an optional present value to accumulate a future value at a specific interest rate. This is an extended version of the CTERM and TERM functions.

Syntax

NPER(*Rate,Pmt,Pv*<,*FV,Type*>)

All parameters to this function are numeric values

Argument	**Description**
Rate	the amount of the periodic payment, greater than -1
Pmt	the fixed interest rate per payment period. This can any value except 0.
Pv	the present value of the investment
Fv	the expected future value of the investment
Type	1 for an ordinary annuity, 0 for an annuity due

Remarks

The options parameters, *Type* and *Pv* are both assumed to be 0 (zero) if ommitted.

See Also

CTERM	PV
FV	RATE
PMT	TERM
PPAYMT	

NPV Function

See Also Financial Functions A-Z Function Reference

Description

Calculates the net-present value of a series of a series of cash flows, discounted at a fixed periodic rate

SYNTAX

NPV(*Rate,Value1,Value2,...*)

Rate is the rate of discount over the length of a period.
Value1, Value2,.. represent the numeric values of the cash outflows

NPV assumes that the cash outflows occur at equal time intervals, and that the investment is an ordinary annuity.

See Also

IRR
PV
PVAL

Numeric Constants

FormulaBuilder accepts all legal numeric values within its range of precision. Numbers in scientific notation are also accepted. Numbers without decimals are stored internally as integers (vtINTEGER). Fractional values may begin with a period e.g. .25

OnFindVariable Event

See Also

Applies to

TExpression

Declaration

```
Property OnFindVariable : TFindVariableEvent;
```

Description

The OnFindVariable event for an expression occurs when the expression parser encounters an unknown identifier which may be a variable.or field. In this event the programmer identifies whether or not the identifier is a variable, gives its type, and optional information to speed subsequent lookups. By handling this event, you gain the ability to handle variables external to the core FormulaBuilder engine.

Note - this event must be used in conjunction with the OnGetVariable event (and optionally the OnSetVariable event if assignment statements are to be used). Both must be defined for the engine to properly handle externally defined variables. If these events are defined, the Variables and VariableList properties will not have access to variables handled in this manner.

see also

OnGetVariable event
OnSetVariable event
UseEvents Property

OnGetVariable Event

See Also

Applies To

TExpression

Declaration

```
Property  OnGetVariable : TGetVariableEvent;
```

Description

The OnGetVariable occurs when the expression engine needs the value of a variable used in an expression. This event is used in conjunction with the OnFindVariable event (and optionally the OnSetVariable event) to implement programmer-defined variable handling.

See Also

OnFindVariable event
OnSetVariable event
UseEvents property

OnSetVariable Event

See Also

Applies To

TExpression

Declaration

`**Property** OnSetVariable : TSetVariableEvent;`

Description

The OnSetVariable occurs when an expression containing an variable assignment is made and the value of the variable therefore needs updating. This event is not meaningful apart from the OnFindVariable and OnGetVariable events. These events must all be handled to implement programmer-defined variable processing.

See Also

OnFindVariable event
OnGetVariable event
UseEvents property

Operands
Operands are the data the expression manipulates and combines with operators to derive a value.

Operator Precedence

The result of an expression depends on the order in which operations are performed. Each operator is assigned a *precedence,* and operations are performed in order of precedence. This eliminates possible ambiguities in expressions. An operation can be given higher precedence by surrounding it with parentheses.

Below is the operator precedence list from highest to lowest priority:

Operators	**Precedence**	**Category**
(,)	First	Prioritization
not, - , +	Second	Unary
^,**, *, /, mod, div	Third	Multiplicative
+, - , and , &, or, |, xor	Fourth	Additive
=, <>, <, <=, > , >=,like	Fifth	Relational
:=	Sixth	Assignment

Rules of Precedence

1. An operand between two operators of different precedence is bound to the operator with higher precedence.
2. An operand between two equal operators is bound to the one on its left.
3. Expressions within parentheses are evaluated before being treated as a single operand.

Operators
Operators specify actions to occurs on operands

Order Form
FormulaBuilder 1.0

FormulaBuilder can be registered electronically on Compuserve via the Software Registration (SWREG) system. The registration number is 10343. When on Compuserve, GO SWREG.

Checks must be drawn on a US bank, and both checks and money orders should be made payable to *YGB Software, Inc*.

Send check or money order to
YGB Software, Inc.
161 Pearl St.
Paterson, NJ 07501
USA

Prices guaranteed through March 31, 1995.

FormulaBuilder 1.0	Single Copy _____	copies at $ 65 each	= _______
Georgia residents add	5% sales tax		+ _______
	Total Payment		_______

Please make sure to include the following information:

Name :____

Company :____

Address :____

City :_________________________ **State** :______ **Zip Code** :____________

Country (if not U.S.) :____

E-mail Address :___________________________________

Optional Information

Where did you get this package from?

[ ] CompuServe
[ ] America Online
[ ] another Online Service
Name ______________________
[ ] the Internet
Site ________________________
[ ] Shareware Catalog Vendor: Vendor Name::
[ ] A friend
[ ] None of the above
[ ] other

Comments and Suggestions for improvements:

If you have any question concerning your order please contact us on CompuServe at 103515,1757 (or via the Internet at 103515.1757@compuserve.com) attention YGB.

PADCENTER Function

[See Also](#) [String Functions](#) [A-Z Function Reference](#)

Description

Centers a string in a specified width, filling it out on both sides by a specified character

Syntax

PADCENTER(*source,len,padch*)

Source is the string to be padded.
len is the length of the resulting string
padch is a string. The first character in padch will be the padding character

See Also

PADLEFT

PADRIGHT

PADLEFT Function

[See Also] [String Functions] [A-Z Function Reference]

Description

Returns a string of length len (maximum 255 chars), such that s is leftmost in the string and is filled on the right with pad.

Syntax

PADLEFT(*s , len, pad*)

Source is the string to be padded.
len is the length of the resulting string
padch is a string. The first character in padch will be the padding character

Example

PADLEFT('$'+str(2500.55),15,'*') = ‘$2500.55*******'

See Also

PADCENTER
PADRIGHT

PADRIGHT Function

See Also String Functions A-Z Function Reference

Description

Returns a string flushed right within a specified length. The string is filled on the left with a specified character.

Syntax

PADRIGHT(*source,length,pad*)

source is the string to be padded
length is the desired length of the resulting string
pad is the padding character to use

See Also

PADCENTER
PADLEFT

PAYMT Function

See Also	Financial Functions	A-Z Function Reference

Description

Returns the payment on a loan at the interest rate *rate* for a specified number of payment periods.

Syntax

PAYMT(*Rate,Nper,Pv,*<*,Fv,Type*>)

Argument	Description
Rate	the fixed rate of periodic interest
Nper	the total number of payment periods for the loan. This is a number > 0
Pv	a number representing the principal of the loan (the amount borrowed)
Fv	a number representing the value the investment is expected to reach at a future date. Use a positive value for FV to determine the size payment that would have to be made to accumulate *FV* after *Nper* periods.
Type	a number indicating whether payments are made at the end (0) or the beginning of the payment period (1).

See Also

IPAYMT
PMT
PPAYMT

Pi = 3.14159265358979...

PMT Function

See Also   Financial Functions   A-Z Function Reference

Description

Returns the payment required on a loan at a given interest rate, for a specified number of payment periods.

Syntax

PMT(*Pv,Rate,Nper*<*,Type*>)

Parameter	**Description**
Pv	the principal
Rate	the decimal value representing the interest rate on the loan. This value must be greater than -l
Nper	the number of payment periods.
Type	Payment type. 0 for an ordinary annuity, 1 for an annuity due. Type is 0 by default.

Remarks

Rate and *Nper* must be expressed in the same increments. For instance if payments are made monthly, then *Rate* must be the monthly interest rate for the loan.

See Also

NPER
PAYMT
PPAYMT
PV
RATE

PPAYMT Function

See Also   Financial Functions   A-Z Function Reference

Description

Calculates the portion of a loan that is the principal (as opposed to interest).

Syntax

PPAYMT(*Rate,Per,Nper,Pv,<Fv,Type>*)

Argument	***Description***
Rate	the fixed rate of periodic interest, > -1
Per	a numeric value, the number of periods into the loan for which the principal is desired.
Nper	the total number of payment periods for the loan. This is a number > 0
Pv	a number representing the principal of the loan (the amount borrowed)
Fv	a number representing the value the investment is expected to reach at a future date. Use a positive value for FV to determine the size payment that would have to be made to accumulate *FV* after *Nper* periods.
Type	a number indicating whether payments are made at the end (0) or the beginning of the payment period (1).

See Also

- IPAYMT
- PAYMT
- PMT

PRODUCT Function

See Also Math/Trig Functions A-Z Function Reference

Description

Multiplies all the numbers in a list of numeric values.

Syntax

PRODUCT(*number1, number2,*....)

number1, number2,... are 1 to MAXPARAMS values for which you want the product.

See Also

- AVG
- MAX
- MIN
- SUM

PROPER Function

[See Also](#) [String Functions](#) [A-Z Function Reference](#)

Description

Converts the first letter of every word in s to uppercase. A word is defined as an unbroken string of alphabetic characters. Non-alphabetic characters are unaffected.

Syntax

PROPER(*sourcestring*)

sourcestring is a string value

Example

Proper('JAMES morriS wiLLiams')	returns	'James Morris Williams'

See Also

LOWER
UPPER

PV Function

See Also Financial Functions A-Z Function Reference

Description

Returns the present value of a series of equal payments.

Syntax

PV(*Pmt,Rate,Nper,Type*)

Parameter	Description
Pmt	the amount of the periodic payment
Rate	the interest rate
Nper	he number of periods over which payments are made. This value must be > 0
Type	a number representing the type of payment. 0 for an ordinary annuity, 1 for annuity due.

See Also

FV	PMT
FVAL	PVAL
NPV	RATE

PVAL Function

[See Also] [Financial Functions] [A-Z Function Reference]

Description

Determines the present value of an investment, with a specific future value, based on a series of equal payments, discounted at a periodic interest rate over a number of equal periods.

Syntax

PVAL(Rate,Nper,Pmt<,Fv,Type>)

Parameter	Description
Rate	a value > -1 representing the periodic interest rate
Nper	a positive integer representing the number of payment periods
Pmt	a numeric value representing the amount of the periodic payment
FV	the future value of the investment
Type	0 if payments are made at the end of each period, 1 if they are at the beginning.

See Also

- FV
- FVAL
- NPV
- PMT
- PV
- RATE

ParseAddConstant Method

See Also

Applies To

All FormulaBuilder Components

Declaration

`**Procedure** ParseAddConstant(**const** *cname* : string;*expr* : string);`

Description

Create a constant with the name *name*, setting its value to the result of the expression *expr*. The new constant takes the type of *expr.* If the identifier *name* exists, an EXPR_DUPLICATE_IDENT error is returned.

See Also

AddBooleanConstant
AddConstantPrim
AddDateConstant
AddNumericConstant
AddStringConstant

ParseAddVariable Method

See Also

Applies to

All FormulaBuilder Components

Declaration

Procedure ParseAddVariable(**const** v*name* : string;*expr* : string);

Description

Create a variable with the name *name*, setting its initial value to the result of the expression *expr*. The new variable takes the type of *expr.*

See Also

AddVariable

Parsing Phase

In the Parsing Phase, the string formula is decomposed into its constituent parts - constants, variables, fields, operators and functions. These tokens are assembled into a more efficient but functionally equivalent internal representation of the original expression.

Pascal External Function Example

Suppose we wanted runtime access to a function "myfunc()" . For the sake of our discussion, our function "myfunc()" will include parameters of each type supported by the FormulaBuilder engine. The declaration of our function, in Pascal would be as follows :

```
Function myfunc(l : longint;b : BOOLEAN;d : double;s : string;dt :
TFBDate) :string;
```

We could use this in a FormulaBuilder expression as follows :

```
const
   MYEXPR : string = '"myfunc() returns " + myfunc(12345, true, 10.0245,'+
             '"myfunc string",today() )' + #0;

var
   myHandle   : HEXPR;
   answer     : string;
   buf        : array[0..120] of char;
   ptr        : pchar;
   expr       : pString;

begin
       myHandle := FBInitExpression( 100 ); {}
       FBSetExpression(myHandle,@MYEXPR[1]);
       ptr := @buf;
       FBEvaluate(myHandle,ptr,sizeof(buf)-1);
       answer := strpas(ptr);
end;
```

Implementing The Callback

In order to make myfunc() available, we have to create an exportable callback function with the prototype TCBKExternalFunc. Note that the **export** directive is absolutely necessary. Our implementation of the function follows:

```
Procedure myfunc(     bParamcount     : byte;
                      const params    : TActParamList;
                      var retvalue    : TVALUEREC;
                      var errcode     : integer;
                      exprdata        : longint); export;
var
    result   : string[120];
    datestr  : string[20];
    intval   : longint;
    boolval  : boolean;
    floatval : double;
    strval   : string[80];
    dateval   : TFBDate;

begin
    intval   := params[0].vInteger;
    boolval  := params[1].vBoolean;
    floatval:= params[2].vFloat;
    dateval  := params[4].vDate;
    strval   := params[3].vpString^;
    dateval  := dateTostr(dateval);
    result   := format(' int : %ld  bool : %d float : %f str : %s date : %s ',
       [intval,boolval,floatval,strval,datestr]);

    retvalue.vpString = FBCreateString(result);
```

```
   { ExprData is 100 , the same as in call to FBInitExpression }

    errcode := EXPR_SUCCESS; /* not really necessary, since this is  its value on entry
*/
end;
```

Note that the value of the ExprData parameter is same as the programmer defined value passed as the parameter in the FBInitExpression call.

Registering The Function

Now that our callback function is written, we need simply to register the function with the FormulaBuilder parser. We do so by means of the FBRegisterFunction call.

```
var myFnId : integer;
begin
 myFnId := FBRegisterFunction('myfunc',vtSTRING,'ibfsd',5,myfunc);
end
```

The first parameter tells FormulaBuilder the name of your function, the second its type (see the vtXXX constants). The third parameter describes the parameters expected for the function (integer, boolean, float, string and date respectively). FormulaBuilder guarantees that the elements of the *params* parameter passed to *myfunc* will be exactly of the type and in the order listed. The next parameter instructs the parser to expect a minimum of 5 parameters. This value could have been any value from 0 to the length of the previous parameter. The *nParamcount* parameter of the callback routine, upon entry, contains the number of parameters the user entered. The final parameter, of course, is a pointer to the function which implements "myfunc".

FBRegisterFunction returns EXPR_INVALID_FUNCTION if the call is unsuccessful, otherwise it returns a positive integer > 100 which uniquely identifies your function. You may use the return value from the registration call to unregister the function.

Thats It ! Youve successfully added a function to FormulaBuilder. "myfunc" will be treated like any of FormulaBuilder's other functions. As you can see, practically any function can be added, including wrapper functions for the Windows API.

Passing Data to External Functions

Callbacks and the ExprData (Expression Data) parameter

Every function implementation callback (type TCBKExternalFunc) has a longint argument *ExprData* as its last parameter. *ExprData* provides a means of passing data from the expression instance to the callback. If you were observant, you would notice that FBInitExpression also has an *ExprData* parameter   The value specified here is the same value passed to the callbacks in your code.

Uses Of This Technique

Windows uses this technique (application defined data passing) extensively in the API. In fact every Windows message carries two parameters wParam and lParam which carry additional information related to the message. Certain Windows API functions require callbacks which have an additional parameter for programmer defined data.

Example

To demonstrate the usefulness of this technique, we present a code snippet using the Windows Enumwindows function to get a list of all top level windows and the associated handles.

ExprData and The Delphi Wrapper Components

The ExprData parameter allows us to pass 32 bits of information to our function implementation routines. If you examine the constructor for TExpression in FBCOMP.PAS, you will notice the following statement :

```
fhandle  := FBInitExpression(longint(self));
```

Delphi classes are reference based and allocated on the heap. Therefore this statement actually sets the expression data to a pointer to the just created TExpression or descendant. Using this knowledge, we can now access the instance which called our callback procedure from within the callback itself.

Example 1
Example 2

Priority Property

Applies To
TDSFilter

Declaration
```
Property Priority : integer;
```

Description
The Priority property determines the order of execution of multiple filters attached to the same Datasource. (1=default means first filter to work on, the filter with 2 would be 2nd and so on).

PropInfo Property

Applies To

TInstanceProperty

Declaration

```
Property PropInfo : PPropInfo;
```

Description

Returns a pointer to the RTTI PropInfo record which provides information on the instance property. See TYPINFO.INT for the declaration.

Propname Property

example

Applies To

TInstanceProperty

Declaration

```
Property Propname : string;
```

Description

Returns the property name of the instance property. Setting a value for Propname will change the property which the instance of TInstanceProperty encapsulates. If a published property with the new name does not exist for the current value of the Instance property, and Instance is not NIL, an exception is raised.

Propname Property example

```
Procedure TForm1.PropnameExample;
var
   TestProp : TInstanceProperty;

begin
   TestProp := TInstanceProperty.CreateFromPath(Font,'Caption');
   Try
     Panel1.Caption := TestProp.Typename; {caption will show 'Caption'}
   Finally
     TestProp.Free;
   End;
end;
```

RADIANS Function

See Also Math/Trig Functions A-Z Function Reference

Description

Converts an angle in degrees to its equivalent in radians.

Syntax

RADIANS(*x*)

Remarks

x is any number floating point or integer value. The resulting value is PI/180 * X

See Also

DEGREES
Pi

RAND Function

[Math/Trig Functions](#) [A-Z Function Reference](#)

Description

Returns a pseudo-random number.

Syntax

RAND(*<num1,num2>)*

Num1 and *num2* are both numbers

Remarks

without parameters, RAND() returns a random floating point number between 0 and 1.
RAND(*num1*) returns a random number between 0 and num1.
RAND(*num1,num2*) returns a floating point number between *num1* and *num2.*

RATE Function

See Also Financial Functions A-Z Function Reference

Description

Returns the interest rate per period of an annuity, given a series of constant cash payments made over a regular payment period.

Syntax

RATE(*Fv,Pv,Nper*)

Parameter	**Description**
Pv	the present value of the annuity
FV	the future value - the value you wish for the investment to reach after the last payment
NPer	the total number of payment periods in the annuity.

Remarks

Rate produces a value in the same increment as *NPer*. If *Nper* represents years, an annual rate results; If *NPer* represents months, a monthly interest results, and so on.

See Also

FV	PMT
IPAYMT	PPAYMT
IRATE	PV

REPLACE Function

[See Also](#) [String Functions](#) [A-Z Function Reference](#)

Description

Replace all occurrences of a string with another.

Syntax

REPLACE(*source, search , replacement*)

Source is the original string
Search is the string to replace
Replacement is what *Search* is to be replaced with if found in *Source*

Example

Replace('Please send the IRS your taxes','the IRS','me') = 'Please send me your taxes'

See Also

TRIM

REPLICATE Function

String Functions A-Z Function Reference

Description

Repeats text a given number of times

Syntax

REPLICATE(*Source,Count*)

Remarks

Replicate returns a string containing *count* copies of *Source*, to a maximum length of 255 characters

ROUND Function

See Also Math/Trig Functions A-Z Function Reference

Description

The Round function rounds a float type value to an the nearest integer, or to an optional number of decimal places.

Syntax

ROUND(X[,Places])

X is any number
Places is an optional integer specifying the number of decimal places

Remarks

X is a floating point type value or expression. Round(X) returns an float value that is the value of X rounded to the nearest whole number. If *X* is exactly halfway between two whole numbers, the result is the number with the greatest absolute magnitude.

If the *Places* parameter is specified, X is rounded to *Places* decimal places..

See Also

CEILING
FLOOR
INT

RTRIM Function

See Also String Functions A-Z Function Reference

Description

Removes all instances of a specific character from the right side of a string.

Syntax

RTRIM(*source* <, *trimchar*>)

Source is the original string
the first character of *trimchar* will be removed from the right of source. If this parameter is ommitted, *source* will be right trimmed of all spaces.

Example

RTRIM('200,000','0') returns '2'
Rtrim('2000 ') returns '2000'

See Also

LTRIM
TRIM

RTTIError Object

Unit

FB_RTTI

Declaration

```
Type
 RTTIError = Class(Exception)
 public
     Constructor Create( ecode : integer );
     Property ErrorCode : integer read fErrorCode write fErrorCode;
  end;
```

Description

RTTIError is the error type generated by FormulaBuilder when RTTI related errors are encountered. Upon being raised the ErrorCode property contains one of the following error codes :

Error	Code	Description
RTTI_INVALID_OBJECT	210	An attempt was made to access the property of a **nil** object instance.
RTTI_INVALID_PROPERTY	211	Invalid property. Most likely an invalid property name was passed to a routine, or an unknown property type was encountered.
RTTI_INVALID_PROPVALUE	212	An invalid value was assigned to a property.
RTTI_INVALID_PROPPATH	213	An invalid property name or path was specified.
RTTI_PROP_READONLY	214	An attempt was made to assign a value to a readonly property.

See Also

TExpression

Refresh Method

See Also

Applies To

TDSFilter

Declaration

```
Procedure Refresh;
```

Description

Refreshes the Datasource (and consequently the Dataset) assigned to the TDSFilter instance.

See Also

AutoRefresh

Register Procedure (FBDBComp)

Unit
FBDBComp

Declaration
```
Procedure Register;
```

Description
Registers the Data-Aware components TDBExpression, TDSExpression and TDSFilter with the Delphi Form Designer. See the section entitled Installing FormulaBuilder Components To the Component Palette

Register Procedure

Unit

FBComp

Declaration

```
Procedure Register;
```

Description

Registers the TExpression component with the Delphi Form Designer. See the section entitled Installing FormulaBuilder Components To the Component Palette

Registration

Copyright Order Form License Disclaimer

FormulaBuilder (FB) ™ is not free software. It is released as Demoware. The Demoware version will display a short reminder screen for each task that uses the DLL, but otherwise has all the features and capabilities of the registered version. Use of the Demoware version of FormulaBuilder beyond a 30 day evaluation period requires registration. To register, fill out the attached order form and mail it along with a check or money order for $79 US ($65 before April 1, 1996) to the address below. Checks must be for US funds, drawn on a US bank, and both checks and money orders should be made payable to **YGB Software, Inc**.

Mail Should be sent to
YGB Software, Inc.
161 Pearl St.
Paterson, NJ 07501
USA

FormulaBuilder may also be registered on CompuServe via **SWREG** (number **#10343**). Registration includes 6 months free technical support and automatic eligibility for discounts and special upgrade prices on future products, including the 32-bit edition of FormulaBuilder.

Once your registration is received, you will be sent the current released version of FormulaBuilder, (including source to the non-DLL portions of the package, compiled components, examples and user-level documentation) via E-mail.

Relational Operators
Relational operators are used to compare two or more values. The values being compared are of the same type.

Relational Operators

FormulaBuilder supports the standard relational operators:

Operators	Description
=	Equal
<	Less Than
>	Greater Than
<>	Not Equal To
>=	Greater Than Or Equal To
<=	Less Than Or Equal to
LIKE	Wildcard string match (both operands must be strings)

Removing Variables

Individual variables may be freed using the FreeVariable method. If you wish to remove all variables from a TExpression's variable list, use FreeVariableList.

Note. You should call Reparse after variables have been removed to ensure that the expression remains valid. If a variable is removed that is referenced in an expression, a GPF will occur when you attempt to evaluate that expression.

Reparse Method

Applies to

All FormulaBuilder Components

Declaration

```
Procedure Reparse;
```

Description

Reparses the infix string assigned to the expression instance via the Formula, StrFormula, or Lines properties. This is necessary for subclasses of TExpression (TDSExpression, for example) which derive their variable data from external sources. If the external source changes (if the Dataset property of TDSExpression changes, for example), the expression needs to be reparsed to reset internal variables and to verify if the infix expression is still correct for the new data source.

ReturnType Property

Applies to

All FormulaBuilder Components

Declaration

```
Property ReturnType : byte;
```

Description

Read-only. Returns the vtXXX constant describing the type of the expression, whether or not it has been evaluated. Returns vtTYPEMISMATCH if there is an error in the original expression.

Root Property

Applies to
TRTTIExpression

Declaration
```
Property Root : TObject;
```

Description
Reads and sets the top level object whose properties the expression will have access to. All properties of Root, and recursively the properties of all its named components (if Root is a component type) are available to the expression.

Variable/Property names take the form of "dot notated" identifiers giving the full path to the property. Root serves as the enclosing scope. For example, if Root is a TForm, valid variables are

```
[Caption]
[Font.Name]
```

If on the other hand, Root is the Application instance, and our form is named InvoiceForm (the Name property of the TForm was set at form activation) we would use the following :

```
[InvoiceForm.Caption]
[InvoiceForm.Font.Name]
```

Setting the values of variables in an assignment has the expected effect at runtime. For instance, the following moves the form Form1 down 5 units,

```
Procedure TForm1.MoveItDown;
Var expr : TRTTIExpression;
begin
    expr := TRTTIExpression.Create(NIL);
    TRY
      expr.Root := Self;
      expr.Formula := '[Top] := [Top] + 5';
      expr.AsString;  { Force evaluation }
    FINALLY
      expr.free;
    END;
end;
```

NOTE
In order for the expression to access a particular property, all nodes in the path to the property must be named. The default value of Root is the Forms.Application variable.

SEC Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the secant of the given angle.

Syntax

SEC(*number*)

number is the angle, in radians, for which you want the secant. Use the RADIANS function to convert degrees to radians.

Remarks

The secant function is defined as SEC(X) = 1 / COS(x)

See Also

- CSC
- COS
- SECH

SECH Function

See Also   Math/Trig Functions   A-Z Function Reference

Description

Returns the hyperbolic secant of an angle.

Syntax

SECH(x)

X is the angle in radians.

Remarks

If you wish to convert a value expressed in degrees to radians, use the RADIANS function.

See Also

- ASEC
- EXP
- SEC

SECOND Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the integer value in the range 0 to 59 corresponding to the second portion of a date/time serial number.

Syntax

SECOND(*Serial_Number*)

Serial_Number is the time value from which to derive the second.

See Also

HOUR

MINUTE

SGN Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the sign of a number as an integer value.

Syntax

SGN(*number*)

number is any float or integer value.

Remarks

SGN returns a value as follows
if number < 0, return -1
if number > 0, return 1
otherwise return 0

See Also

ABS

SIN Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the sine of its argument.

Syntax

SIN(*x*)

Remarks

x is the angle in radians for which you want the sine. If the argument is in degrees, convert it to radians with the Radians function.

See Also

ASIN
PI

SINH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the hyperbolic sine of a number.

Syntax

SINH(*number*)

number is any number

See Also

ASINH
COSH
TANH

SLN Function

See Also Financial Functions A-Z Function Reference

Description

Uses the Straight Line depreciation method to calculate the amount of depreciation in one period.

Syntax

SLN(*Cost,Salvage,Life*)

Cost is the original cost of the asset
Salvage is the expected selling price of the asset at the end of its life.
Life is the number of periods (usually in years) the asset is expected to be in use.

See Also

DB
DDB
SYD

SOUNDALIKE Function

See Also

Description

Determines whether two words sound alike, based on the Soundex algorithm.

Syntax

SOUNDALIKE(*str1,str2*)

str1 and *str2* are the strings to be compared. SOUNDALIKE returns TRUE is the strings match, FALSE otherwise.

SOUNDEX Function

See Also

Description

The SOUNDEX() function returns the soundex code for a string.

Syntax

SOUNDEX(*string1*)

string1 is the text string for which you wish to determine the soundex value.

Remarks

See Also

Like Operator

SOUNDALIKE

SQR Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the Square of a number , n - that is (n * n).

Syntax

SQR(*number*)

number is any number

See Also

SQRT

SQRT Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the square root of a positive number.

Syntax

SQRT(*number*)

number is a positive number. If a negative number is passed to this function, FormulaBuilder returns an error with status EXPR_DOMAIN_ERROR.

See Also

SQR

STR Function

See Also String Functions A-Z Function Reference

Description

Converts any value to its string equivalent.

Syntax

STR(*value* <,*places*>)

Remarks

The places parameter is relevant only for numeric arguments and specifies an optional number of decimal places. If ommitted, the function returns the string representation of *number* as it would be normally displayed.

See Also

VAL

SUM Function

See Also String Functions A-Z Function Reference

Description

Returns the sum of all numbers in the list of numeric arguments.

Syntax

SUM(*number1, number2,*....)

number,number2 are the values for which you wish to find the sum, up to a total of MAXPARAMS values

See Also

AVG
PRODUCT

SYD Function

See Also Financial Functions A-Z Function Reference

Description

Uses the Sum-of the-Years-Digits depreciation method to calculate the amount of depreciation in one period.

Syntax

SYD(*Cost,Salvage,Life,Period*)

Argument	Description
Cost	the original cost of the asset
Salvage	the expected selling price of the asset at the end of its life.
Life	the number of periods (usually in years) the asset is expected to be in use. Sometimes called the useful life of the asset.
Period	the period for which you wish to find the depreciation.

Remarks

The following must hold :

Cost >= Salvage >= 0

Life >= Period >= 1

See Also

DB
DDB
SLN

SetVariableCallbacks Method

Applies to

TExpression

Declaration

```
Procedure SetVariableCallbacks(CBKVFind  : TCBKFindVariable;
                               CBKVGetval : TCBKGetVariable;
                               CBKVSetVal : TCBKSetVariable;
                               CBKData    : longint);
```

Description

Register functions to enable external variable processing. Setting callbacks overrides the internal variable handling routines. All variables must be handled externally . An explanation of the parameters follow in the discussion of the chapter on “Extending FormulaBuilder”.

See Also

SOUNDEX

LIKE Operator

Status Property

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Property Status : integer;
```

Description

Returns the last error reported by FormulaBuilder for this object. This value is available regardless of the state of the UseExceptions property, i.e. exceptions are generated in the UseExceptions = True state after the Status property is set. The values returned are the EXPR_XXX constants.

Consult the topic Handling Expression Errors for further details.

See Also

StatusText Method

UseExceptions Property

StatusText Method

See Also

Applies To

All FormulaBuilder Components

Declaration

```
Function StatusText : String; Virtual;
```

Description

Returns the string description of the value of the Status Property. Override this in descendant classes of TExpression to add your own application specific status messages.

See Also
EXPR_XXX Constants
Status Property

StrFormula Property

See Also Example

Applies to

All FormulaBuilder Components

Declaration

```
Property StrFormula : pchar;
```

Description

Allows read/write access to the original expression as a null-terminated string. Values obtained from this property should be disposed of with StrDispose;

StrFormula Property Example

The StrFormula Property allows the use of null terminated strings as the source for expression text. The following code snippets illustrate the use of the StrFormula property :

```
Procedure TFORM1.SetStrFormula;
{ Set Expression Text from Memo }
var tmp : Pchar;
Begin
   tmp := Memo1.GetText;
   TRY
       Expression1.StrFormula := Memo1.GetText;
   FINALLY
       StrDispose(Temp);
   END;
end;
```

Notice very carefully that the TExpression object does not own the memory allocated to the string assigned by the StrFormula property. It retains its own copy of the string data and stores it internally. The caller is responsible for freeing memory as appropriate.The same applies for reading the StrFormula property :

```
var
   tmp : Pchar;

begin
   tmp := Expression1.StrFormula;
   Try
       ResultMemo.Text := 'You entered : '+Strpas(tmp);
   Finally
       StrDispose(tmp);
   End;
end;
```

See Also

Formula Property

Lines Property

String Constant Examples
"This is a string constant"
'this is another string constant'
"this is a string constant with 'mixed quotes'"
'a concatenated string with a " quote ' +" plus a ' quote"

String Constants

String Constants consist of a series of zero or more characters surrounded by delimiters defining the beginning of the constant. Either single or double quotes may be used as delimiters, provided that the same quote type that opens a string constant must be used to close it. String constants can be a maximum of 255 characters long, including quotation marks. See here for an examples...

String Functions

The string functions manipulate character strings.

Function	Description
ASC	Returns the ASCII code for the first character in a string
CHAR	Returns the character corresponding to an ASCII code.
CLEAN	Removes all unprintable characters from a string.
CODE	Returns the ASCII code for the first character of a string.
EXTRACT	Returns a specific delimited word from a string.
FIND	Locates text within a string.
FIRST	Returns a specified number of characters from the beginning of a string.
INSERT	Inserts a substring into a string at a specific position.
LAST	Returns a specified number of characters from the end of a string.
LENGTH	Returns the length of a string.
LOWER	Converts text to lowercase.
LTRIM	Removes all instances of a specific leading character from a string.
MAXSTR	Returns the maximum value in a list of strings.
MINSTR	Returns the minimum value from a list of strings.
MID	Returns a substring of a string.
PADCENTER	Centers a string within a given width.
PADLEFT	Pads a string on the right with spaces to a specified length.
PADRIGHT	Pads a string on the left with spaces to a specified length.
PROPER	Capitalizes the first letter of every word in a string, and lowercases all other characters.
REPLACE	Replaces one substring with another.
REPLICATE	Duplicates a string a specific number of times.
RTRIM	Removes all instances of a specific trailing character from a string.
SOUNDEX	Returns the Soundex code for a string.
SOUNDALIKE	Determines if two strings sound alike, based on their Soundex codes.
STR	Returns the string equivalent of a value.
TRIM	Removes a specific leading and trailing character from a string.
UPPER	Converts a string to uppercase.
VAL	Converts a string to its numeric equivalent.
WORDCOUNT	Returns the number of words in a string.

String Operators

The string operators are used to perform operations on string operands.

Operator	Description
+	Concatenation. Joins two strings together.
-	Deletes the first occurance of the second operand from the first.
LIKE	Performs a wildcard match on the operands. A LIKE B returns TRUE if A matches the wildcard specification B. For Example "AUTOEXEC.BAT" LIKE "*.BAT" returns true.

StringResult Property

See Also Example

Applies to

TExpression, TDBExpression, TDSExpression, TRTTIExpression

Declaration

```
Property StringResult : String;
```

Description

Read only. Evaluates the expression, returning its string result. A type mismatch error EXPR_TYPE_MISMATCH will be generated if the expression type is not vtSTRING. The expression result type can be pre-determined by using the ReturnType property. To get the result as a string, use the AsString property.

StringResult Example

The following example assumes that there is a Customer table, table1 with the fields SALUTATION, LAST_NAME and FIRST_NAME :

```
var
   nameExpression : TDSExpression;
begin
  NameExpression := TDSExpression.Create(Self);
  with nameExpression do
  begin
      Formula := 'UPPER(SALUTATION) + " " + ' +
                 'PROPER(LAST_NAME) + " " + PROPER(FIRST_NAME)';
       namePanel.Caption := NameExpression.StringResult;
  end;
end;
```

See Also

ReturnType
AsString

StringSetToInt Function

Unit

FB_RTTI

Declaration

```
Function StringSetToInt(root : TObject;SetString : String) : Cardinal;
```

Description

This function converts a set string (expressed as a bracketed list of identifiers separated by commas) into the equivalent bitmapped integer value. All the identifiers in SetString must belong to the same enumerated type. Root is the top-level object which is searched recursively to find the enumerated type or set to which the identifiers in SetString belong. There must be a published enumerated or set property in the search path to which they belong, or an exception will be raised.

Example

```
StyleBitmap := StringSetToInt(Form1,'[fsBold,fsItalic]');
GridBitmap  :=
StringSetToInt(Grid1,'[goFixedHorizLine,goHorzLine,goRangeSelect]');
```

StringValues Property

See Also Example

Applies To

All FormulaBuilder Components

Declaration

```
Property StringValues[const name : TVarName]: string
```

Description

Allows read/write access to the value of a variable as a string. This applies to variables added by calls to the AddVariable method. If you assign a value to this property for a variable name that does not exist, a variable will be created and given the value of the evaluated string. For instance

```
Expression.StringValues['NewDate'] := 'Today()';
```

creates a new DateTime variable (vtDATE)with a value of today's date.

Note, however, that for existing variables, expressions are not accepted. Only a valid string representation of the variable's value is accepted.

StringValues Property Example

The StringValues property is useful for setting variables based on input from editboxes.

This code assumes we have an initialized TExpression instance named Expression1, and EditBoxes for each of five variables named Name, BirthDate, Married, Children, and Salary.

```
Procedure TForm1.AddVariables;
begin
  with Expression1 do
  begin
    { Note that the variables were added before the expression }
    { involving them was assigned to the Formula property }
    AddVariable('Name',vtSTRING);
    AddVariable('BirthDate',vtDATE);
    AddVariable('Married',vtBOOLEAN);
    AddVariable('Children',vtInteger);
    AddVariable('Salary',vtFLOAT);
    AddVariable('PIN',vtFLOAT);
  end;
end; { AddVariables }

Procedure TForm1.StringValues_SaveEdits;
begin
  With Expression do
  begin
    StringValues['Name']      := NameEdit.Text;
    StringValues['BirthDate'] := BirthDateEdit.Text;
    StringValues['Married']   := MarriedEdit.Text;
    StringValues['Children']  := ChildrenEdit.Text;
    StringValues['Salary']    := SalaryEdit.Text;
  end;
end;

Procedure TForm1.StringValues_ValuesToForm;
begin
  With Expression do
  begin
    NameEdit.Text := StringValues['Name'];
    BirthDateEdit.Text := StringValues['BirthDate'];
    MarriedEdit.Text   := StringValues['Married'];
    ChildrenEdit.Text  := StringValues['Children'];
    SalaryEdit.Text    := StringValues['Salary'];
  end;
end;
```

See Also

AddVariable Method
VariableList Property
Variables Property

TAN Function

[See Also](#) [Math/Trig Functions](#) [A-Z Function Reference](#)

Description

Returns the tangent of a specified angle

Syntax

TAN(*angle*)

angle is the angle, in radians, for which you want the tangent. To convert an angle in degrees to radians, use the RADIANS function.

See Also

[ATAN]
[ATAN2]
[ATANH]
[Pi]

TANH Function

See Also Math/Trig Functions A-Z Function Reference

Description

Returns the hyperbolic tangent

Syntax

TANH(*number*)

number is the cosine of the angle. The cosine can range from 1 to -1.

Remarks

The formula for the hyperbolic tangent is

TANH(*X*) = SINH(*X*)/COSH(*X*)

See Also

ATANH
COSH
SINH

TActParamList Data Type

Pascal

TActParamList = array[0..MAXFUNCPARAMS-1] of TValueRec;

C/C++

typedef   TValueRec TActParamList[MAXFUNCPARAMS]
typedef   TActParamList, FAR *LPPARAMLIST;

Description

TActParamList is the array type whose values represent the values passed to programmer defined external functions. The number of array elements,as well as their order and type, are guaranteed to be the same as specified in the call to FBRegisterFunction when the external function is registered.

TCBKEnumFunctions Callback Type

Pascal

```
TCBKEnumFunctions = function(name : pchar; vtype : byte;parms :
pchar;minPrms :byte;EnumData : longint):integer;
```

C/C++

```
typedef FBERROR (CALLBACK *TCBKEnumFunctions)(LPSTR name,BYTE vtype,LPSTR
parms,BYTE minPrms,LONG EnumData);
```

Description

Used in conjunction with FBEnumFunctions to enumerate FormulaBuilder run-time functions. A function of this type gets called for each registered FormulaBuilder function, both built-in and programmer-defined.

Parameter	Description
name	the name of the function. Function names are not case sensitive.
vtype	function return type. See the vtXXX constants in the Constants Reference.
parms	a null-terminated string in which each character represents the type of parameter for that position

Type	Character
Integer	'I'
String	'S'
Date	'D'
Float	'F'
Boolean	'B'
Any	'A'

minPrms	the minimum allowable number of parameters, for functions with variable parameter lists
EnumData	The actual parameter specified for the *EnumData* parameter in the FBEnumFunctions call. This field is simply a means by which you can pass data to the callback function. It is strictly programmer defined, and passed untouched by FormulaBuilder. See the FBEnumFunctions Example for a typical use of this parameter.

TCBKExternalFunc Callback Type

Pascal

```
TCBKExternalFunc = procedure(paramcount :Byte;
                        const Params : TActParamlist;
                        var retvalue : TValueRec;
                        var errcode  : integer;
                            ExprData : Longint);
```

C/C++

```
typedef void (CALLBACK *TCBKExternalFunc)(BYTE paramcount, LPPARAMLIST params,
LPVALUEREC retvalue, LPINT errcode,LONG ExprData);
```

Description

This callback is defined to add programmer defined functions to FormulaBuilder. A routine of this type must be supplied to FBRegisterFunction for each function the programmer wants to implement. The function must be declared as _export (in Pascal/Delphi, the procedure header must include the *export* keyword). The *errcode* parameter is set to EXPR_SUCCESS on entry and need only be modified in case of an error in the callback.

Parameter	Description
paramcount	count of parameters passed to the callback
params	a zero based array of *paramcount* TValueRecs containing the parameter values to the function. The expression parser ensures that the type, count and order of these parameters match those specified when the function is registered.
retvalue	return value type of type TValueRec. The appropriate variant of this record is set to the function return value. The parser sets the tag field before the callback is called.
errcode	set to EXPR_SUCCESS on entry, this parameter is for programmer use to flag errors which occur in the function callback. Other values will cause the expression evaluator to trigger an error when the callback returns. This value is returned as the result of the currently executing evaluation function (FBEvaluate, FBEvaluatePrim, etc).
exprdata	a user-defined field to allow the programmer to pass data to the callback. The actual parameter when the callback is executed is the *exprData* value passed as the argument to the FBInitExpression function.

NOTE It is important to note, in regards to the ***params*** argument, that FormulaBuilder performs automatic type conversions between compatible types to ensure that the type specified for a function argument matches its registered type. For example the built-in function CHAR takes an integer parameter and returns the corresponding ASCII character. The parser will happily accept 190.78 as an argument, but will truncate it to 190 before passing it to the function.

TCBKExternalFunc Function Implementation Callback

```
Type
TCBKExternalFunc = procedure(paramcount :Byte;
                        const Params : TActParamlist;
                        var retvalue : TValueRec;
                        var errcode  : integer;
                            ExprData : Longint);
```

TCBKFindVariable Callback Type

Pascal

```
TCBKFindVariable = function(varname : pchar;var vtype : byte;var vardata : longint;CKBData : longint):integer;
```

C/C++

```
typedef FBERROR (CALLBACK *TCBKFindVariable)(LPCSTR varname,LPBYTE vtype,LPLONG vardata,LONG CBKData);
```

Description

This callback is called when the FormulaBuilder expression parser encounters an unknown identifier *varname* in the parsing phase , to determine if it represents a valid variable or field identifier. If so, the function should return EXPR_SUCCESS as its value after setting the appropriate variable/field type in *vtype* (see the vtXXX constants). The *vtype* parameter should be set to vtNONE if the *varname* does not represent a valid programmer defined variable.

Remarks

This is the means by which the parser gathers information about the variable/field that is used in the construction of the expression. The *vardata* parameter is programmer-definable parameter you may use as a convenient means of passing data between the two evaluation phases. The parser does nothing with this field. In most cases, this can serve as a unique identifier for the variable/field, a typecasted pointer to the storage location of the variable, or an array index if the variable were stored in a list. This allows us to either eliminate the need for, or limit the overhead of a lookup in the evaluation phase when the value of the variable is required. The data collected by this event is the same data that is passed to the *CBKGetVariable*,*and* *CBKSetVariable* callback events.

TCBKGetVariable Callback Type

See Also

Pascal

```
TCBKGetVariable = Function(varname : pchar;var value : TValueRec;vardata :
longint;CBKData : longint):integer;
```

C/C++

```
typedef FBERROR (CALLBACK *TCBKGetVariable)(LPCSTR varname,LPVALUEREC
value,LONG vardata,LONG CBKData);
```

Description

This event is fired during the evaluation phase when a value is needed for a variable or a field encountered in the expression. The *CBKData* parameter is the user-defined value passed in the call to FBSetVariableCallbacks   The *varname* parameter identifies the name of the variable/field,and the *vardata* parameter is the programmer defined value initialized in the TCBKFindVariable Callback. The tag field of *value*, vtype, is set to the vtXXX constant denoting the type requested by the callback.The programmer simply assigns the variable's value to the appropriate field of the *value* record. See the notes concerning TValueRec for more details.

By default, variable are handled internally by the DLL, but this event gives you the flexibility of deciding how variables are implemented and how they are stored.

See Also

TCBKFindVariable
TCBKSetVariable

TCBKSetVariable Callback Type

Pascal

```
TCBKSetVariable = function(varname : pchar;var value :TValueRec;vardata :
longint;CBKData : Longint):integer;
```

C/C++

```
typedef FBERROR (CALLBACK *TCBKGetVariable)(LPCSTR varname,LPVALUEREC
value,LONG vardata,LONG CBKData);
```

Description

This event is fired during the evaluation phase when the value of a variable or field on the left hand of an assignment changes. For instance, in the expression :

```
Force := Mass * Velocity
[parts->OnHand] := [Parts->onHand] - 100
```

the value of Force needs updating after the right hand side of the expression is calculated. The *varname* parameter is the name of the variable. *vardata* is the same as the programmer defined in the TCBKFindVariable callback for the variable name.*value* is the new value to be assigned to the variable. The parser ensures that the type of the variable matches the type the programmer specified in the TCBKFindVariable Event. The programmer has the responsibility of updating the variable/field with *value*.

TDBExpression Component

See Also Properties Methods

Unit

FBDBComp

Description

This subclass enhances the TExpression class by adding access to fields of all BDE (Borland Database Engine) datasets open on its Database property. These fields can then be treated in the same manner as variables in expressions.

The syntax for database fields is '[' tablename'->'fieldname']'. For example :

```
TotalCostExpr := ' [Items->Price]*[Items->Quantity]*(1 + Vendor->TaxRate])';
```

TDBExpression Example

Suppose we have an order entry system with a Customer and an Order Table. The following example calculates how much is owed by overdue customers. It could, of course be written using SQL, but this example shows the flexibility and ease of use of the TDBExpression class.

```
Procedure TForm1.CalcOverdue;
var exprFilter,exprCost : TDBExpression;
    fTotal : extended;
begin
    CustomerTable.Open;
    OrderTable.Open;
    exprCost := TDSExpression.Create(NIL);
    With ExprCost do begin
        DataBase  := OrderTable.Database;
        Formula := '[Orders->TOTAL] * (1 + [CUSTOMER->TAX_RATE])';
    end;
    exprFilter := TDBExpression.Create(NIL);
    exprFilter.Database := OrderTable.Database;
    exprFilter.Formula  := '([Customer->BALANCE] > 0) AND '
                           '((TODAY() - [Customer->LASTPMTDATE]) > 30)';
    fTotal := 0;
    OrderTable.First;
    while not OrderTable.EOF do
    begin
       if exprFilter.AsBoolean do
          ftotal := fTotal + exprCost.AsFloat;
       orderTable.next;
    end;
    OrderTable.Close;
    CustomerTable.Close;
    resultPanel.Caption := FloatToStr(fTotal);
end;
```

Methods

AddBooleanConstant	Clear	GetVarPtr
AddConstantPrim	Create	ParseAddConstant
AddDateConstant	Destroy	ParseAddVariable
AddNumericConstant	EvaluatePrim	Reparse
AddStringConstant	FreeVariable	StatusText
AddVariable	FreeVariableList	

TDBExpression Properties

TDBExpression adds the Property Database

All Other properties are derived from TExpression

See Also

- TDSExpression
- TDSFilter
- TExpression

TDSExpression Component

See Also Properties Methods

Unit

FBDBComp

Description

The TDSExpression subclass enhances the TExpression class by adding access to fields of the BDE (Borland Database Engine) dataset assigned to its Dataset property. These fields can then be treated in the same manner as variables in expressions. When the expression is recalculated, the value of the variables are read directly from the fields of the dataset.

Example

If a TTable instance *LineItems* is open on a line item table containing the fields QUANTITY and PRICE, the following would be valid :

```
ExtensionExpr := TDSExpression.Create(NIL);
TRY
  ExtensionExpr.Dataset := LineItems;
  ExtensionExpr.Formula := 'PRICE * QUANTITY‘;
  total := 0;
  while not LineItems.eof do
  begin
    total := total + extensionExpr.AsFloat;
    lineItems.Next;
  end;
FINALLY
  ExtensionExpr.Free;
END;
```

Properties

AsString	Dataset	ReturnType
AsBoolean	Formula	Status
AsDate	FunctionCount	StrFormula
AsFloat	Handle	StringResult
AsInteger	IsNull	UseExceptions

See Also

TDSFilter
TDBExpression
TExpression
TRTTIExpression

TDSFilter Component

See Also Properties Methods Tasks

Unit

FBDBComp

Description

The TDSFilter component implements a high level interface to BDE-level dataset filtering. Using this component, you are able to filter a datasource based on any valid FormulaBuilder boolean expression. This component has a major enhancement over the filtering/sorting methods of VCL for LOCAL databases.

Filters can be based on any valid FormulaBuilder expression returning a boolean result.

they can be applied to any existing local Table or Query. All other filters and ranges of the dataset are respected

they are completely independent of the current index, and return a dataset that is fully editable. This avoids the ORDERS BY restriction of Borland's Local SQL, whereby "live" result sets cannot be returned for certain variations of ORDERS BY clauses.

Credits

TDSFilter is based directly on and incorporates DBFILTUZ.PAS Version 1.06 COPYRIGHT (C) by UZ [INFOPLAN], CIS ID Address : Uli Zindler 100271,313

Methods
TDSFilter introduces the Refresh method
All other methods are inherited from TDSExpression

NOTE
Unlike for PDX-tables (**IDPDX01.DLL**), the BDE triggers an exception when a dBASE-table's callback-filter is aborted (**IDDBASE01.DLL**, firing EOF-condition). As a workaround to the EOF-trap exception-box, exceptions are temporarily caught when a filter is to be aborted. UNFORTUNATELY,   when "stop on exceptions" in OPTIONS |ENVIRONMENT set to true, and the debugger is running you'll still be thrown into that "exception... ,program stopped" mode, but you can resume, by pressing F9 or the RUN-button. This exception will be visible ONLY occur at Design Time, and will not appear to users.

Properties

= Key property

Active	FilterHandle	Lines
AutoRefresh	Formula	Priority
Datasource	LoadActivated	UseExceptions

See Also

TExpression

TDSExpression

TDSFilter Tasks

Using FormulaBuilder With Delphi

Using the TDSFilter Component

Let's suppose we have a DataSource-object (DataSource1) on a form, and that it's linked to either a TTable or TQuery-object as its DataSet (let's call this one DataSet1)

place a TDSFilter control on your form
connect it to DataSource1
set the Formula or Lines property to a desired filter expression. The fields of Dataset1 are available to the expression. Field names are handled as variables.
set the AutoRefresh and Priority properties to desired values
set the Active Property to true

Records are included or excluded based on the expression entered in the Formula or Lines property. As such, the expression must evaluate to a boolean, otherwise a EXPR_TYPE_MISMATCH error will be generated. Filtering is implemented through a private method of the TDSFilter. ***NOTE***: number of calls varies thru the BDE caching and buffering mechanisms of DELPHI

LIMITATIONS

This component works only on Datasources connected to LOCAL datasets. It may not work on correctly on remote SQL-driven databases, etc.

Do NOT try to access detail-controls (in case the active filtered DataSet is the master-side in an 1:n relation), they may be invalid at the time. Instead, use appropriate checks on the DETAIL-side of two 1:n linked DataSets, where, the master-controls will be reflect proper data when the Dataset's current record is being filtered.

NOTE

TERM Function

See Also   Financial Functions   A-Z Function Reference

Description

Returns the number of payment periods required to accumulate an investment (future value) given a regular series of payments and a fixed interest rate.

Syntax

TERM(*Pmt,Rate,Fv,Type*)

Parameter	**Description**
Pmt	a numeric value representing the amount of the fixed periodic payment.
Rate	a numeric value representing a fixed, periodic interest rate accrued by the investment
FV	a numeric value representing the amount to which the investment will grow (the future value)
Type	a numeric value denoting the payment type - 0 for an ordinary annuity (the default) or 1 for an annuity due.

See Also

CTERM

NPER

See Also

- TDBExpression
- TDSExpression
- TDSFilter
- TRTTIExpression
- EFBError

TExpression Component

See Also Properties Methods Events Tasks

Unit

FBComp

Description

TExpression is the basic component wrapper around the FormulaBuilder DLL. It provides convenient OOP access to the functionality of FB. It also serves as the ancestor class for TDSExpression and TDBExpression, which are Data-Aware.

Variable Support

Variable support may be handled in 2 ways

- using the default processing of the FormulaBuilder engine. This is the standard behavior.
- by delegation. Simply assign methods to the OnFindVariable, OnGetVariable and the OnSetVariable event properties.

You can programmatically control how variables will be handled by setting the UseEvents property. Setting it to FALSE (default) means that FormulaBuilder will handle all variables internally. Setting the property to TRUE means that the variable handling events will be invoked to allow you to manage variables in your own code in addition to the internally handled variables.

Field Support

Fields were added to allow for variables whose names do not fit the usual convention for variable names. There is therefore no default handling of fields, and using them requires that the event properties. OnFindVariable, OnGetVariable and optionally OnSetVariable be assigned to methods which identify, retrieve and set the values of fields identified in text expressions. In the OnFindVariable event, a field may be distinguished by the fact that the varname parameter still contains the field delimiters "[" and "]".

In all other respects, however, fields are handled identically to variables.

Events

OnFindVariable
OnGetVariable
OnSetVariable

Methods

AddBooleanConstant	Clear	GetVarPtr
AddConstant	Create	ParseAddConstant
AddDateConstant	Destroy	ParseAddVariable
AddNumericConstant	EvaluatePrim	Reparse
AddStringConstant	FreeVariable	StatusText
AddVariable	FreeVariableList	

Properties

AsBoolean	FunctionCount	StringResult
AsDate	Handle	StringValues
AsFloat	IsNull	UseEvents
AsInteger	Lines	UseExceptions
AsString	Returntype	VariableCount
Constants	Status	VariableList
Formula	StrFormula	Variables

TFBDate Type

Pascal

```
Type
      {$IFDEF VER80} (* compiler is Delphi *)
      TFBDate = TDateTime;
      {$ELSE}
      TFBDate = Double;
      {$ENDIF}
```

C/C++

```
typedef double TFBDate,TFBDATE, FAR *LPFBDATE;
```

Description

TFBDate is the internal type used to store Date/Time values. It is actually a double which stores the date value as the integer portion and the time value as the fractional portion.

TFBString Type

Pascal

```
Type TFBString = PString;
```

C/C++

```
typedef char FBString[256];
typedef FBString *TFBSTRING,FAR *LPFBSTRING;
```

Description

TFBString is the type that FormulaBuilder uses internally to store string values. It is a pointer to a Pascal byte string, i.e. a string in which the first byte represents the length of the string, followed by the string data. Since the length of the string is byte-sized, FormulaBuilder strings and string results are limited to 255 characters. C/C++ and VB users should use the Utility Functions to deal with variables of this type.

See Also

TCBKFindVariable Callback

TFindVariableEvent Type

See Also

Unit

FBCOMP

Declaration

```
TFindVariableEvent = Procedure(const varname  : string;
                              var   vtype    : byte;
                              var   errcode  : integer
                              var   vardata  : longint) of object;
```

Description

This event is called when the parser encounters an unknown identifier *varname* in the parsing phase , to determine if it represents a valid variable or field. If so, the function should set the *errcode* parameter value to EXPR_SUCCESS after setting the appropriate variable/field type in *vtype* (see the vtXXX constants). The *vtype* parameter should be set to vtNONE if the *varname* does not represent a valid programmer defined variable or field. *errcode* can be set to any of the EXPR_XXX constants to give more details on errors which occur in the event. It is set to EXPR_SUCCESS on entry, and if changed will cause an error to be generated when the event returns.

Note

Field names include the enclosing brackets when passed into the varname parameter.

Remarks

The *vardata* parameter may be set to any programmer defined value which may help in subsequent lookups of the variable. This is the same value that will be passed to the TGetVariableEvent event type when the value of the variable *varname* is needed in the evaluation phase. In most cases, this can serve as a unique identifier for the variable/field, a typecasted pointer to the storage location of the variable, or an array index if the variable/field were stored in a list.

See Also

OnGetVariable event

TGetVariableEvent Type

See Also

Unit

FBCOMP

Declaration

```
TGetVariableEvent   = Procedure(const varname   : string;
                                var   value     : TValueRec;
                                var   errcode   : integer;
                                      vardata   : longint) of object;
```

Description

The TGetVariableEvent type points to a method that gets called when the expression engine needs the value of the variable or field *varname* that was previously identified in the OnFindVariable event. *vardata* is the programmer defined value set in the OnFindVariable event. The value of the variable should be copied to the appropriate field of the *value* parameter. See the definition of TValueRec for additional information. The tag field of value is set on entry. *errcode* should be set to any appropriate EXPR_XXX constant to indicate any errors occurring during the event. It is only necessary to set *errcode* in the event of an error, since it is set to EXPR_SUCCESS on entry to the event.

Note

Field names include the enclosing brackets when passed into the varname parameter.

TIME Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the date/time serial number from individual Hour, Minutes and Seconds values.

Syntax

TIME(*hour,minutes,secs*)

hour a number between 0 and 23, representing the hour, where 0 is 12:00am and 23 is 11:00pm
minutes a number between 0 and 59
secs a number between 0 and 59

Remarks

If the specified values are not within range, an EXPR_CONVERT_ERROR error is raised. The resulting value represents the fraction of the day represented by the time *hour:minutes:secs*

See Also

TIMENOW
TIMETOSTR
TIMEVALUE

TIMENOW Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the time-serial number of the current time according to the system clock.

Syntax

TIMENOW()

Remarks

TIMENOW returns the current time as a fractional portion of the day. This is stored in the fractional part of the returned value. the integer portion of the returned value will be 0 (zero).

See Also

NOW
TIMETOSTR

TIMETOSTR Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns the string representation of a time serial number

Syntax

TIMETOSTR(*time_serial*)

time_serial is a time serial value

See Also

STR
TIMEVALUE

TIMEVALUE Function

[See Also](#) [Date/Time Functions](#) [A-Z Function Reference](#)

Description
Returns the time serial number of a text string.

Syntax
TIMEVALUE(*Timestr*)

Timestr is a text string in any valid time format.

Remarks
The time serial number represents a decimal fraction representing the times from 0:00:00 (12:00:00 A.M.) to 23:50:59 (11:59:59 P.M.)

For example
0.0 represents 12 midnight
0.5 represents midday (12:00 P.M.)

See Also

TIMETOSTR

See Also

AsString

See Also

AsString

See Also

[AsString]
[Kind]
[Typename]

See Also

[AsBoolean]
[AsString]
[Kind]
[Typename]

See Also

Kind

See Also

AsBoolean
AsChar
AsFloat
AsInteger
AsMethod
AsObject
Kind

TInstanceProperty Object

Properties Methods

Unit

FB_RTTI

Description

The TInstanceProperty class encapsulates a single published property of an object instance. This provides a higher level interface to the TYPINFO unit provided by Borland to access Runtime Type Information (RTTI).

Reading and setting the AsString property of this class changes the value of the associated instance property and produces the expected Delphi runtime behaviour. The PropName property returns the name of the instance property, and Instance reads and sets the object Instance to which the property belongs. TypeName returns a string with the Object Pascal type the property was defined as. Use IsReadOnly to determine if the property is readonly, and isStored to determined if it is automatically stored using Delphi's streaming mechanism. The IsDefault property returns true if the value of the instance property is the default value for that property.

Methods

Create
CreateFull
CreateFromPath
CreateFromSearch

Properties

AsBoolean	AsString	Kind
AsChar	Instance	Propname
AsFloat	IsBoolean	TypeName
AsInteger	IsDefault	TypeData
AsMethod	IsReadOnly	PropInfo
AsObject	IsStored	

TODAY Function

See Also Date/Time Functions A-Z Function Reference

Description

Returns today's date as a date serial number value

Syntax

Today()

See Also

NOW
TIMENOW

TRIM Function

See Also String Functions A-Z Function Reference

Description

Trims a string of a specified character on both the left and right

Syntax

TRIM(*source* <, *trimchar*>)

source is the original string

source is left and right trimmed of the first character in *trimchar*. If *trimchar* is not specified, space is assumed.

See Also

LTRIM
RTRIM

TRTTIExpression Component

See Also   Properties   Methods   Tasks

Unit

FBRTComp

Description

The TRTTIExpression component allows one access to the published properties (recursively) of a given Delphi object using Delphi's Runtime Type Information (RTTI). Properties are handled as "dot-notated" FormulaBuilder field identifiers, and reading and setting them has the expected Delphi runtime behaviour.

Properties

Formula
Lines
Root
UseExceptions

See Also

[RTTIError]
[TDBExpression]
[TDSExpression]
[TDSFilter]
[TExpression]

TRTTIExpression Tasks

Using FormulaBuilder With Borland Delphi

Using the TRTTIExpression Component

Since this expression type retrieves runtime type information, its properties must also be set at runtime. The Root property must be set before text is assigned to the Formula or Lines properties.

place a TRTTIExpression control in your form

in the FormCreate event, set the Root property to the object or component whose properties you wish to access. To have access to all published properties in an application, set the Root property to *Application.*

set the Formula or Lines property to a desired expression. The properties of Root are available to the expression. Property names are handled as variables.

Property Paths

Setting the Root property establishes scope for expression property variables. When using property names in expressions, the full path to the property from Root must be used.

For instance, if Root is set to an instance of a TForm, valid property paths would be

```
[Caption]
[Font.Name]
```

Note also that you also have (recursive) access to the properties of named components contained in the Components array of components. For instance, given the same form which contains a TDataSource named CustomerSource, we could use the following property :

```
[CustomerSource.Dataset.Tablename]
```

CustomerSource is visible to the expression since the Delphi Form Designer automatically adds the *CustomerSource* (and all owned components) to the Form.Components array.

If the Root property were set to *Application*, and our form were named *CustomerForm*, we would write the properties as follows :

```
[CustomerForm.Caption]
[CustomerForm.Font.Name]
[CustomerForm.CustomerSource.Dataset.Tablename]
```

Type Equivalences

Delphi Property Types are mapped to FormulaBuilder types as follows :

Delphi	**FormulaBuilder**
tkInteger	vtINTEGER
tkEnumeration	all except boolean are mapped to vtINTEGER. Boolean is mapped to vtBOOLEAN.
tkSet	vtINTEGER
tkFloat	vtFLOAT
tkString	vtSTRING
tkClass	vtPOINTER
tkMethod	vtPOINTER

Check the interface file TYPINFO.INT in the \DELPHI\DOC\ directory for the Delphi Property types.

Enumerated Type and Set Identifiers

Enumerated and Set type identifiers may be used for a published property of that type. For instance, the following is valid for the above CustomerForm example

```
[CustomerForm.Font.Style] := fsBold or fsItalic
```

LIMITATIONS

This component works for all types except Classes and Methods. Assignment and equality testing of Class and Method types will be supported in a later release.

TSetVariableEvent Type

Unit

FBCOMP

Declaration

```
TSetVariableEvent  = Procedure(const varname  : string;
                               const value    : TValueRec;
                               var   errcode  : integer;
                                     vardata  : longint) of object;
```

Description

This event type points to a method which is called when a field or variable in an expression needs to be updated with the results of the calculation. This occurs only for assignment expressions, i.e. expressions of the form

variable := *expr* or
field := *expr*

When the right side of the statement is calculated, the OnSetVariable event is called to allow the programmer to update the value of the field or variable. *varname* is the name of the field/variable. vardata is the programmer defined data set in the OnFindVariable event. *value* is the TValueRec structure describing the new value for the variable/field. It is the programmer's responsibility to update the value of the variable/field with the appropriate field of the *value* record.

See Also

OnSetVariable

TSetVariableEvent Type

See Also

Unit

FBCOMP

Declaration

```
TSetVariableEvent  = Procedure(const varname    : string;
                               const value      : TValueRec;
                               var   errcode    : integer;
                                     vardata    : longint) of object;
```

Description

This event is fired during the evaluation phase when the value of a variable or field on the left hand of an assignment changes. For instance, in the expression :

```
'Force := Mass * Velocity'
'[parts->OnHand] := [Parts->onHand] - 100'
```

the values of Force and parts->onHand need updating after the right hand side of the expression is calculated. The *varname* parameter is the name of the variable. *vardata* is the same application specific data the programmer assigned in the OnFindVariable event for the variable. *value* is the new value to be assigned to the variable/field. The parser ensures that the type of the variable (the *vtype* field of the value structure) matches the type the programmer specified in the OnFindVariable Event. The programmer has the responsibility of updating the variable/field with *value*. The programmer has the responsibility of updating the variable with *value*. You may set the *errcode* parameter to any one of the EXPR_XXX constants to indicate an error condition. It is not necessary to set it otherwise - the value of *errcode* on entry to this method is EXPR_SUCCESS.

TValueRec Type

Pascal

```
PValueRec = ^TValueRec;
TValueRec = record
      flags : byte;
      case vtype  : byte of
         vtINTEGER : (vInteger  : Longint);
         vtSTRING  : (vpString  : PString);
         vtPOINTER : (vPointer  : Pointer);
         vtBOOLEAN : (vBoolean  : Boolean);
         vtCHAR    : (vChar     : Char);
         vtFLOAT   : (vFloat    : Float);
         vtDATE    : (vDate     : TDateTime);
       end;
```

C/C++

```
typedef struct tagTValueRec {
     BYTE      flags;
     BYTE      vtype;
     union {
        long               vInteger;
        boolean            vBoolean;
        unsigned char      vChar;
        float              vfloat;   // double
        TFBString          vpString;
        TFBDate            vDate;
        LPVOID             vPointer;
    }
 } TVALUEREC,TValueRec, *PVALUEREC, FAR *LPVALUEREC;;
```

Remarks

The vtype Field describes the expression return type, with the corresponding variant holding the appropriate value.

Integer values are stored as long (32 bit) integers
Float values are stored as 8 byte Doubles
String values are stored as a pointer to a byte string (Pascal type string). The first byte contains the length of the string, with the string data immediately following. C and Basic programmers should use the utility functions to deal with the vpString field of the TValueRec
Date/Time values (or serial numbers) are stored as a double
Numbers to the right of the decimal point represent the fractional portion of the day. For example 0.5 represents noon (12:00 PM), 0.75 represents 6 PM, and 0 represents midnight.
Numbers to the left of the decimal point represent the number of days since 1/1/001, minus 1. i.e 1.0 represents the date 1/1/0001.
Date and time values may be combined to uniquely identify a time and date.

TVariable Type

Declaration

```
Type
      TVariable = Record
            Name : string[30];
            Value : TValueRec;
       end;
```

Description

This is the variable type used in the Delphi wrapper class TExpression and its descendants. It is returned from the array properties Variables and VariableList. If a variable is assigned to a TVariable , FBFreeValue should be called on the *value* field after the variable is no longer needed.

Technical Support

YGB Software,Inc can help you with any problem you encounter installing or using FormulaBuilder. Support for FormulaBuilder will be primarily through e-mail, though other means may be made available.

E-Mail

You can contact us through CompuServe at 103515,1757, or via the Internet at 103515.1757@compuserve.com

Postal-Mail

Please address your correspondence to:

Technical Support Department
YGB Software, Inc.
161 Pearl St.
Paterson, NJ 07501
USA

The Evaluation Process

Expression Evaluation is broken into two discrete steps

Parsing Phase - the parser scans the input stream for a valid infix expression, translating it into tokenized RPN (Reverse Polish Notation) form . The intermediate step may seem superfluous, but has its inherent advantages. First, RPN is a more compact means of representing expressions than infix. Given equivalent expressions in infix and RPN, the RPN representation can be evaluated as least as quickly, and in most cases quicker, than the corresponding infix representation. Second, the infix expression is completely tokenized in this phase - all functions, variables, fields, constants, operators and other tokens are pre-identified. FormulaBuilder scans and parses the expression only once, regardless of the number of times the expression is evaluated. In the majority of cases, expressions will be calculated in loops, with only the value of variables being modified. This approach eliminates the overhead of the parsing process on subsequent evaluations of the expression

Evaluation Phase - the intermediate representation is translated into a single value representing the result of the input expression.

The Usual Approach

The Old Way

We can certainly proceed as we have done in the previous examples and call AddVariable (or FBAddVariable) for each of the variables in the equation. Then each time we want to evaluate the formula for a new scenario, we would have to set each variable's value before we recalculate the expression. This approach raises a few concerns :

If we want to allow the user to modify the equation. we will have to hard-code ALL the variables which could possibly be used in the equation. This may be plausible for database tables, but restrictive for even a medium sized spreadsheet. Added to this, we would have to do this for EACH expression instance.

The old approach can be tedious and inefficient for a large set of variables. In our example, data will be derived from a spreadsheet as well as a database, so we will have to access the appropriate data source to obtain the values , THEN we would call the appropriate routine to set the variable to the value. This can be very inefficient, since it is quite likely that not all of the variables will be used in the equation.

Suppose we have multiple sources of data. For instance, say we wish to have two sets equations - one based on last year's financial data, and one based on the current year's performance. This is not a problem if the structure of the database and spreadsheet are the same between fiscal periods. If this assumption cannot be made, however, **generalizing the retrieval of variable values may be difficult.**

The vtANY Type : Example 1

FormulaBuilder has a built-in function IIF which returns the value of one of two expressions based on a boolean condition. Its syntax is

IIF(*condition*, *true_expr*, *false_expr*)

If *condition* evaluates to true, the value of *true_expr* is returned, otherwise *false_expr* is returned. Both *true_expr* and *false_expr* may be of any of the types supported by FormulaBuilder. Since this is the case, the IIF function must also be able to return any type.

An implementation of the IIF function is as follows :

```
procedure IIFProc(paramcount          : Byte;
                  const Params        : TActParamlist;
                  var   Retvalue      : TValueRec;
                  var   errcode       : integer;
                        ExprData      : longint); export;
var condition : boolean;
begin
  condition  := params[0].vBoolean;
  if condition then
     retvalue := FBCopyValue(params[1]) {retvalue must be a copy, since }
  else                                 { params array is destroyed after return }
     retvalue := FBCopyValue(params[2]);
end;
```

IIFProc is registered as follows :

```
IIFFnid  :=  FBRegisterFunction('IIF',vtANY,'baa',3,IIFProc);
```

The vtANY Type : Example 2

It is not immediately obvious from the IIFProc example that the arguments can be of different types. To demonstrate this, we will implement a function PARMINFO which returns a string describing the parameters passed to it

```
Procedure ParamInfoProc( paramcount     : byte;
                         const params   :  TActParamList;
                         var   retvalue : TValueRec;
                         var   errcode  : integer;
                               exprdata : longint); export;
var i       : integer;
    tmpstr : string[255];
    anycount,intcount,stringcount,
    floatcount, boolcount, datecount : integer;

begin
  intcount    := 0;
  floatcount  := 0;
  boolcount   := 0;
  datecount   := 0;
  anycount    := 0;
  stringcount := 0;
  if paramcount = 0 then
  begin
    tmpstr  := ' No parameters '+#0;
    retvalue.vpString := FBCreateString(@Tmpstr[1]);
    exit;
  end;
  for i := 0 to pred(paramcount) do
  with params[i] do
  begin
    case vtype of
       vtInteger : inc(intCount);
       vtstring  : inc(stringcount);
       vtFloat   : inc(floatcount);
       vtboolean : inc(boolCount);
       vtdate    : inc(datecount);
       vtany     : inc(AnyCount); { should NEVER get here }
    end;
  end;
  tmpstr := ' %d Params : %d Ints, %d Strings,%d Booleans, %d Floats, '
            +'%d Dates , %d variants ';
  tmpstr := format(tmpstr,[paramcount,intcount,stringcount,
                           boolcount,floatcount,datecount,AnyCount]) + #0;
  retvalue.vpString := FBCreateString(@tmpstr[1]);
end;
```

The registration statement should look as follows :

```
ParamInfoFnId :=
FBRegisterFunction('PARMINFO',vtSTRING,'aaaaaaaaaaaaaaaa',1,ParamInfoProc);
```

The vtANY Type : Example 3

The built in SUM function takes only numeric values, and will raise an error if other types are entered as parameters. It is sometimes useful, however, to permit other types of arguments, whether or not the function uses them. Spreadsheets for example have functions such as @SUM and @AVG which work on ranges which may contain non-numeric data. In such cases those cells with non-numeric data are ignored.

We will implement a sum function which works along the lines of a spreadsheet summation function, in other words, we will simply ignore non-numeric values rather than raise an error.

```
Procedure AtSumProc( paramcount     : byte;
                     const params   :  TActParamList;
                     var   retvalue : TValueRec;
                     var   errcode  : integer;
                           exprdata : longint); export;
var i   : integer;
    sum : extended;

begin
  sum := 0;
  for i := 0 to pred(paramcount) do
  with params[i] do
  begin
    case vtype of
       vtInteger : sum := sum + vInteger;
       vtFloat   : sum := sum + vFloat;
    end;
   retvalue.vFloat := sum;
end;
```

We register ATSUM as follows :

```
var AtSumFnId : integer;
begin
   AtSumFnId := FBRegisterFunction('AtSum',vtFLOAT,
                           'aaaaaaaaaaaaaaaa',1,AtSumProc);
end;
```

Type And Constant Reference

This section provides an alphabetical reference to the types, constants and variables not otherwise covered in this document.

EXPR_XXX Constants
HEXPR
TActParamList
TFBDate
TFBString
TValueRec Type
TVariable
vtXXX Constants

Limits

Floating point results	5E-45 to 1.7E308, 15 significant digits
Integer results	-2147483648 to 2147483647
String Results and Constants	255 characters
Max number of DLL clients	limited by memory
Max number of expressions	16,000
Max number of variables per expression	limited by memory
Max size of expression	32k
Number of Variables Per Expression	16,000
Number of Constants	limited by memory

Typename Property

see also example

Applies To

TInstanceProperty

Declaration

```
Property Typename : String;
```

Description

Returns the Object Pascal type identifier describing the type of the instance property.

Typedata Property

see also

Applies To

TInstanceProperty

Declaration

```
Property Typedata : PTypedata
```

Description

Returns a PTypeData pointer to the property's type data. See TYPINFO.INT for more information.

see also

Typename property

Typename Property example

```
Procedure TForm1.DisplayFontStyleType
var
   StyleProp : TInstanceProperty;

begin
   StyleProp := TInstanceProperty.CreateFromPath(Font,'Style');
   Try
     Panel1.Caption := StyleProp.Typename; {caption will show 'TFontStyles'}
Finally
     StyleProp.Free;
   End;
end;
```

See Also

Typedata property

UPPER Function

See Also String Functions A-Z Function Reference

Description

Returns a string with all uppercase characters.

Syntax

UPPER(*source*)

Source is the string to be uppercased

See Also

LOWER

PROPER

UseEvents Property

Applies To

TExpression

Declaration

```
Property UseEvents : boolean;
```

Description

Read/write. This property determines whether the events OnFindVariable, OnGetVariable and OnSetVariable are used by the expression instance. If UseEvents is FALSE (the default), the expression instance only references the internally managed variables added with AddVariable and ParseAddVariable. You simply need to call AddVariable for each variable you wish to use, and those variable names may then be used in the expression text.

Setting UseEvents to TRUE allows you to handle variables in your own code in addition to the default behavior. In this state, the expression instance will call the OnFindVariable, OnGetVariable and optionally the OnSetVariable event handlers that you define in your code. If you set UseEvents to TRUE, and have not assigned methods to at least the OnFindVariable and OnGetVariable, you will get an error message.

UseExceptions Property

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Property UseExceptions : boolean;
```

Description

This property determines whether FormulaBuilder errors will be generate exceptions in the expression classes. If not, the EXPR_XXX constant representing the error will returned in the Status property of the component.

See Also

EFBError
Status
StatusText

Using FormulaBuilder

FormulaBuilder 1.0 is implemented as a standard Windows 16bit Dynamic Link Library. If you are familiar with accessing DLLs from your programming environment, the process of using FormulaBuilder in your projects should be straightforward. There are header files provided for each of the environments supported by FB.In addition, we have provided several classes to greatly simplify the use of the toolkit for Delphi programmers. This is covered in the Delphi Component Reference:

Click on one of the following for information on using FormulaBuilder in your development environment,

Delphi

Visual Basic

Advanced users and users of other programming tools should go to the DLL Reference section of this document. Regardless of the environment, using FormulaBuilder follows these basic steps.

Using FormulaBuilder With Visual Basic

Using FormulaBuilder in any project follows a basic flow. The following provides further details specific to the Visual Basic environment.

Adding FormulaBuilder To a Visual Basic Project
Initializing An Expression
Freeing the Expression
Assigning The Text To Be Evaluated
Retrieving The Expression Text
Clearing An Expression
Determining an Expression's Return Type
Getting Results
Using Variables
Handling Expression Errors

Using FormulaBuilder with Delphi

This section outlines the basic tasks associated with incorporating FormulaBuilder into a Delphi project. If you have not yet done so, please consult the Delphi Component Reference for an overview of the available components. Using FormulaBuilder in any project follows a basic flow. The following provides further details specific to the Delphi environment.

Tasks

Using The Data-Aware Expression Types

The FormulaBuilder Delphi Classes TDSExpression and TDBExpression allow you to use expressions based on BDE datasets. The TDSFilter expression implements a flexible FB expression based filter for datasources.

Initializing the Data-Aware classes follows the same procedure as the non-data-aware TExpression. Their use is also similar with the exception of where variable data is derived from.

Setting The Data Source

Using The Variable/Field Callback Functions

External Variable/Field Handling

The Variable/Field Callback functions allow us to generalize the handling of variables. this has numerous advantages :

Variables become totally dynamic and virtualized. We no longer have to hard-code variable names. When FormulaBuilder detects an identifier it suspects may be a variable it calls your code to identify it. If our Stock Market program had the capability of linking to various databases and tables with different field structures, we would not necessarily need to know the structure of tables beforehand. Our callback routine would compare the unknown identifier with the list of field names to see if there is a match. This is precisely the approach taken in the data-aware classes TDSExpression and TDBExpression.

Since we are also dealing with spreadsheets in our example we can examine the unknown identifier to see is its format fits the cell naming convention for our spreadsheet. If so, we can use whatever means we need (DDE for example) to access the value of the variable.

Variable values can be derived directly from their source. In the evaluation phase of the expression evaluation process, FormulaBuilder calls a routine of the type TCBKGetVariable to retrieve the current value of a variable.

Using our example, the callback routines retrieve the current value directly from either the database or the spreadsheet. Since no restriction is placed on what occurs in the callback, we have no limitations on how we access the variable data. We could, if we wish, retrieve the data over a DDE link, or perform some transformations on the value before passing it back to FormulaBuilder.

There is no need for a list of FBSetVariable or (SetVariable) calls. The callback routine will only be called for the variables that have been used in the formula.

Multiple expression instances can derive their data from the same data sources.

Using Variables

Default Variable Processing

FormulaBuilder makes it easy for you to define and use variables in expressions. By default, all variable handling is taken care of automatically : a list of variables is maintained internally for each TExpression TDBExpression and TRTTIExpression instance.

Adding Variables
How Many Are There ?
Getting And Setting Variable Values
Removing Variables

Advanced Variable Processing

For more sophisticated needs, you may elect to process variables within your own code. By delegating variable handling events to custom methods and setting the UseEvents property to TRUE, you gain the flexibility of handling variables in the manner most appropriate for your application. See Advanced Variable/Field Handling for greater detail.

Utility Routines

The following routines are provided to handle FormulaBuilder types and perform conversions between those types and types native to Windows programming environments.

FBCopyValue
FBCreateString
FBDateToPasString
FBFreeValue
FBlpzToDate
FBPasStringToDate
FBStringToDate
FBStrncpy

VAL Function

[See Also](#) [String Functions](#) [A-Z Function Reference](#)

Description

Converts a string into its numeric (float) equivalent

Syntax

VAL*(numstring)*

numstring is a string with a valid string representation of an integer or floating point number.

See Also

STR

VB : Adding Variables Example

Example

```
Sub Form_Load ()
  handle& = FBInitExpression&(0)
  status% = FBAddVariable%(handle&,"Age",vtINTEGER)
  status% = FBSetIntVariable(handle&,"Age",31)
  status% = FBAddVariable%(handle&,"Married",vtBOOLEAN)
  status% = FBAddVariable%(handle&,"Dependents",vtINTEGER)
  status% = FBAddVariable%(handle&,"Salary",vtFLOAT)
  status% = FBParseAddVariable(handle&,"Name"," Proper('John' + 'Smith') ")
  status% = FBParseAddVariable(handle&,"BirthDate"," TODAY() - (365 * Age) ")
End Sub
```

VB : Assigning The Text To Be Evaluated

Expression Text is assigned to the evaluation engine by a call to FBSetExpression

Example

```
' This example assumes that handle& has been defined and set by
' a call to FBInitExpression
'
'      handle& = FBInitExpression(0)

Declare Function SetFormula(ByVal formula$) as Integer
Dim result As String * 256

  status% = FBSetExpression%(handle&, formula$)
  If status% <> EXPR_SUCCESS Then
     beep
     FBGetErrorString Status%,result$,254
     MsgBox "Error In Expression : "+result$
     SetFormula = FALSE
  Else
     SetFormula = TRUE
  End If
End function
```

VB : Clearing An Expression

The FBClearExpression function sets the text and tokenized versions of an expression to NULL, and returns an expression instance to the state it would be in after a call to the FBInitExpression call.

Note It is not necessary to clear an expression before changing the expression text. For instance, there is no need for a FBClearExpression in the following code :

```
Sub UpdateCalc()
Dim result as String * 256

  Status% = FBSetExpression%(handle&,"Sin(X^2) * Abs(X * COS(Y))")
  Status% = FBEvaluate%(handle&,result$,255)
  Panel1.Caption = result$
  Status% = FBSetExpression%(handle&,"IIF(WeekDay(Today()= 2,TRUE, FALSE )")
  Status% = FBEvaluate%(handle&,result$,255)
  Panel2.Caption := result$
End Sub
```

VB : Determining an Expression's Return Type

As soon as the text of an expression is set using the FBSetExpression% call the engine "compiles" the text expression into a tokenized form. A benefit of this process is that the result type of the expression may then be determined without evaluating the expression.

If the expression is valid, the FBGetReturnType% function returns one of the vtXXX constants describing the result type the expression will return upon evaluation. If the expression is invalid, the function will return vtTYPEMISMATCH.

For example, if we had called FBSetExpression% with each of the following strings, the FBGetReturnType% would reflect the type of result that would be expected :

Text Expression	Return Type
'Sin(X) / LN(X^2)'	vtFLOAT
'TODAY() - 365'	vtDATE
'WEEKDAY(TODAY()) > 5'	vtBOOLEAN

Note There are certain built-in functions (CHOOSE and IIF for example) which may return any of the standard FormulaBuilder types. If these functions are used in a text expression, FormulaBuilder will try to determine the return type based on the other operators and operands used in the expression. In certain cases it is impossible for the engine to figure out the return type beforehand. In these instances a vtANY is returned.

VB : Getting Expression Results

Once an expression has been properly initialized with FBInitExpression, and given a valid text formula to evaluate (by calling FBSetExpression), we can get the results from the expression with a call to the FBEvaluate function.

To get the result of an expression in native format, use the FBGetXXXResult functions, where XXX represents the variable type. To determine the result type of an expression, use the FBGetReturnType% function.

VB : Handling Expression Errors

Most FormulaBuilder functions return one of the EXPR_XXX constants describing the status of the operation. To get a text description of the error code, use the FBGetErrorString procedure.

Example

```
Sub DisplayErrorMessage(ByVal code%)
Dim errorText As String * 121

   if code% = EXPR_SUCCESS then
   ' no error, just exit
     exit sub
   End if
   FBGetErrorString code%, errorText$, 120
   Beep
   MsgBox errorText$, MB_ICONHAND
End Sub
```

VB : Retrieving The Expression Text

Expression text previously set with FBSetExpression can be retrieved with the FBGetExpression call.

Example

```
Dim formula$ as String * 256
Dim Status as Integer

Status% = FBGetExpression%(handle&,formula$,255)
```

VB : Using Variables

Adding Variables   Example

Variables are added using the FBAddVariable function. You may also add a variable based on the value of a text expression using the FBParseAddVariable function.

Disposing Variables

FBFreeVariable may be used to dispose of a single named variable. To dispose of all variables related to an expression, us the FBFreeVariableList function.

Counting Variables

You can determine the number of variables added by calls to FBAddVariable and FBParseAddVariable by using the FBGetVariableCount function.

Getting Variable Values

Depending on the type of variable, its value may be retrieved using one of the FBGetXXXVariable functions, where xxx represents the variable type. To retrieve the string value of a variable, regardless of its type, use the FBGetVarAsString function.

In addition, the FBPeekVarVB functions allows access to variables by numeric index. Use this along with the FBGetVariableCount function to iterate through the list of variables associated with an expression.

Setting Variable Values   Example

Depending on the type of variable, its value may be set using one of the FBSetXXXVariable functions, where xxx represents the variable type. To set the value of a variable from a string, use the FBSetVarFromString function.

VB : Variable Setting Example

```
' Plots the graph of Formula$ for numpts% evenly spaced values of x between
' XMin# and XMax#.
' Assumptions :
'      Formula$ contains an expression in terms of a variable X,
'          e.g. "Sin( radians(x) ) + Cos( ln(X) )
'      Xmin# < XMax#, numpts% >= 2
'      There is a VB Graph control on the form named FormulaGraph
'

' SUB to display FormulaBuilder Errors
Sub DisplayErrorMessage (ByVal code%)
    Dim errtxt As String * 256

    FBGetErrorString code%, errtxt$, 255

    If code% <> EXPR_SUCCESS Then
       errtxt$ = "ERROR > " + errtxt$
       Beep
    End If

    MsgBox errtxt$, MB_ICONHAND, "Expression Error"

End Sub



Sub PlotGraph (ByVal Formula$, ByVal XMin#, ByVal XMax#, ByVal numpts%)
    Dim Ydata#, xDelta#, Xdata#, YMin#, yMax#
    Dim tmp#, cnt%
    Dim errcode%, returntype%
    Dim graphExprH&


    graphExprH& = FBInitExpression(0)
    If graphExprH& < 0 Then
       MsgBox "Cannot initialize graph expression", MB_ICONHAND
       Exit Sub
    End If

   ' Add a float variable to the engine. FormulaBuilder manages it
   ' it for you
   '
    errcode% = FBAddVariable%(graphExprH&, "X", vtFLOAT)
    If errcode% <> EXPR_SUCCESS Then
       DisplayErrorMessage (errcode%)
       Exit Sub
    End If

   ' Set the expression text for the graph
    errcode% = FBSetExpression%(graphExprH&, Formula$)
    If errcode% <> EXPR_SUCCESS Then
       DisplayErrorMessage (errcode%)
       Exit Sub
    End If

    ' At this point, were only interested in floating point types
    returntype% = FBGetReturnType(graphExprH&)
    If (returntype% <> vtFLOAT) Then
        MsgBox "Floating point expression expected"
        Exit Sub
```

```
    End If

    ' Determine x increment value based on xmin and xMax
    xDelta# = (XMax# - XMin#) / numpts%

    ' expand left and right endpoints by x increment
    XMax# = XMax# + xDelta#
    tmp# = XMin#
    XMin# = XMin# - xDelta#

    cnt% = 0

   ' Setup preliminary graph parameters
    FormulaGraph.AutoInc = 0
    FormulaGraph.DrawMode = 0 ' delay painting until all points calculated
    FormulaGraph.GraphCaption = "Calculating ....."
    FormulaGraph.NumPoints = numpts%
    FormulaGraph.ThisSet = 1

  ' Set the beginning value of X
    errorcode% = FBSetFloatVariable%(graphExprH&, "X", tmp)

    While (Xdata# <= XMax#) And (cnt% < numpts%)

        ' Evaluate the formula for the current value of X
        errorcode% = FBGetFloatResult%(graphExprH&, Ydata#)

        ' Pass values on to graph
        FormulaGraph.ThisPoint = cnt% + 1
        FormulaGraph.GraphData = Ydata#
        FormulaGraph.XPosData = Xdata#

     ' increment our loop count
        cnt% = cnt% + 1

      ' calculate new value of x
        Xdata# = Xdata# + xDelta#

      ' Set x variable to new value
        errorcode% = FBSetFloatVariable%(graphExprH&, "X", Xdata#)
    Wend
    FormulaGraph.GraphTitle = "Graph Of : " + Formula$
    FormulaGraph.DrawMode = 2 'repaint graph
End Sub  ' PlotGraph
```

ValueAsString Function

Unit

FBComp

Declaration

```
Function ValueAsString(const FValue : TValueRec):String;
```

Description

Converts a TValueRec record to its equivalent string representation.

Variable Handling Functions

The routines in this section deals with variable processing internal to FormulaBuilder. If you wish to handle variables in your own code, you should refer to the section on Callbacks in the chapter “Extending FormulaBuilder”.

Adding Variables

Disposing Variables

Counting Variables

Getting Variable Values

Setting Variable Values

Variable Parameter List Example 1

The LOG function, as it is implemented in FormulaBuilder, has one mandatory and one optional parameter :

LOG(*number* <,*base*>)

The log function returns the logarithm of a *number* to a specified *base*. In the current implementation, if *base* is ommitted, it is assumed to be 10.

We will show an implementation of the LOG function :

```
Procedure LogProc( paramcount     : byte;
                   const params   : TActParamList;
                   var   retvalue : TValueRec;
                   var   errcode  : integer;
                         Exprdata : longint);export;
var number : extended;
    base   : integer;
begin
  number := params[0].vFloat;
  if paramcount = 1 then
     base := 10
   else
     base := params[1].vInteger;
  try
    retvalue.vFloat := ln(number) / ln(base);
  except
    on EInvalidOp do errcode := EXPR_DOMAIN_ERROR;
    on EZeroDivision do errcode := EXPR_ZERO_DIVISION;
  end;
end;
```

Since we don't know beforehand how many parameters the function will receive, we must examine the paramcount parameter. Notice that true to our previous definition of LOG, base is set to 10 if only the first parameter (number) is entered.

We notify the parser that we have a variable parameter list when we register the callback with FBRegisterFunction

```
LogFnId := FBRegisterFunction('LOG',vtFLOAT,'fi',1,LogProc);
```

As you know from previous examples, the fourth parameter of FBRegisterFunction is an integer value specifying the minimum number of parameters expected for the programmer-defined function. Since it is less than the maximum number of parameters expected by the function (length('fi') = 2), the parser will allow no less than one and no more than 2 parameters.

Variable Parameter List Example 2

The SUMSQ function returns the sum of the squares of its arguments. We can have as few as 1 and as many as 16 parameters of type float.

```
Procedure SumSqProc(   paramcount  : byte;
                       const params   : TActParamList;
                       var   retvalue : TValueRec;
                       var   errcode  : integer;
                             Exprdata : longint); export;
var i   : integer;
    sum : extended;
    sqr : Extended;

begin
  sum := 0;
  for i := 0 to pred(paramcount) do
  begin
    number := params[i].vFloat;
    sum    := sum + (number * number);
  end;
  retvalue.vFloat := sum;
end;
```

We register SUMSQ as follows :

```
SumSqId := FBRegisterFunction('SUMSQ',vtFLOAT,
                          'ffffffffffffffff',1,SumSqProc);
```

Although this example uses only float parameters, we could just as easily mix parameter types. In other words, using variable parameters does not mean that all parameters must be of the same type.

VariableCount Property

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Property VariableCount : integer;
```

Description

Read-only. Returns the number of variables successfully added with the AddVariable method or the Variables property.

See Also

AddVariable
ParseAddVariable
VariableList
Variables

VariableList Example

```
Function TForm1.GetVariableListing : TStringList;
var i   : integer;
    tmp : String[10];
    curvar : TVariable;

begin
  result := TStringList.Create;
  For i := 0 to Expression.VariableCount - 1 do
  begin
    curVar := Expression.VariableList[i];
    tmp    := curVar.Name + #9 + ValueAsString(curVar.Value);
    FBFreeValue(curVar.Value); { in case of string variables }
    result.Add(tmp);
  end;
end;
```

VariableList Property

See Also Example

Applies to

All FormulaBuilder Components

Declaration

```
Property VariableList[i : integer]:TVariable;
```

Description

This array property provides read/write access to the expression's variable list by a numerical index i, where i is between 0 and VariableCount-1. If a variable is assigned to a this property , FBFreeValue should be called on the value field after the variable is no longer needed.

See Also

AddVariable Method
StringValues Property
Variables Property

Variables

Variables are symbols which represent unknown values. Variable names in FormulaBuilder begin with an alphabetic character, followed by any combination of alphanumeric characters. Variable identifiers are not case sensitive. They may be handled automatically by FormulaBuilder, or in programmer code by using the FBSetVariableCallbacks function call (or the Onxxx events in the Delphi Components).

Variables Property

See Also

Applies to

All FormulaBuilder Components

Declaration

```
Property Variables[const vname  : TvarName]:TValueRec;
```

Description

This array property provides read/write access to the variable values by a name. If you attempt to assign a value to a variable that does not exist, a variable with name *vname* is created and given the value of the rvalue of the assignment. TValueRec is described in the appendix. If a variable is assigned to a this property , its value should be disposed of with FBFreeValue after the variable is no longer needed

Example

```
Interest := MortgageExpr.Variables['Interest'];
Interest.vFloat := Interest.vFloat + inflation;
MortgageExpr.Variables['Interest'] := Interest;
```

Variables Property Example

Assume we have a form of type TForm1, an initialized TExpression instance Expression1, and a record (Person) with the following structure :

```
Type
      TPerson = record
            Name     : String[45];
            Salary   : Double;
            Married  : boolean;
            Children : byte;
            BirthDate : TDateTime;
      end;

Procedure TForm1.AddVariables;
begin
  with Expression1 do
  begin
    { Note that the variables were added before the expression }
    { involving them was assigned to the Formula property }
    AddVariable('Name',vtSTRING);
    AddVariable('BirthDate',vtDATE);
    AddVariable('Married',vtBOOLEAN);
    AddVariable('Children',vtInteger);
    AddVariable('Salary',vtFLOAT);
    AddVariable('PIN',vtFLOAT);
    Formula := 'PIN := Length(Name) + DAY(BirthDate) -
                (Sqrt(Age) * Salary) * IIF(Married,Kids,0)';
  end;
end; { AddVariables }

Procedure TForm1.SetVariables;
var Salary, name, age, DOB, married, kids : TValueRec;
begin
 { First we set the type of variable }
   Salary.vtype    := vtFloat;
   Salary.vFloat   :=  Person.Salary;

   married.vtype     := vtBOOLEAN;
   married.vBoolean  := Person.Married;

   Kids.vtype         := vtINTEGER;
   Kids.vInteger      := Person.Children;

   name.vtype        := vtSTRING;
   name.vpString     := FBCreateString(Person.Name);

   dob.vtype         := vtDATE;
   dob.vDate         := Person.BirthDate;

  { now set our variable values }
  { note that even though we assume that the variables have already been
   added by a call to AddVariable, the Variables property will automatically
   create variables that do not exist. }
   With Expression do
   begin
     Variables['Name']      := Name;
     Variables['BirthDate'] := DOB;
     variables['Married']   := Married;
     variables['Children']  := Kids;
     Variables['Salary']    := Salary;
   end;
```

```
  end;


Procedure TForm1.GetVariables;
  var temp : TValueRec;
  begin
     With Expression1 do
     begin
       temp := Variables['Name'];
       Person.Name := temp.vpString^;
       FBFreeValue(temp);
       temp := Variables['BirthDate'];
       Person.BirthDate := temp.vDate;
       Person.Married  := variables['Married'].vBoolean;
       Person.Children := variables['Children'].vInteger;
       Person.Salary   := Variables['Salary'.vFloat;
     end;
  end;
```

See Also

AddVariable Method
ParseAddVariable Method
VariableList Property

WEEKDAY Function

See Also		Date/Time Functions		A-Z Function Reference

Description

the numeric day of the week associated with date1 as an integer between 1 and 7. Sunday is the first day of the week and Saturday is the seventh.

Syntax

WEEKDAY(*DateSerial*)

DateSerial is the date/serial number from which you wish to derive the weekday number.

See Also

DAY
DAYNAME

WORDCOUNT Function

See Also String Functions A-Z Function Reference

Description

Calculates the number of words in a string, based on a set of delimiters.

Syntax

WORDCOUNT(*source, delims*)

source is the string for which words are to be counted
delims is the string of delimiters. A word is considered as an unbroken sequence of alphabetic characters, delimited by *delims.*

See Also

- EXTRACT
- LENGTH

Windows API Callback Example

This code snippet demonstrates the use of the extra longint parameter in Windows callback functions to pass application specific data. Notice the use of the *lParam* parameter :

```
Uses WinProcs, Classes

{Use the Windows Enumwindows API function to get a list of all top-level windows
and their handles  }

{ Callback function called  by the EnumWindows function }
 Function GetParentWindowListProc(Handle : Hwnd;
                                    lparam : longint):BOOL; export;
          { implicit typecast. Convert lparam back to a TStringlist}
 var List      : TStringList absolute lparam;
     len       : word;
     title     : array[0..120] of char;
     ptitle    : PChar;

 begin
   result   := TRUE; { enumerate for all parent windows }
   ptitle   := @Title;
   title[0] := 0;
   len  := GetWindowText(handle,ptitle,sizeof(title)-1);
   if len > 0 then
   begin
     if not assigned(list) then
        List := TStringList.Create;
   { Store the window title and handle. Note that }
   { since Tstringlist expects a Tobject reference }
   { as the second parameter, we must typecast the }
   { handle to quiet the compiler. To use the handle,}
   { we must cast in the opposite direction.
   { Eg. handle := Longint(List.Objects[1]);   }
     list.AddObject( strpas(ptitle), TObject(Longint(Handle)) );
   end;
  end;

  { Main function. Calls EnumWindows to obtain window list }
  Function GetParentWindowList : TStringList;
  begin
    Result := TStringList.Create;
    { pass the reference to the list as the lparam }
    EnumWindows(@GetParentWindowListProc,longint(result));
  end;
```

YEAR Function

[See Also]   [Date/Time Functions]   [A-Z Function Reference]

Description

Returns the year of the date argument.

Syntax

YEAR(*dateSerial*)

dateSerial is the date serial number of   the date from which you wish to extract the year.

See Also

DAY
MONTH
TODAY

annuity due
An annuity due is an annuity in which payments are made at the beginning of each period.

The base of the natural Logarithm
$e = 2.718281828459045235360287471 3$

evaluation phase
the phase of the evaluation process where the internal tokenized representation of the string expression is evaluated to return a singular value.

ordinary annuity
An ordinary annuity is an annuity in which payments are made at the end of each period.

previous examples

- Example 1
- Example 2
- Example 3

vtANY = 11

vtBOOLEAN = 1

vtDATE = 9

vtFLOAT = 3

vtINTEGER = 0

vtNONE = 13

vtPOINTER = 5

vtSTRING = 4

vtTYPEMISMATCH = 14

vtXXX Constants

The vtXXX constants are unsigned character (byte) sized values representing the types handled by **FormulaBuilder**.

Constant	Value	Description
vtINTEGER	0	32 bit integer value (longint)
vtBOOLEAN	1	byte sized boolean
vtCHAR	2	8-bit unsigned character
vtFLOAT	3	double
vtSTRING	4	pointer to a Pascal string (Pstring). The first represents the length of the string, with the string data immediately following.
vtPOINTER	5	a 32bit pointer value
vtDATE	9	DateTime type. A double in which the integer portion represents the number of days since 01/01/0001 and the fractional part represents the fractional part of the day.
vtANY	11	Any type. This type is used to allow the parser to handle functions whose parameters or return value may be of any of the other types supported by the engine. Type checking on ANY parameters or return values is deferred until the evaluation phase. Note that TValueRec does not have a field corresponding to this type.
vtTYPEMISMATCH	14	Type mismatch or invalid result. Most like the operands in an operation are incompatible with an operator.
vtNONE	13	a special value the programmer uses in the TCBKFindVariable callback (or the OnFindVariable event for Delphi programmers) to indicate that the token passed is not a variable