SSL Functions

Version 1.4

Written by: STARLIMS

Date: July 31, 2007

Revision and History

Date	Revision Content	Section Revised	Rev	By Whom
May 08, 2006	Updated from v9 to v10	All	Draft 1	Mirela Popovici
May 31, 2006	Revision for beta rlease	All	Draft 2	Mirela Popovici
2006-08-04	General Formatting	All	Draft 1.1	Carol Zeik
2006-08-04	Revised formatting from Carol	All	1.2	Mirela Popovici
2007-07-31	Marked LimsSecs as obsolete.	All	1.3	Mirela Popovici
2011-04-07	Updated :INCLUDE statement	Include	1.4	Mirela Popovici

Table of Contents

THIS PAGE HAS BEEN LEFT INTENTIONALLY BLANK

Array Functions

Aadd()

Description

Used to add a new element to the end of an array. This function increases the actual length of the target array by one. The newly created array element is assigned the value specified by the New Value parameter.

Aadd is used to dynamically increase the size of an array. It is useful for building dynamic lists or queues. The target array and the element's new value are defined in the parameters.

Parameters

Aadd(Target Array, New Value)

Input Types: (array, any type)

Returns: (new value)

Example

The array called ListArray contains the following values:

{5,10,15}

To add a new element to this array the following Aadd parameters are used:

Aadd(ListArray,20)

and now ListArray is {5,10,15,20}. To add an array of new elements:

Aadd(ListArray,{25,30})

then ListArray becomes {5,10,15,20,{25,30}}

Related Functions

Also see the DelArray function.

AEval()

Description

Used to execute a code block for each element in an array.

Aeval passes the element value to the code block as an argument. The function makes no assumptions about the contents of the array elements it is passing. It is assumed that the supplied code block knows what type of data will be in each element. Also see the LimsCodeBlock function.

The array name, code block to execute, starting element, and the number of elements to process from the starting element (Count) are defined in the parameters. If Count is not defined, the default is all elements to the end of the array.

Parameters

Aeval(Array Name, Code Block, Start, Count)

Input Types: (array, Clipper code type, numeric, numeric)

Returns: (reference to the Array Name)

Notes:

A negative Start value starts from the end. If Count is positive, the default value is 1. If Count is negative, the default value is the length of the array. A negative Count value also starts from the end. The default is all elements to the end of the array.

Example

```
:DECLARE aArray, aResult;

aArray := {1,2,3};

aResult := {};

Aeval(aArray,{|X|  Aadd(aResult,X+1) } );

aResult will be  {2,3,4}
```

Related Functions

Also see the LimsCodeBlock() function.

AEvalA()

Description

Execute a code block for each element in an array and assign the return value to each element in the array.

AEvalA() is similar to AEval() in that they both evaluate a code block once for each element of an array, passing the element value as an argument. The difference is that while AEval() ignores the return value of the code block, AEvalA() assigns the return value to the array element. See AEval() for details.

Parameters

Aeval(Array, Code Block, Start, Count)

Input Types: (array, Clipper code type, numeric, numeric)

Returns: (reference to the Array)

Notes:

A negative Start value starts from the end. If Count is positive, the default value is 1. If Count is negative, the default value is the length of the array. A negative Count value also starts from the end. The default is all elements to the end of the array.

Example

```
:DECLARE aArray;

aArray := {1,2,3};

AevalA(aArray,{|X|  X+1 } );

aArray will be  {2,3,4}
```

Related Functions

Also see the LimsCodeBlock() function.

AEvalOld()

Description

Execute a code block for each element in an array.

AEvalOld() passes the element value and the element index as arguments. AEvalOld() makes no assumptions about the contents of the array elements it is passing. It is assumed that the supplied code block knows what type of data will be in each element.

AEval() passes the element value to the code block as an argument. Use the AEvalOld() function to also pass the element index or use AEvalA() to process individual elements.

Parameters

AEvalOld(<aArray>, <cbBlock>, [<nStart>], [<nCount>]) ---> aArray

<aArray> The array to traverse.

<cbBlock> The code block to execute.

<nStart> The starting element. A negative value starts from the end. If **<nCount>** is positive, the default value is 1; if **<nCount>** is negative, the default value is the length of the array.

<nCount> The number of elements to process from **<nStart>**. A negative value starts from the end. The default is all elements to the end of the array.

Input Types: (array, Code Block, numeric, numeric)

Returns: (reference to the Array)

Example

This next example changes the contents of the array element depending on a condition. If the condition is false, array elements are merely displayed. Notice the use of the code block arguments:

```
:DECLARE aArray;

aArray := {, , , , ,}

AFill     (aArray, "old")

AEvalOld   (aArray, {|cValue, nIndex| IF(cValue == "old", ArrayPut (aArray, nIndex,"new"),QOut(cValue))})
```

Related Functions

Also see the LimsCodeBlock() function.

AFill()

Description

AFill() fills the specified array with a single value of any data type by assigning <uValue> to each array element in the specified range.

Warning! Using AFill() with a multidimensional array could overwrite subarrays used for the other dimensions of the array.

Parameters

AFill(<aTarget>, <uValue>, [<nStart>], [<nCount>]) ---> aTarget

<aTarget> The array to fill.

<uValue> The value to place in each array element.

<nStart> The starting element. A negative value starts from the end. If **<nCount>** is positive, the default value is 1; if **<nCount>** is negative, the default value is the length of the array.

<nCount> The number of elements to process from **<nStart>**. A negative value starts from the end. The default is all elements to the end of the array.

Input Types: (array, any type, numeric, numeric)

Returns: (reference to the aTarget Array)

Example

This example creates a 3-element array. The array is then filled with the logical value .F.. Finally, elements in positions 2 and 3 are assigned the new value .T.:

```
:DECLARE aLogic;          // aLogic is {NIL}
aLogic := {.T., .T., .T.};
AFill(aLogic, .F.)        // aLogic is {.F., .F., .F.}
AFill(aLogic, .T., 2, 2)    // aLogic is {.F., .T., .T.}
```

This example fills up the individual rows of a multidimensional array:

```
:DECLARE a2;              //a 2-dimensional array
a2 := {{1,2,3},{4,5,6},{7,8,9}}
AFill(a2[1], "One")
AFill(a2[2], "Two")
AFill(a2[3], "Three")
```

Related Functions

Also see the LimsCodeBlock() function

ArrayCalc()

Description

Used to perform several calculation types to an array indicated by the Option keyword.

The array and option keyword are defined in the parameters.

Parameters

ArrayCalc(Array, Option, SecArray, Beg, Cnt)

Option Keywords include:

"MAX" Maximum value

"MIN" Minimum value

"SUM" Sum of the elements

"AVG" Average value – null or 0 (zero) values are excluded

"AVG1" Average value – used to include also the null or 0 (zero) values

"DEV" Standard deviation value – null or 0 (zero) values are excluded

"DEV1" Standard deviation value – used to include also the null or 0 (zero) values

"SORT" Sorts the Array

"MERGE" Concatenates two arrays

“COPY” Copies the elements of the first array into the second

“DUP” Creates a new array as a duplicate of the Array

“DEL” Deletes the element of the firs array specified by <Beg>

“ADD” Adds the two arrays (like Aadd())

“FILL” Fills the first array with the value specified in the second array. A starting position and a counter will be specified in the arguments <Beg> and <Cnt>. By default <Beg> is 1 and <Cnt> is the number of elements of the first array (<Array>).

“INS” INS inserts a new element into a specified array at the position specified by <Beg>.

“RESIZE” RESIZE changes the actual length of <Array>. The new length will be specified by <SecArray>

<SecArray> The <SecArray> parameter is the name of the other array that will be concatenated with the first. If the one of the following options MERGE,COPY is used, then the SecArray parameter is mandatory. This parameter may not always consist of an array (see the “RESIZE” option).

MIN and MAX options – The values in the array have to be numeric. This can be done using To_number in the Select statement or the Val() function.

COPY copies elements from <Array> to <SecArray>. <SecArray> must already exist and be large enough to hold the copied elements. If <Array>

has more elements, some elements will not be copied. COPY copies values of all data types. If an element of <Array> is a sub array, the corresponding element in <SecArray> will contain only a reference to the sub array. Thus, COPY will not create a complete duplicate of a multidimensional array. To do this, use the DUP option.

DUP creates a complete duplicate of <Array>. If <Array> contains subarrays, DUP creates matching subarrays and fills them with copies of the values in the <Array> subarrays.

INS inserts a new element into a specified array. The newly inserted element is NIL until a new value is assigned to it. After the insertion, the last element in the array is discarded, and all elements after the new element are shifted down one position.

Warning! INS must be used carefully with multidimensional arrays. Using INS in a multidimensional array discards the last element in the specified target array which, if it is an array element, will cause one or more dimensions to be lost. To insert a new dimension into an array, first add a new element to the end of the array using ADD or RESIZE before using INS.

RESIZE changes the actual length of <Array>. The array is shortened or lengthened to match the specified length (<Cnt>). If the array is shortened, elements at the end of the array are lost. If the array is lengthened, new elements are added to the end of the array and assigned NIL.

Input Type: (Array, String- Option keyword, Array or various, Integer, Integer)

Returns: (various)

Example

```
ArrayCalc({1,2,3},"MAX")
```

Returns: 3

```
ArrayCalc({1,2,3},"SUM")
```

Returns: 6

```
ArrayCalc({1,2,3},"AVG")
```

Returns: 2

```
ArrayCalc({1,2,3},”RESIZE”,4)
```

Returns: {1, 2, 3, NIL }

```
ArrayCalc({1,2,3},”INS”,2)
```

Returns: {1, NIL, 2 }

```
ArrayCalc({1,2,3},”FILL”,’TRUE’,2,2)
```

Returns: {1, TRUE, 2 }

Related Functions

ArrayNew()

Description

Create an un-initialized array with the specified number of elements and dimensions.

Parameters

ArrayNew(ElementList)

ElementList A comma-separated list representing the number of elements in each dimension. If more than one element number is specified, a multidimensional array is created with the number of dimensions equal to the number of arguments in ElementList.

Input Type: (list of integers)

Returns: (array)

Example

This example creates a one-dimensional array of five elements using the ArrayNew() function, then shows the equivalent action by assigning a literal array of NIL values:

```
aArray := ArrayNew(5);

aArray := {NIL, NIL, NIL, NIL, NIL};
```

This example shows two different statements which create the same multidimensional array:

```
aArray := ArrayNew(3, 2)

aArray := {{NIL, NIL}, {NIL, NIL}, {NIL, NIL}}
```

This example creates a multidimensional array:

```
aArray := ArrayNew(3, 2, 5)
```

For a better performance and reduced memory fragmentation follow these suggestions:

1. If the size of the array that should be created is known, use aMyArr:=ArrayNew(size) instead of aMyArr:={}

2. Use direct assignation instead of the AAdd function:

```
aMyArr:=ArrayNew(3);
aMyArr[1]:=1;
aMyArr[2]:=2;
aMyArr[3]:="a";
```

Instead of:

```
aMyArr:={};
Aadd(aMyArr,1);
Aadd(aMyArr,2);
Aadd(MyArr, "a");
```

3. If the size of the array is not known at the moment of the declaration, try to use a "maximum" size if can be approximated

4. Use the ArrayCalc() function with the FILL option to fill an array with the same value, instead of using a loop.

5. Try to use the function AevalA() to fill an array with variable information.

6. Use ArrayCalc with the COPY option to transfer elements from an array to another one.

7. Use ArrayCalc() with the RESIZE option, to truncate an array.

Related Functions

Ascan()

Description

Used to scan an array until a value is found or a code block returns TRUE.

If Search Value is not a code block, its value is compared to the first target element. If there is no match, the function proceeds to the next element in the array. This process continues until either a match is found or the range of elements to scan is exhausted.

If Search Value is a code block, the function scans the target array, executing the code block for each element accessed. As each element is encountered, **Ascan** passes the element's value as an argument to the code block, then evaluates the code block. The scanning operation stops when the code block returns TRUE or when the range of elements to scan is exhausted.

The target array, the value or code block to search for (Search Value), the starting element, and the number of elements to process from the starting element are defined in the parameters.

Parameters

Ascan(Target Array, Search Value, Start, Count)

Input Types: (any type - consistent with array type, string, numeric, numeric)

Returns: (If Search Value is a code block, returns the position of the first element for which the code block returns TRUE. Otherwise, returns the position of the first matching element. **Ascan** returns 0 if no match is found.)

> **Notes:** Unless the Search Value argument is a code block, it must match the data type of the elements in Target Array. A negative Start value starts from the end. If Count is positive, the default value is 1. If Count is negative, the default value is the length of the array. A negative Count value also starts from the end. The default is all elements to the end of the array.

Example

The following example shows how **Ascan** is used.

```
MyArray := {"John", "Sue", "Dave"};
Ascan(MyArray, "Sue")
```

Returns: 2

Related Functions

ASort()

Description

Sort an array.

ASort() sorts all or part of an array. The array may contains values (USUAL) of mixed types. Data types that can be sorted include character, date, logical, and numeric.

Strings are sorted in ASCII sequence; logical values are sorted with FALSE as the low value; date values are sorted chronologically; and numeric values are sorted by magnitude.

Warning! ASort() will not directly sort a multidimensional array. To sort a multidimensional array, you must supply a code block which properly handles the subarrays.

Parameters

ASort(<aTarget>, [<nStart>], [<nCount>], [<cbOrder>) ---> aTarget

<aTarget> The array to sort.

<nStart> The starting element. The default value is 1.

<nCount> The number of elements to process from **<nStart>**. The default is all elements to the end of the array.

<cbOrder> A code block used to determine the sort order. This argument is used to change the sorting order to descending or dictionary order. Each time it is evaluated, two elements from the target array are passed as arguments. The code block returns TRUE if the elements are in sorted order. See the examples below.

Input Types: (array, numeric, numeric, Code Block)

Returns: (reference to the Array)

Example

This example creates an array of five unsorted elements, sorts the array in ascending order, then sorts the array in descending order using a code block:

```
aArray := {3, 5, 1, 2, 4}

ASort(aArray)                    // {1, 2, 3, 4, 5}

ASort(aArray,,, {|x, y| x >= y})  // {5, 4, 3, 2, 1}
```

This example sorts a mixed-type array.

```
a := {"Z", "A", "one", 2, 1, "Three"}

ASort(a)
```

This example sorts an array of strings in ascending order, independent of case. It does this by using a code block that converts the elements to uppercase before they are compared:

```
aArray := {"Fred", "Kate", "ALVIN", "friend"}

ASort(aArray,,, {|x, y| Upper(x) <= Upper(y)}) // {ALVIN, FRED, FRIEND, KATE}
```

This example sorts a multidimensional array using the second element of each sub array:

```
aKids := {{"Mary", 14}, {"Joe", 23}, {"Art", 16}}
```

aSortKids := ASort(aKids,,, {|x, y| x[2] <= y[2]}) // Result:{{"Mary", 14}, {"Art", 16}, {"Joe", 23}}

Note: The "<" and ">" operators can be used in the code block if you are sure that there will be no duplicates; otherwise, it is more appropriate to use "< =" and "> =", as they properly allow for duplicate values.

Related Functions

BuildArray()

Description

Used to create an array from a delimited string.

Parameters

BuildArray(String, CR-Flag, Separator, U-Flag)

Input Types: (String, Logic, Char, Logic)

The last 3 parameters are optional, but if the CR-Flag is set to .F. all CTRL characters will be stripped. The default is .T.

The Separator is defaulted to “,” (comma).

U-Flag – Either .T. or .F. – If set to .T., only a set of unique values will be returned, otherwise if set to .F. all values will be returned.

Returns: (Array)

Example

The following **BuildArray** parameters,

BuildArray("Jim, John, Mary, Sue, Steve,"+Chr(13)+Chr(10)+" Julie, Jake,Jim,Jim",.F.,”,”,.T.);

Creates the array:

Jim
John
Mary
Sue
Steve
Julie
Jake

Related Functions

Also see the BuildArray2() function or the BuildArray() function.

BuildArray2()

Description

Used to create a 2 dimensional array. Does just the opposite of the BuildString2() function

Parameters

BuildArray(String, LineSeparator, ColSeparator)

Input Types: (String , Char, Char)

LineSeparator – defaulted to ";" (semi colon)

ColSeparator - defaulted to "," (comma)

The string that will be used to create a two dimensional array must follow the following structure.

All the data for the first row, must be separated by a comma as shown:

Barry,George,Sharon,Mary

Next comes a semi colon that shifts to the next row of data followed by the next row of data;

Barry, George, Sharon, Mary ; Male, Male, Female, Female

Returns: (two dimensional Array)

Example

The following **BuildArray2** parameters:

ATest := BuildArray2("Barry, George"+Chr(59)+"Male, Male", Chr(59), ",");

Chr(59) is used in stead of ";" because the StarLIMS compiler interprets an explicit semicolon as end of code line.

Creates the array aTest with the following data:

Barry	George
Male	Male

Related Functions

Also see the BuildArray() function or the BuildArraySQL() function

BuildArraySQL()

Description

Used to create an array from a *semi colon* delimited string.

Parameters

BuildArraySQL(string, CR-Flag, Separator)

Input Types: (string, Logic, Char)

Separator defaulted to ";" (semi colon)

CR-Flag – If set to .F. all CTRL characters will be stripped. The default is .T.

Returns: (array)

Example

The following **BuildArraySQL** parameters:

```
BuildArraySQL("Jim"+Chr(59)+"John"+Chr(59)+"Mary"+Chr(59)+"Sue"+C
hr(59)+"Steve"+Chr(59)+"Julie"+Chr(59)+"Jake");
```

Chr(59) is used in stead of ";" because the StarLIMS compiler interprets an explicit semicolon as end of code line.

Creates the array:

Jim
John
Mary
Sue
Steve
Julie
Jake

Related Functions

Also see the BuildArray() function or the BuildArray2() function

BuildString()

Description

Used to create a delimited string from an array.

Parameters

BuildString(Array, Start, Count, Delimitator)

Input Types: (array, numeric, numeric, string)

Returns: (comma delimited string)

nStart The starting element. A negative value starts from the end. If nCount is positive, the default value is 1; if nCount is negative, the default value is the length of the array.

nCount The number of elements to process from **nStart**. A negative value starts from the end. The default is all elements to the end of the array.

Delimitator The default value is comma (,)

Example

The following **BuildString** parameters:

```
aTest := {"Elian", "Grace", "Barry", "Gelu"};

BuildString(aTest,2,2)
```

Returns the comma delimited string:

```
Grace, Barry
```

Related Functions

See also the LFontName function or the ExecAction() function, or the BuildString() function, or the Lcase()- Obsolete function

BuildString2()

Description

Used to create a semi-colon and comma delimited string from a two dimensional array. The function takes the two dimensional array and places the information from the first row into the string and separates the data with a comma. It then separates the rows of data with a semi-colon. It will then add the next row of data as above and will continue to do so until the array has been emptied into string format. The is the opposite of the BuildArray2() function.

Parameters

BuildString2(array, LineSeparator, ColSeparator)

Input Types: (array, char, char)

Returns: (semi-colon, comma delimited string)

Example

The following **BuildString2** parameters:

```
BuildString2({{"Grace","Barry","Charley"},{"Yes","Yes","No"}});
```

Returns the semi-colon, comma delimited string:

```
"Grace,Barry,Charley;Yes,Yes,No"
```

Related Functions

See also the BuildArray2() function, or the BuildArray() function, or the BuildStringSQL() function.

BuildStringSQL()

Description

Used to create a semi colon delimited string from an array.

Parameters

BuildStringSQL(array, startIndex, Count)

Input Types: (array, numeric, numeric)

Returns: (semi colon delimited string)

Example

The following **BuildStringSQL** parameters:

```
aTest := {"Elian", "Grace", "Barry", "Gelu"};

BuildStringSQL(aTest,2,2)
```

Returns the comma delimited string:

```
Grace, Barry
```

Related Functions

See also the BuildArray2() function, or the BuildArray() function, or the BuildString2() function.

CompArray()

Description

Used to compare two arrays for matching content. This will compare both arrays and will check to ensure that the structures are the same and all matching elements are equal.

Parameters

CompArray(Array1, Array2)

Input Types: (array, array)

Returns: (Logical)

Example

To compare the contents of two arrays using **CompArray**, the parameters are:

Comp(Array1, Array2)

Which returns :

.T. where both arrays have the same structure and all matching elements are equal.

.F. where the arrays are different.

Related Functions

DelArray()

Description

Used to delete an element from an array. The array and the position number of the element in the array that you want to delete are defined in the parameters. This will reduce the array size by one, where all the elements after the deleted element will be moved up one position. For example, in an array of 5 elements, element 3 is deleted. Element 4 will now be element 3 and element 5 becomes element 4.

Parameters

DelArray(Array, Position Number)

Input Types: (array, numeric)

Returns: (modified array)

Example

To delete element 3 using **DelArray**, the parameters are:

DelArray({"Today", Today(),3.14,"Tomorrow",Today()+1},3)

Which returns the array:

"Today", Today(),"Tomorrow", Today()+1

Related Functions

Also see the Aadd function.

ExtractCol()

Description

Used to extract the column # of a specified array and return it as a single dimensional (vector) array. The array and the column # are defined in the parameters.

Parameters

ExtractCol(Array, Column #)

Input Types: (array, numeric)

Returns: (single dimensional array)

Example

To extract the second column of an array, the following **ExtractCol** parameters are used:

```
ExtractCol({{1,2},{3,4}},2)
```

which returns the single dimensional array: {2,4}

Related Functions

LimsCodeBlock()

Description

Used to execute a code block for each element in an array. **LimsCodeBlock** passes the element value to the code block as an argument. The function makes no assumptions about the contents of the array elements it is passing. It is assumed that the supplied code block knows what type of data will be in each element.

The array name, code block to execute, starting element, and the number of elements to process from the starting element (Count) are defined in the parameters. If Count is not defined, the default is all elements to the end of the array.

Parameters

LimsCodeBlock(Array Name, Code Block, Start, Count)

Input Types: (array, Clipper code type, numeric, numeric)

Returns: (reference to the Array Name)

Notes:

A negative Start value starts from the end. If Count is positive, the default value is 1. If Count is negative, the default value is the length of the array. A negative Count value also starts from the end. The default is all elements to the end of the array.

Example

```
:DECLARE aArray, aResult;

aArray := {a,b,c};

aResult:= {};

LimsCodeBlock(aArray,"X","Aadd(aResult,CHR(39)+ X +CHR(39))");

The aResult array will contain: {'a','b','c'}
```

Related Functions

SortArray()

Description

Used to sort an array in *ascending* order, such as 1-9 or A-Z.

Parameters

SortArray({Array})

Input Types: (array)

Returns: (sorted array)

Example

Using the **SortArray** function, the following array,

SortArray({23,2,14,16,44,8})

will be sorted as:

{2, 8, 14, 16, 23, 44}

Related Functions

Database Functions

BeginLimsTransaction()

Description

This function notifies the server to begin to record all transactions in its roll back segments. Used in conjunction with an EndLimsTransaction() function. This will allow the user to check the updating of the database by SQL statements, to find out if the updates were actually done by using the SQLExecute() function. When using this function, you must also use the EndLimsTransaction() function that will commit the changes that were made.

Parameters

BeginLimsTransaction(Database DSN)

Input Type: (string)

"DATABASE" or "DICTIONARY"

Returns: (logical)

Example

```
:DECLARE SQLOK, PRUNNO, ORDLIST;

PRUNNO := GetCurrent("SYSADM_272","PRUNNO");

ORDLIST := SqlExecute("Select DISTINCT ORDNO from ORDTASK
where SAMPTYPE is not null and PRUNNO=?PRUNNO? ");

ORDLIST := BuildString(ExtractCol(ORDLIST,1));

BeginLimsTransaction();

/* delete added controls;

SqlExecute( "Delete from ORDERS where SAMPTYPE is not null and
ORDNO in ("+ORDLIST+")" );

SqlExecute("Delete from ORDTASK where SAMPTYPE is not null and
ORDNO in ("+ORDLIST+")" ) ;

SqlExecute("Delete from RESULTS where SAMPTYPE is not null and
ORDNO in ("+ORDLIST+")");

SqlExecute("Update ORDTASK set PRUNNO=NULL,PCUPNO=NULL
where PRUNNO=?PRUNNO?");

EndLimsTransaction();

ExecInternal(LimsAPP("SYSADM_272"),"DELREC");
```

Related Functions

Also see the EndLimsTransaction() function, or the SQLExecute() function

EndLimsTransaction()

Description

Used to commits all transactions performed since the last BeginLimsTransAction().

Parameters

EndLimsTransaction(Database DSN)

Input Type: (string)

"DATABASE" or "DICTIONARY"

Returns: (logical)

Example

See the BeginLimsTransaction() function.

Related Functions

GetTableOwner()

Description

This function is available only on a Microsoft SQL Server database.

The function retrieves the owner of a certain table.

Parameters

GetTableOwner(Database DSN, TableName, Qualifier)

Input Types: (string, string, string-optional)

Returns: (string)

Example

```
GetTableOwner("MYDB", "ORDERS","PRODUCTION");
```

Returns "dbo" if the table exists within the database MYDB under the database name "PRODUCTION". This example indicates that "PRODUCTION" is a database name under a MSSQL database. (PRODUCTION.dbo.ORDERS)`

Related Functions

GetConnectionByName()

Description

This function returns the database or dictionary connection, receiving as a parameter the global identifier (“DATABASE”, “DICTIONARY”). This function was developed for actions that have to run in StarLIMS and also in the batch processor. There is a difference in the order of the connection in StarLIMS compared with the batch processor LimsBtch.exe, and previously, a certain connection was being retrieved from an array, based on an index. Because the order of the elements in this array is different in StarLIMS and batch, unpredictable results were produced or different actions were needed for each situation.

Parameters

GetConnectionByName(ConnectionIdentifier)

Input Types: (string)

Returns: (SQLConnection Object)

Example

```
GetConnectionByName(“DATABASE”);
```

Note: This function is to be used when the connection object is needed. To identify a connection for functions like SQLExecute, the string identifier is required, and not a connection object. The connection is resolved internally.

Related Functions

IsTable()

Description

Used to determine if a table exists in a named database source. The database source and table name are defined in the parameters. The Qualifier contains the schema name for Oracle or the database name for SQL Server. The fourth parameter indicates whether the third parameter indicates a schema name (Oracle) – (.T.) or a database name (SQL Server) – (.F.). The default value for this parameter is .T. - true

Parameters

IsTable(Database DSN, TableName, Qualifier, IsOwner)

Input Types: (string, string, string, logic)

Returns: (logic)

Example

IsTable("MYDB", "ORDERS",”LIMSUSA”);

Returns .T. if the table exists within the database MYDB under the schema/user “LIMSUSA” or .F. if it does not. This example indicates that LIMSUSA is a schema under a Oracle database. (LIMSUSA.ORDERS)

IsTable("MYDB", "ORDERS",”dbo”, .F.);

Returns .T. if the table exists within the database MYDB under the database name “PRODUCTION1” or .F. if it does not. This example indicates that PRODUCTION1 is a database name under a SQL Server database. (DEFAULT_DATABASE.dbo.ORDERS)

Related Functions

Also see the IsTableFld() function.

IsTableFld()

Description

Used to determine if a field within a table in a named database source exists. The database source, table name, and field name are defined in the parameters. . The Qualifier contains the schema name for Oracle or the database name (username) for SQL Server

Parameters

IsTableFld(Database, TableName, FieldName, Qualifier)

Input Types: (string, string, string, string)

Returns: (logic)

Example

IsTableFld("MYDB", "ORDERS", "DUEDATE",”LIMSUSA”)

Returns .T. if the field DUEDATE exists within the ORDERS table in the database MYDB, under the schema/user “LIMSUSA” or .F. if it does not.

Related Functions

Also see the IsTable() function.

LimsSetCounter()

Description

This function is used to generate unique values for columns in other tables. If we need for example a new ORDNO, there's no need anymore to lock the ORDERS table, to select MAX(ORDNO) + 1, to insert a new row with that value in the ORDNO field, and then to unlock the table. In fact, this approach is obsolete and should not be used any more. Instead, a new function does all this process for us. The function LimsSetCounter() is using the LIMSCOUNTERS table, which stores a row for each combination of table and column for which we need a unique value. In the example above, this table will contain a row for the ORDERS table and ORDNO field, with the associated value of the counter. If I request a new ORDNO value, the function will look in the table LIMSCOUNTERS for a record containing the table name "ORDERS" and the column name "ORDNO". If no one exists, zero will be returned. Otherwise, the current counter is extracted from the row, incremented, and then put back in the row. The incremented value of the counter (i.e. the new ORDNO) is returned to the caller, and if necessary a new row is inserted in the ORDERS table.

So what we have to do is populating LIMSCOUNTERS with information, and start using the two function

In order to use this function create a table called LIMSCOUNTERS under your User Schema(Account). Use this SQL statement to do that:

Create Table LIMSCOUNTERS

```
(
  FLDNAME     VARCHAR2(30),
  LIMSCOUNTER NUMBER(8,0),
  PREFIX      VARCHAR2(40),
  TABLNAME    VARCHAR2(30),
  ORIGREC     NUMBER(8,0)
)
```

Parameters

LIMSSetCounter(TableName, FieldName, Prefix, {Array of Fields to use in the Insert Into statement}, {Array of the Field values for the same Insert Into Statement});

Input: (string, string, string, array, array)

TableName - Actual table required a counter for one of the fields

FieldName - The field referred as COUNTER (Unique calculated value)

Prefix - the prefix, if any. For a numeric column, this parameter is skipped. For a string column, it is something like "Folder-" for example.

In case the user wants to calculate the next value and Insert the record as well use :

Array of fields - array of strings containing column names (for example {"APPRDATE","APPRSTS","CLIENTID"}). Besides returning the next counter, this function also creates a new row in the table named by Table Name.

Array of Values - array of values for the fields described above. The fields named in the previous array will receive the values in this array respectively. For the example above, a possible array of values could be {Today(),"Logged","ACMElabs"}. Each value must be formatted to match field data type

NOTE: If we want no row inserted in the TableName table, we have to supply arrays of different lengths. In this case, we have to do the insert myself, at a later moment. An example could be:

```
nNextCounter := LimsSetCounter( "FOLDERS",

"FOLDERNO",

"Folder-",

{"NO_INSERT"});
```

In this case, the first array has the length one (a dummy array with one element called "NO_INSERT"), and the second array has the length zero (it's not even sent as a parameter). In this case, we have to make the insert in the FOLDERS table later. We have to concatenate the prefix ("Folder-") with whatever greater than zero value was returned by LimsSetCounter(), and use the resulting string for the FOLDERNO column.

If we do want the insert done for us, we should code something like this:

```
nNextCounter := LimsSetCounter( "FOLDERS",

"FOLDERNO",

"Folder-",

{"APPRDATE","APPRSTS","CLIENTID"},

{Today(),"Logged","ACMElabs"});
```

Returns: (integer).

If the function returns zero, this means the function call was unsuccessful. If greater than zero, I can safely use this value as a unique counter. If I also sent a prefix as a parameter, I have to concatenate that prefix with this number as a string, in order to obtain the whole column's value.

Example

```
nNextCounter := LimsSetCounter( "FOLDERS",

"FOLDERNO",

"Folder-",

{"APPRDATE","APPRSTS","CLIENTID"},

{Today(),"Logged","ACMElabs"});
```

- gets the new counter for FOLDERNO and inserts a new record into FOLDERS. The fields and values for the insert are specified in the parameters.

or

```
nNextCounter := LimsSetCounter( "FOLDERS",

"FOLDERNO",

"Folder-",

{"NO_INSERT"});
```

- gets a new value for the field FOLDERNO without performing an insert into FOLDERS table

Related Functions

LimsSQLDisconnect()

Description

Used to disconnect from an SQL ODBC source from within an action.

Parameters

LimsSQLDisconnect(String);

Input: (string)

where: String is the ODBC connection name

Returns: (logical)

Example

```
LimsSQLDisconnect("WABC");
```

This will disconnect from the ODBC source named WABC when executed.

Related Functions

LimsSQLGetConnect()

Description

Used to return an array of all the currently active data source (ODBC) connections.

Parameters

LimsSQLGetConnect()

Input (none)

Returns: (array)

Example

```
CURCON :=  LimsSQLGetConnect()
```

Related Functions

LSelect()

Description

Used to run a select SQL command, with parameters and create an array from records and fields selected from a data base table. Every row identifies a record and every column identifies a field from the original table.

The function uses an SQL Select expression to find specific records and returns the values for the fields that are specified in the second parameter. The SQL Select expression, array of fields, database and parameters are specified as parameters. The database and Parameters (fourth parameters) is optional.

The SQL Select statement send as the first parameters can contain the question mark sign, that will be replaced during execution of the statement with the corresponding value from the fourth parameter. So send the corresponding values to be replaced in the SQL Select statement, in an array, in the order in which the ? appear in the statement.

Parameters

LSelect(SelectCommand, FieldList, ODBCsource, Parameters)

The placement of the question marks '**?**' in the SQL statement will be replaced by the parameters placed into the array. If the '**?'** is used as a replaceable variable, then the number of '**?**' marks must be exactly the same as the number of elements in the array.

Input Types: (string, array, string, array)

Select Command – a string value containing an SQL select statement that can contain ? question marks where values will be substituted

FieldList – an array of string values indicating the field names for field names

ODBCsource – a string: ODBC Data Source Name (DSN) and can be a valid DSN name registered in the system or either “DATABASE” or “DICTIONARY”. Optional. The default is DATABASE

Parameters – an array of values that will replace the ? in the SQL statement in the order in which they are defined. Optional, but are required if the SQL statement contains ?

Returns: (two dimensional array)

If the SQL select statement completes successfully, the matrix will be returned, otherwise returns an empty array.

Example

```
:DECLARE aRet, sVar;

sVar := "TG";

aRet := LSelect("Select * from INSTR where INSTR = ?",
                {"INSTR",  "DESCR"},
                 "DATABASE",
                 {sVar});
```

This will return a 2 column array listing records whose INSTR value is equal to the parameter variable sVar. The question mark inside the **SelectCommand** represents a placeholder for the first parameter in the **Parameters** array. The second array, **FieldList** represents a list of fields that will become the columns in the 2-dimensional array aRet. The third parameter represents the ODBC Data Source Name (DSN) and can be a valid DSN name registered in the system or the strings “DATABASE” or

"DICTIONARY".

The difference between **LSelect()** and **LSelect1()** is that in the second function the number of columns returned is influenced only by the number of fields selected in the **SelectCommand** whereas in **LSelect()** the user can further limit the number of fields selected through the **FieldList** parameter. Another difference between the two is that **LSelect1()** brings more performance but can only be used to select 9999 records.

The **Lselect()** function:

```
Lselect("Select * from EMPLOYEES where SEX = 'M'",{"Name", "Age"})
```

Returns a two dimensional array of all male employees' names and ages, as partially shown in the following table:

Name	**Age**
Doe, John	45
Stuart, Jimmy	55
Dumpty, Humpty	30
Pan, Peter	25
Levi, Dave	40

The array elements are assigned the same types as they are in the Employees table. Therefore, Name and Age are character types.

Related Functions

LSelect1()

Description

Used to create an array from records and fields selected from a data base table.

Performs an SQL command.

Parameters

LSelect1(SelectCommand, ODBCsource, Parameters)

Input Types: (string, string, array)

Returns: (two dimensional array)

If records are found, returns an array of length n if n records were found.

Example

```
:DECLARE aRet, sVar;

sVar := "TG";

aRet := LSelect("Select * from INSTR where INSTR = ?", "DATABASE",
{sVar});
```

This will return a 2 column array listing records whose INSTR value is equal to the parameter variable sVar. The question mark inside the **SelectCommand** represents a placeholder for the first parameter in the **Parameters** array. The second parameter represents the ODBC Data Source Name (DSN) and can be a valid DSN name registered in the system or the strings “DATABASE” or “DICTIOANARY”.

The difference between **LSelect()** and **LSelect1()** is that in the second function the number of columns returned is influenced only by the number of fields selected in the **SelectCommand** whereas in **LSelect()** the user can further limit the number of fields selected through the **FieldList** parameter. Another difference between the two is that **LSelect1()** brings more performance but can only be used to select 9999 records.

Related Functions

Lsearch()

Description

Used to search for a field value. The function uses an SQL Select expression to find a specific field value and returns this value. If the field value does not exist, the default value is returned. This function is similar to the Lselect() function, except instead of returning an array, a single value is returned. The SQL Select expression, default value, and database are specified in the parameters. The database parameter is optional.

Parameters

Lsearch(SELECT Expression, Default, Database)

Input Types: (string, any type, string)

Returns: field or default value

Example

The **Lsearch** function with the parameters:

Lsearch("Select MATNO from MATERIAL where ORIGREC = 5","No Material", "DATABASE")

Returns the MATNO for ORIGREC =5, or "No Material".

Related Functions

Also see the Lselect() function.

RunSQL()

Description

Performs an SQL statement.

Parameters

RunLScript(sSQLStatement, sConnection, aParameters)

sSQLStatement - String **-** contains the SQL statement to be executed.

sConnection – String - contains the ODBC connection name (can also be “DICTIONARY” and “DATABASE”)

aParameters – Array – can be a multiple dimension array that contains values for the parameters indicated by question marks in sSQLStatement.

Input (string, string, array)

Returns (logic) .T. or .F. depending on the success of the operation.

The role of the aParameters array is to contain values for the parameters indicated by question marks in sSQLStatement. This array can have multiple dimensions and this means that the same SQL Statement will be executed with different parameters for each of the elements of the array. This increases the performance of the function, certain tasks inside the function are being done only once, so instead of running RunSQL in a loop and changing the values of a single element array of parameters, the function can be ran only once with a multi-dimension array of parameters.

Example

RunSQL(“Insert into TESTS(TESTNO,TESTCODE,TESTDESC) values(?, ?, ?)”,**"**DATABASE**", {{“Test1”, 334, “This is a nice test”}, {“Test2”, 335, “This is a nice test”}, {“Test3”, 336, “This is a nice test”}})**;

The SQL statement : “Insert into TESTS(TESTNO, TESTCODE, TESTDESC) values(?, ?, ?)” is executed 3 times (the number of parameter arrays in aParameters array). Each time the values of the parameters indicated by (?) in the statement is changed with the values contained in the next element of aParameters.

This function replaces a WHILE loop like this one:

```
aParameters := {{“Test1”, 334, “This is a nice test”}, {“Test2”, 335, “This is a nice test”}, {“Test3”, 336, “This is a nice test”}};

:WHILE i<= Len(aParameters);

      RunSQL( sSQLStatement, “DATABASE”, aParameters[i]);

      i := i +1;

:ENDWHILE;
```

Related Functions

SQLExecute()

Description

Used to run an SQL statement in an action and return a logical value whether or not the statement completed successfully or not.

Parameters

SQLExecute(SQL Statement, Datasource)

Input Types: (string, string)

Returns: Logical

Datasource is optional.

Example

The SQL Statement may contain variables previously defined. These variables need to be enclosed between question marks (**?**). At run-time these variables will be replaced by their value.

Example #1:

```
SqlExecute("Select ORDNO from ORDTASK where PRUNNO =
?PRUNNO? ");
```

Example #2:

```
:DECLARE SQLOK, PRUNNO, ORDLIST;

PRUNNO := GetCurrent("SYSADM_272","PRUNNO");

ORDLIST := SqlExecute("Select DISTINCT ORDNO from ORDTASK
where SAMPTYPE is not null and PRUNNO=?PRUNNO? ");

ORDLIST := BuildString(ExtractCol(ORDLIST,1));

BeginLimsTransaction();

/* delete added controls;

SqlExecute( "Delete from ORDERS where SAMPTYPE is not null and
ORDNO in ("+ORDLIST+")" );

SqlExecute("Delete from ORDTASK where SAMPTYPE is not null and
ORDNO in ("+ORDLIST+")" ) ;

SqlExecute("Delete from RESULTS where SAMPTYPE is not null and
ORDNO in ("+ORDLIST+")");

SqlExecute("Update ORDTASK set PRUNNO=NULL,PCUPNO=NULL
where PRUNNO=?PRUNNO?");

EndLimsTransaction();

ExecInternal(LimsAPP("SYSADM_272"),"DELREC");
```

Example #2

```
:IF SQLExecute( "Update ANALYSTS set CITY = ?MYCITY?, STATE =
    ?MYSTATE? where ORIGREC INCURRENTSELECTION",
    "DATABASE");
        Branch("LABEL_GOOD");
:ELSE Branch("LABEL_BAD");
:ENDIF;
```

Where: MYCITY and MYSTATE are predefined variables.

Or

```
:IF SQLExecute( "Update ANALYSTS set CITY = 'Ft. Lauderdale', STATE
    = "Florida' where ORIGREC in CURRENTSELECTION"',
    "DATABASE");
        Branch("LABEL_GOOD");
:ELSE Branch("LABEL_BAD");
:ENDIF;
```

Related Functions

ODBC Extensions

Use ODBC extensions to write one SQL statement for all database platforms supported by STARLIMS.

String	Numeric	Timedate	System
ASCII	ABS	CURDATE	IFNULL
CHAR	ACOS / ASIN	CURTIME	USER
CONCAT	ATAN / ATAN2	DAYNAME	
INSERT	CEILING	DAYOFMONTH	
LCASE	COS / SIN	DAYOFWEEK	
LEFT	COT	DAYOFYEAR	
LENGTH	EXP	HOUR	
LOCATE	FLOOR	MINUTE	
LTRIM	LOG / LOG10	MONTH	
REPEAT	MOD	MONTHNAME	
RIGHT	PI	NOW	
RTRIM	POWER	QUARTER	
SOUNDEX	ROUND	SECOND	
SPACE	SIGN	WEEK	
SUBSTRING	SQRT	YEAR	
UCASE	TAN		

Date and time data extensions

Description

The escape clauses ODBC uses for date and time data

Parameters

{d 'value'}

{t 'value'}

where **d** indicates *value* is a date in the "yyyy-mm-dd" format, **t** indicates *value* is a time in the "hh:mm:ss" format

Example

```
:DECLARE strSQL;
/*the old way;
/*:BEGINCASE;
/*
/*    :CASE PLATFORMA=="ORACLE";
/*          strSQL := "select * from folders where
logdate = '12-JAN-05'";
/*    :EXITCASE;
/*
/*    :CASE PLATFORMA=="MSSQL" .or.
PLATFORMA=="SYBASE";
/*          strSQL := "select * from folders where
logdate = '2005-01-12'";
/*    :EXITCASE;
/*
/*:ENDCASE;

/*the new way;
strSQL := "select * from folders
           where logdate = {d '2005-01-12'}";


 :RETURN GetDataSet(strSQL);

 /************************************;

 select {fn CURTIME()} DATETIMEFLD from USERS
```

Related Functions

Scalar functions

Description

Scalar function – such as string length, absolute value, or current date – can be used on columns of a result set and columns that restrict rows of a result set.

An application can mix scalar functions that use native syntax and scalar functions that use ODBC syntax

Parameters

{fn *scalar-function*}

where **d** indicates *value* is a date in the “yyyy-mm-dd” format, **t** indicates *value* is a time in the “hh:mm:ss” format

Example

```
:DECLARE strSQL;
/*the old way;
/*:BEGINCASE;
/*
/*    :CASE PLATFORMA=="ORACLE";
/*          strSQL := "select Client || ' -- ' ||
City as Contact from clients";
/*    :EXITCASE;
/*
/*    :CASE PLATFORMA=="MSSQL" .or.
PLATFORMA=="SYBASE";
/*          strSQL := "select  Client + ' -- ' +
City as Contact from clients";
/*    :EXITCASE;
/*
/*:ENDCASE;

/*the new way;
strSQL := "select {fn CONCAT(Client , {fn CONCAT(' -
- ', City)})} as Contact from clients";

:RETURN GetDataSet(strSQL);



 /*********************************************/
select {fn ucase(USRNAM)} as Name, {fn
length(USRNAM)} as Length from USERS

select {fn concat(CLIENT , {fn concat(' -- ',
CITY)})} as Contact from CLIENTS
select {fn CONCAT( {fn ifnull(Client, 'STARLIMS
Corp.')} , {fn CONCAT(' -- ', City)})} as Contact
from clients


select FOLDERNO, {fn CURDATE()} as CurDate from
FOLDERS

select {fn substring(CLIENT, 1, charindex(',',NAME)-
1)} as Client from CLIENTS
```

ODBC defines a special scalar function, **CONVERT**, that requests that the data source convert data from one SQL data type to another SQL data type.

Supported data types:

SQL_CHAR
SQL_VARCHAR
SQL_INTEGER
SQL_LONGVARBINARY
SQL_NUMERIC
SQL_LONGVARCHAR
SQL_DATE

```
select CLIENT, CLDISCNT from CLIENTS where {fn
convert(CLDISCNT, SQL_CHAR)}='0'
```

In a **LIKE** predicate, the percent character (%) matches zero or more of any character and the underscore character (_) matches any one character. The percent and underscore characters can be used as literals in a **LIKE** predicate by preceding them with an escape character.

```
select CLIENT from CLIENTS where CLIENT LIKE
'\%AAA%' {escape '\'}
```

ODBC supports the ANSI SQL-92 left outer join syntax. ODBC uses four outer join is:

{**oj** *outer-join*}

where *outer-join* is:

table-reference **LEFT OUTER JOIN** {*table-reference* | *outer-join*} **ON** *search-condition*

```
select distinct TESTS.TESTCATCODE, TESTS.TESTNO,
RESULTS.ANALYTE
 from {oj TESTS left outer join RESULTS on
 TESTS.TESTCODE = RESULTS.TESTCODE}
 order by TESTS.TESTCATCODE, TESTS.TESTNO
```

Related Functions

Runtime Functions to get/return a DATASET

GetDataSet()

Description

Executes the SQL statement with the parameters, on the DATABASE and returns the result of the statement as a dataset, with or without the schema.

Parameters

GetDataSet(strSQL, arrParamsValues, bWithSchema)

Input Type: (string, array, boolean)

- strSQL (string): SQL Statement to be executed
- arrParamsValues (array): array with parameters' values
- bWithSchema (bool): generates also the schema for dataset, default is TRUE

Works only on the DATABASE

Returns: (dataset)

Example

```
:PARAMETERS MATCODE:='', STARTDDATE:='',
EXPDATE:='';
:DECLARE strSQL, strToday;

strSQL := "SELECT *
  FROM MFGINSTRUCTIONS
  WHERE (EXPDATE is NULL and MATCODE=?) or
 (MATCODE=? and STARTDDATE <= ? and EXPDATE >= ? )";


 :RETURN GetDataSet( strSQL, { MATCODE, MATCODE,
 Today(), Today() });
```

Related Functions

GetDataSetEx()

Description

Parameters

GetDataSetEx(strSQL, strConnection, arrParamsValues, bWithSchema, bHeader)

Input Type: (string, string, array, boolean, boolean)

- strSQL (string): SQL Statement to be executed
- strConnection (string): "DATABASE" or "DICTIONARY"
- arrParamsValues (array): optional. array with parameters' values
- bWithSchema (bool): optional. generates also the schema for dataset, default is TRUE
- bHeader (bool): optional. if FALSE then returns a string only with fields' values

<xml…>

<Dataset>

<Schema>

</Schema>

<Table>

<Field1>value</Field1> …. Fields' values section

<Field2>value</Field2>

</Table>

</Dataset>

This is useful if you want to return a dataset with multiple tables(for creating a hierarchical grid)

Works only on the DATABASE

Returns: (dataset)

Example

```
:RETURN GetDataSetEx( "select * from
limsEnterpriseSettings order by SettingName",
"DICTIONARY" );
```

Related Functions

GetDataSetFromArray()

Description

Returns a dataset from an array

Parameters

GetDataSetFromArray(arrValues, arrFields)

Input Type: (array, array)

- arrValues (array): 2 dimensions array with values
- arrFields (array): optional. array with fields informations; can be a 1 dimension array with only fields names, or each element in this array can be the following array {name, type, length, scale}

  type can be D – date, C – char(string), N – numeric

  if arrFields is empty then the fields will be named as Field1, Field2…..

Returns: (dataset)

Example

```
:DECLARE arayOfData;
arayOfData:={};

Dummy:=AaDd(arayOfData,{1,1,"S01","RED","SAMPLE01","
This Is A Sample"});
Dummy:=AaDd(arayOfData,{2,1,"S01","RED","SAMPLE012",
"This Is A Sample3"});
Dummy:=AaDd(arayOfData,{3,1,"S01","RED","SAMPLE011",
"This Is A Samplede3"});
Dummy:=AaDd(arayOfData,{10,1,"S01","BLUE","SAMPLE01"
,"This Is A Sample4"});
Dummy:=AaDd(arayOfData,{11,1,"S01","BLUE","SAMPLE013
3","This Is A Sample6"});
Dummy:=AaDd(arayOfData,{12,1,"S01","BLUE","SAMPLE015
","This Is A Sample7"});
Dummy:=AaDd(arayOfData,{10,2,"S01","GREEN","SAMPLE01
6","This Is A Sample4"});
Dummy:=AaDd(arayOfData,{11,2,"S01","GREEN","SAMPLE01
7","This Is A Sample7"});
Dummy:=AaDd(arayOfData,{12,2,"S01","GREEN","SAMPLE01
8","This Is A Sample8"});

 :RETURN GetDataSetFromArray(arayOfData);
```

Related Functions

GetDataSetFromArrayEx()

Description

Parameters

GetDataSetFromArrayEx(arrValues, arrFields, strTableName, bHeader)

Input Type: (array, array, string, boolean)

- arrValues (array): 2 dimensions array with values
- arrFields (array): array with fields informations; can be a 1 dimension array with only fields names, or each element in this array can be the following array { name, type, length, scale }

  type can be D – date, C – char(string), N – numeric

  if arrFields is empty then the fields will be named as Field1, Field2…..
- strTableName (string) : table name in dataset, default is Table
- bHeader (bool): if FALSE then returns a string only with fields' values

Returns: (dataset)

Example

Related Functions

RunDS()

Description

Executes a STARLIMS v10 datasource script.

Parameters

RunDS(dsName, arguments, xml)

Input Type: (string, array, string)

- dsName (string): data source name: CategoryName.DataSourceName
- arguments (array) : values to pass to datasource
- xml (bool): if true returns a string representing a dataset, else returns an array with values(like SQLExecute); default is false

If datasource is of type STARLIMS then this function returns whatever the datasource is returning and 'xml' parameter is ignored.

Returns: (dataset)

Example

```
RunDS("ENTERPRISE_SUPPORT.GetPendingCheckinsByUser",
{MYUSERNAME}, .T. );
```

Where ENTERPRISE_SUPPORT.GetPendingCheckinsByUser is a datasource.

Related Functions

Data Type Functions

IntToPtr()

Description

Used to convert an integer to a pointer. It returns a pointer.

Parameters

IntToPtr(intValue)

where: **intValue** an integer

Input (integer)

Returns (pointer)

Example

IntToPtr(25) returns 0x00000019

Related Functions

IsHex()

Description

Used to check if a string contains only hexadecimal characters. Valid characters are 0-9 and A-F. The return values are:

T if the characters are valid

F if the characters are not valid

Parameters

IsHex(cStr)

where: cStr a string

Input (string)

Returns (logical)

Example

IsHex("0AF")

Returns a T (true).

IsHex("0AS")

Returns a F (false).

Related Functions

LFromHex()

Description

Used to convert a hexadecimal string to a regular string.

Parameters

LFromHex(hcStr)

where: hcStr a hexadecimal string

Input (hexadecimal string)

Returns (string)

Example

LFromHex("616263646566") returns "abcde"

Related Functions

LHex2Dec()

Description

Used to receive a hexadecimal string and converts it to a decimal string.

Parameters

LHex2Dec(cHexNum)

where: cHexNum a hexadecimal string

Input (hexadecimal string)

Returns (decimal string)

Example

LHex2Dec("0A") returns 10

Related Functions

LimsConvert()

Description

This function is used in conversions between a binary values read from a file as an alfa-numeric strings and other specified data types.

Typical applications include reading foreign file types in their native format and then saving, reading, decrypting, and transmitting date types in their compressed binary form instead of in strings

Parameters

LimsConvert(sType, uData)

sType – string – indicates the requested data type for the conversion.

uData – any type – the data that needs to be converted converting.

Input (string, any type – see the table)

Returns (any type)

sType	uData Type	Action	Result Type
“DATE”	String - A 32-bit binary date represented as a string — least significant byte first. Only the first 4 bytes are used by the function; all others are ignored.	Convert a string containing a 32-bit binary date to a date data type.	A date value that corresponds to the date specified in <uData>. If <uData> is not a valid binary date, LimsConvert() returns a NULL_DATE.
“STRING”	Date - The date value to convert.	Convert a date to a 32-bit binary date string.	A 4-byte string
“DWORD”	String - A 32-bit unsigned integer represented as a string — least significant byte first. Only the first 4 bytes are used by the function; all others are ignored.	Convert a string containing a 32-bit unsigned integer to a double word.	double word
“DSTRING32”	Numeric Value - Decimal digits are truncated.	Convert a double word to a string containing a 32-bit unsigned integer	A 4-byte string containing a 32-bit unsigned integer.

“FLOAT”	String - An 80-bit floating point number represented as a string — least significant byte first.	Convert a string containing a 80-bit floating point number to a float value	floating point number.
“STRING80”	Float	Convert a float to a string containing an 80-bit floating point number.	A string representing a floating point number.
“SHORT”	String - a 16-bit signed integer represented as a string — least significant byte first. Only the first 2 bytes are used by the function; all others are ignored.	Convert a string containing a 16-bit signed integer to a short integer	Short Integer
“STRING16”	Short Integer	Convert a short integer to a string containing a 16-bit signed integer.	String containing a 16-bit signed integer
“LOGIC”	String - an 8-bit logical represented as a string.	Convert a string containing an 8-bit logical into a logical value.	Logical
“STRING”	Logic	Convert a logical value to a string containing an 8-bit logical value.	string containing an 8-bit logical value

“REAL4”	String - a 32-bit floating point number represented as a string — least significant byte first. Only the first 4 bytes are used by the function; all others are ignored.	Convert a string containing a 32-bit floating point number to a Real4 value.	Real4
“RSTRING”	Real4	Convert a Real4 value to a string containing a 32-bit floating point number.	String containing a 32-bit floating point number.
“WSTRING”	String	Convert a word to a string containing a 16-bit unsigned integer	String containing a 16-bit unsigned integer.
“LSTRING”	Long Integer	Convert a long integer to a string containing a 32-bit signed integer	String containing a 32-bit signed integer.

Example

```
:DECLARE x;

x := LimsConvert("STRING", CtoD("01/01/2001"));

usrmes( , LimsConvert("DATE", x));
  :RETURN;
```

Related Functions

LimsSymbol()

Description

Used to receive a character string and returns a visual object symbol.

Parameters

LimsSymbol(cStr)

where: cStr a character string

Input (string)

Returns (VO symbol)

Example

LimsSymbol("XXX")

Returns #XXX – VO symbol

Related Functions

LimsType()

Description

Used to determine the data type of an expression represented as a string where (string) contains an expression whose type is to be determined.

Parameters

LimsType(String)

where:

(String) can not contain an undeclared variable or function(s) that are not intended for use with macros

Input Types: (string)

Returns: If (String) does not exist, "U" is returned

Return	Meaning
A	Array
B	Block
C	String
D	Date
L	Logical
M	Memo
N	Numerical
O	Object
U	Nil, Local, or static
UE	error, syntactical
UI	error, indeterminate

Note:

Reference to private and public arrays returns an ‘A’. Reference to an array element returns the type of the element.

Example

```
:DECLARE uVar1;

sVar1 := “5AB”;

UsrMes("Type", LimsType(“sVar1”));
```

Displays a ‘C’

or

```
sVar1 := 567;

UsrMes("Type", LimsType(“sVar1”));
```

Displays an ‘N' because all characters are numbers.

Related Functions

LimsTypeEx()

Description

Used to return the data type of the argument. The following values are returned:

"NIL"	if the argument is undefined
"ARRAY"	if the argument is an array
"CODEBLOCK"	if the argument is a code block
"DATE"	if the argument is a date
"NUMERIC"	if the argument is numeric
"LOGIC"	if the argument is a logic (?)
"OBJECT"	if the argument is an object
"PTR"	if the argument is a pointer (?)
"SYMBOL"	if the argument is a symbol
"STRING"	if the argument is a string

Parameters

LimsTypeEx(Arg)

where: arg the argument

Input (variable)

Returns (string)

Example

```
:DECLARE aArray;

aArray := {1,2};

LimsTypeEx(aArray)  returns “ARRAY”

LimsTypeEx(aArray[1]) returns “NUMERIC”
```

Related Functions

LToHex()

Description

Used to convert a regular string to a hexadecimal string.

Parameters

LToHex(cStr)

where: cStr a regular string

Input (hexadecimal string)

Returns (string)

Example

LToHex("abcde")

Returns "616263646566"

Related Functions

PtrToInt()

Description

Used to convert a pointer to an integer. (opposite of IntToPtr)

Parameters

PtrToInt(pointer)

where: **pointer** a pointer

Input (pointer)

Returns (integer)

Example

:DECLARE pVar;

pVar := IntToPtr(25);

PtrToInt(pVar) returns 25

Related Functions

Date Functions

BaseYear()

Description

Used by the system to determine the base year for assigning the century portion of a date / year where a two digit year is entered.

This is normally used at initialization of the system setting to determine the cut off year for establishing what century to assign to a two digit year that is entered as part of a date.

If the year 1950 is used, any two digit year that is entered from 00 to 50 will automatically be assigned the next century. Any two digit year from 51 to 99 will assume the current century.

If the year is not passed as a parameter, the function returns the current base year.

Parameters

BaseYear(YEAR-optional)

Input (numeric)

Returns: (numeric)

Example

```
BaseYear(1950)
```

If the two digit year 48 is part of a date, the system will automatically convert it to 2048. If the two digit year 51 is entered as part of a date, the system will convert it to 1951.

Related Functions

Cmonth()

Description

Used to extract the name of the month from a date. It is a date conversion function used to create formatted date strings for reports, labels, screens, etc.

The date from which you want to extract the month from is defined in the parameters.

Parameters

Cmonth(Date)

Input Types: (date)

Returns: (Name of the month, where the first letter is uppercase and the rest of the string is lowercase. For an invalid or NULL_DATE, a NULL_STRING is returned.)

Example

```
Today()
```

Returns: 05/16/96

```
Cmonth(Today())
```

Returns: May

```
Cmonth(Today() + 17)
```

Returns: June

Related Functions

CtoD()

Description

Used to convert a date string to date format.

This function is a character conversion function that converts a date value originally formatted as a string to a date data type. **CtoD** is the inverse of DtoC(), which converts a date value to a string.

The date string is defined in the parameters.

Parameters

CtoD(DateString)

Input Types: (**string**) (A string of numbers representing the month, day, and year, separated by any character other than a number.)

Returns: (The date value that corresponds to the numbers specified in DateString. If DateString is not a valid date, a NULL_DATE is returned.)

Example

These examples show how **CtoD** converts a string into a date type:

CtoD("05/15/96")

Returns: 05/15/96

CtoD("03.20.96")

Returns: 03/20/96

Related Functions

Also see the DtoC() function or the DtoS() function

DateFormat()

Description

Used to set a *global* date format for the system, overwriting the system's default date format. To activate this function when opening StarLIMS, add an auto action to the Shell Window. The date format is defined in the parameters.

Parameters

DateFormat(Format)

Examples of Format are:

"YY-MM-DD"

"YY-MMM-DD" (Oracle Default Format)
"YYYY-MM-DD"
"MM-DD-YY"

"MM-DD-YYYY"
"DD-MM-YY" (MSSQL Default Format)

"DD-MM-YYYY" (MSSQL Format)

Input Types: (string)

Returns: (empty string)

Example

Using the following **DateFormat** parameters in an auto action in the Shell Window globally changes the system's date format to DD-MM-YY.

```
DateFormat("DD-MM-YY")
```

Related Functions

Also see the LimsDate() function

Day()

Description

Used to extract the number of the day of the month from a date. It is a date conversion function that is used when you need a numeric day value during calculations for such things as periodic reports.

Day is a member of a group of functions that return components of a date value as numbers. The group includes Month() and Year() to return the month and year values as numbers. (See the Month() function and Year() function.)

The date from which you want to extract the day from is defined in the parameters.

Parameters

Day(Date)

Input Types: (date)

Returns: (The day of the month, as a number in the range 0 to 31. For an invalid or NULL_DATE, 0 is returned.)

Example

The following examples show the **Day()** function used several ways:

Today()

Returns: 04/29/96

Day(Today())

Returns: 29

Day(Today()) + 1

Returns: 30

Day(0.0.0)

Returns: 0

This example uses Day() function in combination with Cmonth() function and Year() function to format a date value:

Cmonth(Today()) + Str(Day(Today())) + ', '+ Str(Year(Today()))

Returns: April 29, 1996

Related Functions

Also see the DoW() function, or the Month() function or the ValidDate(sDate,Flag) function or the Year() function

DoW()

Description

Used to extract the number of the day of the week from a date. This function is a date conversion function that converts a date value to a number identifying the day of the week. It is useful when you want date calculations on a weekly basis. The date from which you want to extract the day of the week from is defined in the parameters.

Parameters

DoW(Date)

Input Types: (date)

Returns: (The day of the week as a number from 1 to 7, where 1 is Sunday, 2 is Monday, etc. An invalid or NULL_DATE, Returns 0.)

Example

These examples show how **DoW** extracts the number of the day of the week from a date:

```
Today()
```

Returns: 05/06/96

```
DoW( Today() )
```

Returns: 2

```
DoW( Today() + 3 )
```

Returns: 5

Related Functions

Also see the Day() function, or the Month() function, or the ValidDate(sDate,Flag) function, or the Year() function.

DtoC()

Description

Used to convert a date to a string. This function is a date conversion function used for formatting purposes when you want to display the date as a string. The date to be converted into a string is defined in the parameters.

Parameters

DtoC(Date)

Input Types: (date)

Returns: (A string representation of Date formatted in the current date format. A NULL_DATE returns a string of spaces equal in length to the current date format.)

Example

These examples show how **DtoC** converts a date into a string:

```
Today()
```

Returns: 05/06/96

```
DtoC(Today())
```

Returns: 05/06/96

```
"Today is " + DtoC(Today())
```

Returns: Today is 05/06/96

Related Functions

Also see the CtoD() function, or the DtoS() function

DtoS()

Description

Used to convert a date to a string formatted as YYYYMMDD. This function is a date conversion function used to convert a date value to a string that can be concatenated to any other string. The return value is structured to preserve date order (year, month, and day). The date to be converted into a string is defined in the parameters.

Parameters

DtoS(Date)

Input Types: (date)

Returns: (An 8-character string in the format YYYYMMDD. A NULL_DATE Returns a string of 8 spaces.)

Example

These examples show how the **DtoS** function converts a date into a string:

Today()

Returns: 03/22/2002

DtoS(Today())

Returns: 20020322

Related Functions

Also see the DtoC() function or the CtoD() function

Jday()

Description

Used to return the current Julian date.

Parameters

Jday(Date)

Input Types: (date)

Returns: (numeric)

Example

Today() returns 03/30/2001

Jday(Today()) returns 89.

Related Functions

LimsDate()

Description

Used to return a date as a *string* in the format specified in the parameters. The date to be returned and the format of the string are defined in the parameters.

IMPORTANT NOTICE:

- The Oracle driver version 7 is known to malfunction in date operations when the year part of the date contains only 2 digits (1-DEC-01). For this reason the recommendation is to use the Format parameter with the "ORACLE2000" value. This will return a data that has a 4 digit year and select statements where a range of dates is passed as a condition will return the correct results.

Parameters

LimsDate(Date, Format)

Examples of Format are:

"YY-MM-DD"

"DD-MMM-YY" (Oracle Default Format)
"DD-MM-YYYY"
"MM-DD-YY"

"ORACLE2000" (Oracle Format – DD-MMM-YYYY)

"MM-DD-YYYY"
"DD-MM-YY" (MSSQL Default Format)

"DD-MM-YYYY" (MSSQL Format)

If Format is not specified, the default Oracle date format is returned (i.e., 28-NOV-96).

Input Types: (date, string)

Returns: (Date as a string specified by Format)

Example

These examples return the date as a string in the specified format:

```
LimsDate(Today(),"YY-MM-DD")
```

Returns: 96-11-28

```
LimsDate(Today(),"YYYY-MM-DD")
```

Returns: 1996-11-28

```
LimsDate(Today(),"MM-DD-YY")
```

Returns: 11-28-96

```
LimsDate(Today(),"YYYY.MM.DD")
```

Returns: 1996.11.28

```
LimsDate(Today())
```

Returns: 96-NOV-28

Related Functions

Also see the DateFormat() function

LimsGetDateFormat()

Description

This function returns the default date format or the format that was set by the function DateFormat().

Parameters

LimsGetDateFormat()

Input Types: (none)

Returns: (string - the date format.)

Example

LimsGetDateFormat()

Returns: "MM/DD/YYYY"

Related Functions

Also see the DateFormat() function

LimsSecs()

Description

Obsolete. Use Seconds instread

Used in place of the Seconds() function in Windows NT systems. to return the number of seconds that have elapsed since midnight, in the form seconds. hundredths. Numbers range from 0 to 86,399. This function provides a simple method of calculating elapsed time during program execution, based on the system clock. It is related to the LimsTime() function, which returns the system time in the form of *hh:mm:ss*. The smallest resolution depends on the hardware timer tick.

Parameters

LimsSecs()

Input Types: (none)

Returns: (The number of seconds that have elapsed since midnight.)

Example

This example contrasts the value of LimsTime() with **LimsSecs()**:

LimsTime()

Returns: 10:00:00

LimsSecs()

Returns: 36000.00

Related Functions

Also see the Seconds() function, or the LimsTime() function

LimsTime()

Description

Used in place of the Time() function in Windows NT systems to return the system time in a format determined by the current time as a string. This function returns the system time. **LimsTime** is related to LimsSecs(), which returns the integer value representing the number of seconds since midnight for Windows NT systems.

Parameters

LimsTime()

Input Types: (none)

Returns: (string - system's time)

Example

If the current time is 4:30 PM and the settings are for a 12-hour time with "AM" and "PM" extensions, and colon separator set in Control Panel:

LimsTime()

Returns: 04:30:00 PM

Related Functions

Also see the Seconds() function, or the LimsSecs() function

Month()

Description

Used to extract the number of the month from a date. It is a date conversion function that is used when you need a numeric month value during calculations for such things as periodic reports. **Month** is a member of a group of functions that return components of a date value as numbers. The group includes the Day() function and the Year() function to return the day and year values as numeric. (See the Day() function and the Year() function) Cmonth() is a related function that allows you to return the name of the month from a date value. The date from which you want to extract the month from is defined in the parameters.

Parameters

Month(Date)

Input Types: (date)

Returns: (The number of the month, between 0 to 12. For an invalid or NULL_DATE, 0 is returned.)

Example

These examples return the month of the system date:

```
Today()
```

Returns: 05/15/96

```
Month(Today())
```

Returns: 5

Advance by 30 days:

```
Month(Today()) + 30
```

Returns: 6

This example uses month along with the Day() and Year() functions:

```
"We are", Year(Today()), "years, ",;Month(Today()), "months, and",
Day(Today()),;"days into our history!"
```

This example shows the result when a NULL_DATE is passed:

```
Month(NULL_DATE)
```

Returns: 0

Related Functions

Also see the DoW() function, or the Day() function, or the ValidDate(sDate,Flag) function or the Year() function

NoOfDays()

Description

Used to return the number of days in the Date's month.

Parameters

NoOfDays(Date)

Input Types: (date)

Returns: (numeric)

Example

This example finds the number of days in the month of February, 1997.

```
NoOfDays(CtoD("02/05/97"))
```

Returns: 28

Related Functions

Seconds()

Description

Used to return the number of seconds that have elapsed since midnight, in the form seconds. hundredths. Numbers range from 0 to 86,399. This function provides a simple method of calculating elapsed time during program execution, based on the system clock. It is related to the Time() function, which returns the system time in the form of *hh:mm:ss*. The smallest resolution depends on the hardware timer tick.

Parameters

Seconds()

Input Types: (none)

Returns: (The number of seconds that have elapsed since midnight.)

Example

This example contrasts the value of Time() with **Seconds** functions:

```
Time()
```

Returns: 10:00:00

```
Seconds()
```

Returns: 36000.00

Related Functions

Also see the LimsTime() function or the LimsSecs() function, or the Time() function

Time()

Description

Used to return the system time in a format determined by the current time as a string. This function returns the system time. **Time** is related to the Seconds() function, which returns the integer value representing the number of seconds since midnight.

Parameters

Time()

Input Types: (none)

Returns: (string - system's time)

Example

If the current time is 4:30 PM and the settings are for a 12-hour time with "AM" and "PM" extensions, and colon separator set in Control Panel:

```
Time()
```

Returns: 04:30:00 PM

Related Functions

Today()

Description

Used to return the system date as a date value. This function provides a means of initializing memory variables to the current date, comparing other date values to the current date, and performing date arithmetic relative to the current date. The default format is MM/*DD/YY*.

Parameters

Today()

Input Types: (none)

Returns: (system date)

Example

These examples show the **Today** function used in various ways:

```
Today()
```

Returns: 05/16/96

```
Today() + 30
```

Returns: 06/15/96

```
Today() – 30
```

Returns: 04/16/96

Related Functions

ValidateDate()

Description

Used to check to see if a date is valid.

Parameters

ValidateDate(sDate,Flag)

sDate a date

Flag True or False.

.T. will check for a 4-digit year

.F. will check for a 2-digit year

Input Types: (date, logic)

Returns: (logic)

Example

ValidateDate("01/01/98")

Returns a .T. (true)

ValidateDate("02/30/98")

Returns a .F. (false)

Related Functions

JWeek()

Description

Used to extract the number of the week from a date. It is a date conversion function that converts a date value to a numeric week value. The **JWeek** functions a member of a group of functions that return components of a date value as numbers. The group includes the Day() and the Month() functions to return the day and month values as numbers. (See the Day() function and the Year() function) The date from which you want to extract the week from is defined in the parameters.

Parameters

JWeek(Date)

Input Types: (date)

Returns: (The week of the year, as a 2-digit number.)

Example

These examples illustrate the **JWeek()** function using the system date:

Today()

Returns: 04/29/96

JWeek(Today())

Returns: 14

Related Functions

Also see the DoW() function or the Month() function or the Day() function or the Year() function.

Year()

Description

Used to extract the number of the year from a date. It is a date conversion function that converts a date value to a numeric year value. The **Year** function is a member of a group of functions that return components of a date value as numbers. The group includes the Day() function and the Month() function to return the day and month values as numbers. (See the Day() function or the Month() function) The date from which you want to extract the year from is defined in the parameters.

Parameters

Year(Date)

Input Types: (date)

Returns: (The year of Date, including the century digits, as a 4-digit number.)

Example

These examples illustrate the **Year()** function using the system date:

Today()

Returns: 04/29/96

Year(Today())

Returns: 1996

Year(Today()) + 11

Returns: 2007

Year(05.16.95)

Returns: 1995

Related Functions

Also see the DoW() function, or the Month() function, or the ValidDate(sDate,Flag) function, or the Day() function.

E-mail Functions

SendFromOutBox()

Description

Used to send all the unsent messages from a user's outbox.

Returns .T. or .F.

Input Types: (empty)

Returns: (logic)

Example

```
SendFromOutBox()
```

Related Functions

SendLimsEmail()

Description

Used to send a mail message.

Returns .T. - success

.F. - failed

In this version of email client mail is not sent to the BCC list.

This function also allows the use of SMTP servers that requires authentication with the AUTH protocol. Please check with your system administrator before using these parameters.

Parameters

SendLimsEmail(SmtpServerIP,RecipientsList,SenderEmail,Subject,MessageBody,AttchmntList,CCList,BCCList,ReplyToAddress, UserName, Password)

SmtpServerIp – string – the address of the SMTP server

RecipientsList – array - list of recipients

SenderEmail – string – email address of the sender

Subject – string – the subject of the email

MessageBody – string – the body of the email

AttchmntList – array – list of files attached

CCList – array – list of recipients in the CC list

BCCList – array – list of recipients in the BCC list

ReplyToAddress – string – reply address if different from the SenderEmail parameter

UserName – string – Username for a SMTP server that requires authentication with the AUTH protocol

Password – string – Password for a SMTP server that requires authentication with the AUTH protocol

Input Types: (string, array, string, string, string , array, array, array, string, string, string)

Returns: (logic)

Example

```
SendLimsEmail(“123.123.123.123”,{“mike@limsltd.com”,”george@limsltd.com”},”alin@limsltd.com”, ”New functions”, “Here is a list of new functions”, {“C:\Temp\func1.doc”,” C:\Temp\func2.doc”}, {“radu@limsltd.com”, “dan@limstd.com”},{},”alin@starlims.com”, “userAlin123”, “pass321alin”)
```

Related Functions

SendToOutBox()

Description

Used to send an email message to the OUTBOX table for later delivery either automatic with SendBackgroundMail() or manual with SendFromOutbox().

This function also allows the use of SMTP servers that requires authentication with the AUTH protocol. Please check with your system administrator before using these parameters.

The dictionary table OUTBOX should contain 2 more fields, USERNAME, PASSWORD , varchar fields of 50 to receive these two parameters.

Parameters

SendToOutBox(SmtpServerIP,RecipientsList,SenderEmail,Subject,MessageBody,AttchmntList,CCList,BCCList,ReplyToAddress, Username, Password)

SmtpServerIp – string – the address of the SMTP server

RecipientsList – array - list of recipients

SenderEmail – string – email address of the sender

Subject – string – the subject of the email

MessageBody – string – the body of the email

AttchmntList – array – list of files attached

CCList – array – list of recipients in the CC list

BCCList – array – list of recipients in the BCC list

ReplyToAddress – string – reply address if different from the SenderEmail parameter

UserName – string – Username for a SMTP server that requires authentication with the AUTH protocol

Password – string – Password for a SMTP server that requires authentication with the AUTH protocol

Input Types: (string, array, string, string, string , array, array, array, string, string, string)

Returns: (logic)

Example

```
SendToOutBox("123.123.123.123",{mike@limsltd.com,george@limsltd.com},"alin@limsltd.com", "New functions", "Here is a list of new functions", {"C:\Temp\func1.doc"," C:\Temp\func2.doc"}, {"radu@limsltd.com", "dan@limstd.com"},{},"alin@starlims.com")
```

Related Functions

File Manipulation Functions

DosSupport()

Description

Used to execute DOS commands, the DOS command and the name of the directory is defined in the parameters.

DosSupport(DOS Command, Parameter)

Returns: Logic / Array / String – depending upon the command used. (logic - .T. if command is successfully executed, or .F. if it's not successfully executed where an error message is displayed.)

DOS Commands available:

MD	Make Directory – Creates new directory
RD	Remove Directory – Removes directory
CD	Change Directory – Changes current directory
DIR	Returns the contents of a directory into an array.
WORK	Returns the current StarLIMS working directory as a string.
CURRENTDIR	Returns the current StarLIMS directory path.
CURRENTDRIVE	Returns the current StarLIMS drive letter.
ISDIR	Returns a logical .T. if the directory exists or a logical .F. if not.

Parameters

The Parameter is the name of the directory.

Input Types: (string, string)

Returns: (various -Logic / Array / String)

Example

To make a new directory called "New Tests":

```
DosSupport("MD", "New Tests")
```

Related Functions

Also see the FileSupport() function

FileSupport()

Description

Used to manipulate file without having to use the command feature

Parameters

FileSupport("Fname", "Req", "Dest", "RelPos");

where:

Fname is the file name for the source file.

Dest is the file mane of the destination file.

RelPos indicates a relative position inside the file; can have these values: "TOP", "BOTTOM", "RELATIVE".

Req can have one of the following values:

1) **PATH** – Returns the full path for Fname

2) **SIZE** – Returns the size for Fname

3) **DATE** – Returns the last date when Fname was changed.

4) **TIME** – Returns the last time when Fname was changed.

5) **NAME** – Returns the name of Fname, when Fname is in a full path format.

6) **EXT** – Returns the extension portion of the file name when Fname is in a full path format.

7) **SETATTR –** Sets the file attributes of Fname to the attributes specified in “Dest”: “R” for “Read Only”, “A” for “Archive”, “H” for “Hidden”, “S” for “System”, “N” for “Normal”. Setting the attributes to “N” disables all other settings. If you need to set multiple attributes, you have to use this function for each of them.

8) **GETATTR –** Returns a string containing the attributes of the file. Ex: “AHN” – archive, hidden and normal.

9) **COPY** – copies Fname to the Dest (simple name of full path name).

10) **DELETE** – deletes Fname

11) **RENAME** – renames Fname to Dest

12) **MOVE** – moves Fname to the Dest (simple name of full path name). The Move option will move (rename) either a file or a directory (including all its children) either in the same directory or across directories. The one caveat is that this option will fail on directory moves when the destination is on a different volume. Also, the destination file must not already exist.

Options 6 – 10 return either a .T. or .F.

13) **CHECK** – checks the existence of Fname and returns either .T. or .F.

14) **CREATE** – creates the file Fname. Returns a handle if it is successful, or else returns 0.

15) **OPEN** – opens the file Fname. Returns a handle if it is successful or else returns 0.

In the next options Fname is the handle obtained by option 11 or option 12.

16) **WRITE** – writes the string Dest to the current position of Fname. returns the number of bytes written.

17) **SEEK** – changes the current position of Fname to the integer Dest, according to the RelPos, which can have these values: "TOP", "BOTTOM", "RELATIVE". Returns the new offset from TOP.

18) **TELL** – Returns the current position of Fname, starting from TOP.

19) **READ** – reads the number of bytes specified by the integer Dest, starting from the current position of Fname. Returns what was read as a string.

20) **CLOSE** – closes the file specified by the handle Fname.

21) **READBLK** – Reads a block of data from designated file. (Must use provide the delimiting character(s) in the Rest parameter).

22) **EOF** – Returns .T. if end of file or .F. if not end of file.

23) **BOF** – Return .T. if beginning of file or .F. if not beginning of file.

Input Types: (string, string, string / numeric, string)

Returns: (various – String / Numeric / Logical / FileHandle)

Example

FileSupport("C:\MyText.Doc","RENAME","C:\NewText.Doc");

Returns .T. or .F. depending upon the completion of the task

Related Functions

Lcopy()

Note:

This function is obsolete, but is still supported. You should use the FileSupport() function.

Description

Used to copy an existing file to a new destination or device. The file name and destination are defined in the parameters.

Parameters

Lcopy(File Name, Destination)

Input Types: (string, string)

Returns: (empty string)

Example

To copy the file "Readme.txt" to Drive A:, the following **Lcopy()** parameters are used:

Lcopy("Readme.txt","A:\Readme.txt")

Related Functions

Also see the FileSupport() function.

Ldelete()

Note:

This function is obsolete, but is still supported. You should use the FileSupport() function.

Description

Used to delete the file name specified in the parameters. The full path is supported.

Parameters

Ldelete(File Name)

Input Types: (string)

Returns: (empty string)

Example

To delete the file, "Readme.txt", the following **Ldelete()** parameters are used:

```
Ldelete("Readme.txt");
```

Or

```
Ldelete("C:\Readme.txt")
```

Related Functions

Also see the FileSupport() function.

Ldir()

Description

Used to find and return an array of all files matching the file specification. It can also be used to check for the existence of a specific file.

Parameters

Ldir(File Specification)

Input Types: (string)

Returns: (array)

Example

To return an array of all existing Word DOC files in the WORDDOC directory, the following is used:

```
Ldir("C:\WORDDOC\*.DOC")
```

The next example searches for the existence of a specific file:

```
Ldir("C:\WORDDOC\BTEX.DOC")
```

Related Functions

Also see the FileSupport() function.

Lrename()

Note:

This function is obsolete, but is still supported. You should use the FileSupport() function.

Description

Used to rename an existing file.

Parameters

Lrename(Old File Name, New File Name)

Input Types: (string, string)

Returns: (empty string)

Example

To rename the 'Readme.txt' file to 'Readme.bak', the following Lrename() parameters are used:

```
Lrename("Readme.txt", "Readme.bak")
```

Related Functions

Also see the FileSupport() function.

ReadText()

Description

Used to read a text file and return this text as a string. Plain ASCII text is recommended if the text is to be displayed, though any type of file can be used. The file name is defined in the parameters. This function is also related to the FileSupport() function.

Parameters

ReadText(File Name)

Input Types: (string)

Returns: (string)

Example

To display the text of file "Readme.txt" in a user message, the following parameters are used:

```
UsrMes("Good Morning", ReadText("Readme.txt"))
```

The UsrMes()**)** function formats and wraps the text string as needed

Related Functions

Also see the FileSupport() function.

WriteText()

Description

Used to create a text file which can be either overwritten or appended to with additional WriteText statements. This function is also associated with the ReadText Function

Parameters

WriteText(File Name, Text, Confirm Required *("Y" or "N")*, Append *("Y" or "N")*)

where:

If Confirm Required is "Y", a confirmation box is displayed if the file name already exists. Default is "N". (optional)

If Append is "Y", the new text is automatically appended to an existing file. If Append is "N", the new text will automatically overwrite an existing file. Default is "N". (optional)

Input Types: (string, string, string, string)

Returns: (string)

Example

```
WriteText("Gehtest.txt", "This is the first line", "Y", "N");

WriteText("Gehtest.txt", "This is the second line",, "Y")
```

If the file Gehtest.txt exists, the first line allows the user to confirm overwriting this file. If overwriting is not confirmed, the new text will not be appended.

The second line automatically appends the new text to the file.

Related Functions

Also see the ReadText function

FTP Functions

CheckOnFTP()

Description

Used to check the existence of a file on the FTP server. The return values are:

T if the file exists

F if the file does not exist, a connection couldn't be made or the specified path does not exist

Parameters

CheckOnFTP(Ip,fDir,rfName,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The remote directory where the file exists.
rfName	The file name that you are checking to see if it exists.
usr	(optional) The remote user name.
Pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, numeric, string)

Returns: (logical)

Example

```
CheckOnFTP("FTP.StarLims.com","public","myfile.txt")
```

Related Functions

CopyToFTP()

Description

Used to create several files on remote FTP server and then filling them with the contents of a string. The return values are:

T if the copy was successful.

F if the copy was not successful.

Parameters

CopyToFTP(Ip,fDir,aNames,cStr,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The remote directory where the file exists to be copied.
aNames	The array of the file that will be created.
cStr	The string used to fill this file.
usr	(optional) The remote user name.
pass	optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, array, string, string, numeric, string)

Returns: (logical)

Example

CopyToFTP("FTP.StarLims.com”,"public",{"first.txt","second.txt"},"stringvalue")

In this example, 2 files are created, first.txt and second.txt on the remote FTP server on the public directory. The file (which one?) is filled with the contents of the string.

Related Functions

DeleteDirOnFTP()

Description

Used to delete directory from the remote FTP server. The return values are:

T if the delete was successful.

F if the delete was not successful.

Parameters

DeleteDirOnFTP(Ip,fDir,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The remote directory to be removed.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, numeric, string)

Returns: (logical)

Example

```
DeleteDirOnFTP("FTP.StarLims.com”,"public")
```

In this example, FTP.StarLims.com is deleted. (the directory, not a file ?)

Related Functions

DeleteFromFTP()

Description

Used to delete a file from the remote FTP server. The return values are:

T if the delete was successful.

F if the delete was not successful.

Parameters

DeleteFromFTP(Ip,fDir,rfName,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The directory where the remote file to be removed exists.
rfName	The name of the remote file to be removed.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, numeric, string)

Returns: (logical)

Example

```
DeleteFromFTP("FTP.StarLims.com”,"public","myfile.txt")
```

In this example, myfile.txt is deleted.

If a proxy server is used:

```
DeleteFromFTP("FTP.StarLims.com”,"public","myfile.txt", , , ,
”111.111.111.111” )
```

Related Functions

GetDirFromFTP()

Description

Used to return an array of file names matching the requested criteria.

Parameters

GetDirFromFTP(Ip,fDir,rfName,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The directory where you are looking for the contents to return. Default: root directory
rfName	The name of the file to be received. The file name can contain wild cards (* or ?).
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, numeric, string)

Returns: (array)

Example

GetDirFromFTP("FTP.StarLims.com”,"public","*.txt")

In this example, you will receive all files with .txt as an extension in the public directory.

Related Functions

GetFromFTP()

Description

Used to transfer a remote file from the FTP server to the local server. The return values are:

T if the transfer was successful.

F if the transfer was not successful.

Parameters

GetFromFTP(Ip,fDir,rfName,IFName,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The remote directory that contains the file to be transferred.
rfName	The name of the file to be transferred.
IFName	The name of the new file on the local server. This name should be fully qualified, that is, the entire path name. If not, it will be placed on the default directory.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy address.	(optional) The name of the local proxy server if it exists OR the IP

Input Types: (string, string, string, string, string, string, numeric, string)

Returns: (logical)

Example

```
GetFromFTP("FTP.StarLims.com","public","myfile.txt","C:\Windows\Temp\newfile.txt")
```

In this example, you will receive myfile.txt and put it in the C:\Windows\Temp directory with the file name newfile.txt.

Related Functions

MakeDirOnFTP()

Description

Used to create a directory on the remote FTP server.

Parameters

MakeDirOnFTP(Ip,fDir,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The name of the new remote directory.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, numeric, string)

Returns: (logical)

Example

MakeDirOnFTP("FTP.StarLims.com”,"NewDirectory")

In this example, you will create a directory called New Directory.

Related Functions

MoveInFTP()

Description

Used to move a remote file to another directory on the FTP server. You have the option to also change the file name.

Parameters

MoveInFTP(Ip,fDirFrom,fDirTo,rfNameFrom,rfNameTO,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDirFrom	The name of the remote directory that contains the remote file to be moved.
fDirTo	The name of the remote directory to which the remote file will be moved.
rfNameFrom	The name of the file to be moved.
rfNameTo	(optional?) The new name of the moved file. If the name does not need to be changed, then ?
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, string, string, numeric, string)

Returns: (logical)

Example

```
MoveInFTP("FTP.StarLims.com”,"public","mydirectory","myfile.txt","newfile.txt")
```

In this example, you will move myfile.txt and from the public directory to the mydirectory directory and rename it newfile.txt.

Related Functions

PollFTP()

Description

Used to stop local processing. It will wait for the existence or non-existence of a remote file on the FTP server. The return values are:

0 if the condition is met

1 if the condition is not met

2 if there is an error

Parameters

PollFTP(Ip,fDir,rfName,Exist,Timeout,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The name of the remote directory that contains the remote file.
rfName	The name of the remote file to check.
Exist	True or False – to wait if the file exists or not.
Timeout	The maximum number of seconds to wait until the condition is met.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, logic, numeric, string, string, numeric, string)

Returns: (numeric)

Example

```
PollFTP( "FTP.StarLims.com", "public", "myfile.txt", .T. ,60 )
```

Related Functions

ReadFromFTP()

Description

Used to read the contents of a remote file on the FTP server and returns the value as a string.

Parameters

ReadFromFTP(Ip,fDir,rfName,MaxSize,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The name of the remote directory that contains the remote file.
rfName	The name of the remote file to read.
MaxSize	The maximum size (bits) of the data. If the maximum size > file size, then the entire file is read. Default is 64,000 bits.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, numeric, string, string, numeric, string)

Returns: (string)

Example

```
X:=ReadFromFTP("FTP.StarLims.com”,"public","myfile.txt",120000)
```

In this example, you will read the first 120,000 bits of the contents of the file myfile.txt.

Related Functions

RenameOnFTP()

Description

Used to rename a file on the FTP server.

Parameters

RenameOnFTP(Ip,fDir,rfName,IFName,usr,pass,nPort,cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The name of the remote directory that contains the remote file.
rfName	The name of the remote file to rename.
IFName	The new name of the remote file.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, string, numeric, string)

Returns: (logical)

Example

```
RenameOnFTP("FTP.StarLims.com","public","myfile.txt","newfile.txt")
```

In this example, you will rename the file myfile.txt on the public directory to newfile.txt.

Related Functions

SendToFTP()

Description

Used to send a file from the local server to a remote FTP server.

Parameters

SendToFTP(lp,rfDir,rfName,lFName,usr,pass,nPort,cProxy)

where:

lp	The address or DNS name of the FTP server.
fDir	The name of the FTP destination directory .
rfName	The name of the remote file.
lFName	The name of the file on the local machine. This name should be fully qualified, that is, the entire path name.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, string, numeric, string)

Returns: (logical)

Example

SendToFTP("FTP.StarLims.com","public","myfile.txt","C:\Windows\Temp\newfile.txt")

In this example, you will send myfile.txt from the local computer and put it in the Public directory on the remote FTP server with the file name myfile.txt.

Related Functions

WriteToFTP()

Description

Used to create a file on the remote FTP server and fill it with a string.

Parameters

WriteToFTP(Ip, fDir, rfName, cStr ,usr, pass, nPort, cProxy)

where:

Ip	The address or DNS name of the FTP server.
fDir	The name of the directory that contains the file.
rfName	The name of file
cStr	The string to be written into the file.
usr	(optional) The remote user name.
pass	(optional) The remote user password.
nPort	(optional) The connection port to the FTP server. Default = 21
cProxy	(optional) The name of the local proxy server if it exists OR the IP address.

Input Types: (string, string, string, string, string, string, numeric, string)

Returns: (logical)

Example

```
:DECLARE aResult, cStr;
aResult := SQLExecute("Select ACTION from ACTIONS where ACTIONID
= 'ACT-111'");
cStr := aResult[1,1];
WriteToFTP("FTP.StarLims.com","public","document.txt", cStr);
```

Related Functions

Logical Functions

Calculate()

Description

Used to perform a calculation stored in another table and field.

The function:

Retrieves a specific calculation (RETExpr) identified by the select expression (SELECT Expression).

Performs the calculation formula and returns the result.

If the "picture" is defined, the result is returned according to this format.

If the calculation formula cannot be found, the "default" value is returned (if defined).

The return expression (**RETExpr**), select expression (**SELECT Expression**), **Picture**, and **Default** value are defined in the parameters.

Parameters

Calculate(SELECT Expression, Return Expression, Picture, Default)

Input Types: (string, string, string, any type)

Returns: (any type)

Example

The following example shows how the **Calculate** function is used to perform a calculation stored in another table.

```
:IF <<ORIGSTS>>'='A' ;
      ExecAction('AUDTR', 'AUDCOM');
:ENDIF;

:IF Empty(LOW_A);
      LOW_A := "-9999999"
:ELSE;
      LOW_A := Val(LOW_A);
:ENDIF;

:IF Empty(LOW_B);
      LOW_B := "-9999999"
:ELSE;
      LOW_B := Val(LOW_B);
:ENDIF

:IF Empty(HIGH_A);
      HIGH_A := "-9999999"
:ELSE;
      HIGH_A := Val(HIGH_A);
:ENDIF

:IF Empty(HIGH_B);
      HIGH_B := "-9999999"
:ELSE;
      HIGH_B := Val(HIGH_B);
:ENDIF

RES3 := Val(RES);

CURRENT_NUMRES := Val(RES);
```

```
VC := Calculate("Select ORIGREC,VCOND from ANALYTES where
TESTCODE =" +TMP_TESTCODE+" and ANALYTE =
'"+TMP_ANALYTE+"' ","VCOND",, "If(RES3 = 0, STAT, If(RES3 <
LOW_A, 'OOS-A', If(RES3 > HIGH_A, 'OOS-A', If(RES3 < LOW_B, 'OOS-
B', If(RES3 > HIGH_B, 'OOS-B', 'Done' )))))"));
```

In the above expression, stored in the RESULTS.S calculation field, **Calculate** is used to find the Validation Condition VCOND stored in the ANALYTES table for the current <<TESTCODE>> and <<ANALYTE>>.

If such an expression exists, Calculate will evaluate this expression. However, if an expression doe not exist an alternative expression is evaluated. In this case:

```
If(Val(CURRENT_NUMRES)=0,'Done', If(Val(CURRENT_NUMRES) <
Val(LOW_A) .OR. Val(CURRENT_NUMRES) > Val(HIGH_A), 'OOS-A',
If(Val(CURRENT_NUMRES) < Val(LOW_B) .OR.
Val(CURRENT_NUMRES) > Val(HIGH_B), 'OOS-B',' Done')))")
```

which assigns the S field a value of Done if the result is within limits, assigns it OOS-A if the results are out of LOWA-HIGHA, and assigns it OOS-B if the results are out of LOWB-HIGH-B.

Another example using **Calculate**:

```
Calculate("Select CALCUL from ANALYTES where TESTCODE =
<<TESTCODE>> AND ANALYTE = '"<<ANALYTE>>' ", "CALCUL",
"<<PICTURE>>", '"<<CHRRES>>"')
```

This expression is maintained in the RESULTS.NUMRES Formula field. In this case, the Calculate function performs the calculation in the ANALYTES.CALCUL field for the current <<TESTCODE>> and <<ANALYTE>>.

If a calculation is not found then <<CHRRES>> is returned.

The PICTURE parameter converts the result to the picture specified in the <<PICTURE>> field. (SCI can be used to return scientific notation).

Related Functions

ChkPrm()- Obsolete

Note:

This function is obsolete. Replaced by PrmCount()

Description

Used to check if every element defined in the array contains valid data, which is data that is not null. If the elements contain valid data, TRUE is returned. If one of the elements is NULL, then FALSE is returned and the calculation is not performed.

Parameters

ChkPrm(Array)

Input Types: (array)

Returns: (Logic - True or False)

Example

The following example checks if RN(1) and RN(2) contain values before running the calculation RN(1) / RN(2):

```
ChkPrm({'RN1','RN2'})
```

Related Functions

If()- Obsolete

Note:

This function is obsolete. Replaced by the :IF **and** :ENDIF **statements.**

Description

Used to return the result of an expression based on a condition. The **If** function evaluates a condition within an expression. Using this function, you can convert a logical expression to another data type.

Parameters

If(Logical Condition, True Value, False Value)

where:

Logical Condition = Logical expression to be evaluated

True Value = Value returned if condition is True

False Value = Value returned if condition is False

Input Types: (any type, any type, any type)

Returns: (any type, either True Value or False Value, depending on the value of the Logical Condition.)

Example

The following examples show how If is used:

```
NUMINDAT := Lselect("Select Max(NUMINDAT) from CALCEXER where
ADDEDDATE = Today()");

If(Empty(NUMINDAT[1,1]),1,NUMINDAT[1,1] +1)
```

If the first row and column of the variable NUMINDAT is empty, the condition is TRUE and the value of 1 is returned. If NUMINDAT is not empty then the condition is FALSE and the value returned is the variable + 1.

Related Functions

Lcase()- Obsolete

Note:

This function is obsolete. Replaced by the :BEGINCASE**,** :CASE **and** :ENDCASE **statements.**

Description

This function is similar to the XBase If function, which is used to return the result of an expression based on a condition. However, when using **Lcase** the Else parameter is not mandatory. (See the If()- Obsolete function.) The logical condition, the value if the condition is true and the value if it is false are defined in the parameters.

Parameters

Lcase(Logical Condition, Expression 1,Expression 2)

where:

Logical Condition = Logical expression to be evaluated

Expression 1 = Expression executed or value returned if condition is True

Expression 2 = (optional) Expression executed or value returned if condition is False.

Input Types: (string - logical expression, [any type],[any type])

Returns: (result of Expression 1 or Expression 2 depending on Logical Condition)

Example

Lcase('<<ORIGSTS>>='A',"ExecAction('AUDTR', 'AUDCOM')")

Here, Lcase checks to see if the current ORIGSTS is equal to A, meaning that the audit trailing is necessary. If Yes then the ExecAction function executes the action stored in the AUDTR window and that has the code AUDCOM. Note that the function executed for the true condition is embedded in double quotes.

Related Functions

Also see the :BEGINCASE statement, or the :CASE statement, or the :EXITCASE statement, or the :OTHERWISE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ENDIF statement, or the :ELSE statement.

Nothing()

Description

This function returns a logical true or false, depending on the value of the field or variable.

Parameters

Nothing(<<field name>>) or **Nothing(Variable Name)**

Input Types: (string)

Returns: (Logic - True or False)

Example

```
Nothing(<<MATNO>>)
```

Returns .T. if MATNO is empty or 0. Otherwise returns .F.

Related Functions

Miscellaneous Functions

LimsAPICall()

Description

This function is used to call functions from external Dynamic Link Lybraries.

Because the functions may require specific data types for the parameters, the function LimsCast() handles the conversions. These new functions are intended for the advanced Windows programmer who may want to call a Windows API function in StarLIMS. Before using these functions, the user should contact us directly for assistance.

Parameters

Example

Related Functions

LimsCast()

Description

This function is used in conjunction with LimsAPICall() to alter the type of variables passed to functions from external Dynamic Link Lybraries.

These new functions are intended for the advanced Windows programmer who may want to call a Windows API function in StarLIMS. Before using these functions, the user should contact us directly for assistance.

Parameters

Example

Related Functions

LimsElapsed()

Description

Used to find the difference between two times. The times are entered as string values and returns a string as the result.

Parameters

LimsElapsed(Start Time, End Time)

Input Types: (string, string)

Returns: (string)

Example

To return the difference between two times or time variables the following can be used:

LimsElapsed("05:00:00","07:30:00") returns “02:30:00”

Or

LimsElapsed("23:00:00","02:00:00") returns “03:00:00”

Related Functions

LimsGetAllIPAddr()

Description

Used to find all the IP addresses assigned to all the network interfaces of the local computer.

Parameters

LimsGetAllIPAddr()

Input Types: (none)

Returns: (array)

Example

If the local computer has two network interfaces the function

LimsGetAllIPAddr()

will return: {“192.168.30.1”,” 234.122.5.1”}

Related Functions

LimsGetHostName()

Description

Used return the name of the local computer.

Parameters

LimsGetHostNamer()

Input Types: (none)

Returns: (string)

Example

If the name of the local computer is “PRODUCTION21” the function

LimsGetHostName()

will return the string: “PRODUCTION21”

Related Functions

LimsGetIPAddress()

Description

Used to find the IP addresses of the indicated computer on the network.

Parameters

LimsGetIPAddress(HostName)

Input Types: (string)

Returns: (string)

Example

If we want to find out the IP address of the computer named "PRODUCTION21" from the local network,

LimsGetIPAddress("PRODUCTION21")

will return the string: "192.168.30.1"

Related Functions

StationName()

Description

Used to return the computer name stored in the system registry.

Parameters

StationName()

Input Types: (empty)

Returns: (string)

Example

StationName() returns "LABSTATION1" if this function is run on a computer called "LABSTATION1".

Related Functions

UseClipper()

Description

Used by only non USA customers to allow the American setting to be returned for dates, numbers, etc. if set to .T. or if set to .F. the system will return the native settings.

Parameters

UseClipper(Flag)

where Flag

If set to **.T.** will return the American settings for Dates, Numbers, etc.

If set to **.F.** will return the native settings for Dates, Numbers, etc.

Input Types: (logic)

Returns: (empty string)

Example

```
UseClipper(.T.);
```

Related Functions

UserName()

Description

Used to return the windows current user name. This will be the name of the current user that logged onto the workstation.

Parameters

UserName()

Input Types: (empty string)

Returns: (string)

Example

UserName() returns “MANAGER” if the user that is logged in on the current station has the username “MANAGER”.

Related Functions

WinOSInfo()

Description

Used to return information about the Operating System

Parameters

WinOSInfo()

Input Types: (empty string)

Returns: (string)

Example

This example will return "Microsoft Windows XP Professional (Build 2600)" on a computer with Windows XP installed.

```
:DECLARE sOSInfo;

sOSInfo := WinOSInfo();
UsrMes("OS information",sOSInfo);

:RETURN;
```

Related Functions

Numeric Functions

Integer()

Description

Used to truncate a number with decimal digits to a whole number. **Integer** converts a numeric value to an integer by dropping the decimal point and truncating--not rounding--all digits to the right of the decimal point. This function is useful in operations where the decimal portion of a number is not needed. However, a value of 0.99999999 will be converted to 0.

Note:

Integer is the same as the Int function in *CA-Clipper*, but INT is a reserved word in CA-Visual Objects. Under CA-Visual Objects, Int is a conversion operator. With numbers that are not greater than a short integer, it yields the same results as Integer and maintains compatibility with *CA-Clipper*.

Parameters

The number to truncate (Decimal Value) is defined in the parameters.

Integer(Decimal Value)

Input Types: (numeric)

Returns: (The whole number to the left of the decimal point.)

Example

The following examples show how Integer is used:

Integer(100.00)

Returns: 100

Integer(.5)

Returns: 0

Integer(-100.75)

Returns: -100

Related Functions

MatFunc()

Description

Used to execute a specified numeric function with a value indicated in the parameters to return a result.

The numeric functions include:

ABS	Returns the absolute value of a numeric expression, regardless of its sign.
ACOT	Calculates the arc cotangent of a number.
ATAN	Calculates the arc tangent of a number.
COS	Calculates the cosine of a number.
COT	Calculates the cotangent of a value.
EXP	Calculates the numeric value of a natural logarithm.
FACT	Calculates the factorial of a number.
FRAC	Returns the fractional portion of a number.
LOG	Calculates the natural logarithm of a numeric value.
LOG10	Calculates the common logarithm of a numeric value.
PI	Returns the value of Pi
RAND	Calculates a random number
SIN	Calculates the sine of a number.
SQRT	Returns the square root of a positive number.
TAN	Calculates the tangent of a number.

For the FACT option (calculates the factorial of a number) the maximum value admitted is 12.

For the SIN, COS, TAN the result will be returned in radians.

Parameters

MatFunc(Function Name, Value)

where:

Function Name is one of the numeric functions listed in the Description above.

Input Types: (string, numeric)

Returns: (the numeric result)

Example

Examples **MatFunc** are shown below:

```
MatFunc("ABS",-12)
```

Returns: 12

```
MatFunc("ACOT",20)
```

Returns: 0.05 radians

```
MatFunc("COS",0)
```

Returns: 1.00

```
MatFunc("COT",1.5)
```

Returns: 0.07

MatFunc("EXP",1)

Returns: 2.72

MatFunc("FACT",5)

Returns: 120

MatFunc("LOG",10)

Returns: 2.30

MatFunc("LOG10",10)

Returns: 1.00

MatFunc("SIN",2.5)

Returns: 0.60

MatFunc("SQRT",4)

Returns: 2.00000

MatFunc("TAN",1.38)

Returns: 5.18

Related Functions

Max()

Description

Used to return the larger of 2 values. The inverse of **Max** is Min(), which returns the smaller of 2 expressions. The first and second values to compare are defined in the parameters. The second value must be of the same type as the first value, except that numeric of different types are allowed. The value returned is the same type as the larger value.

Parameters

Max(Value 1,Value 2)

Input Types: (any type, same type as Value 1)

Returns: (the largest value)

Example

In these examples **Max Returns** the larger of 2 numbers:

Max(99, 100)

Returns: 100

Max(100, 99)

Returns: 100

In these examples **Max** compares date values:

Today()

Returns: 06/03/96

Max(Today(), Today() + 1)

Returns: 06/04/96

Related Functions

Also see the Min() function.

Min()

Description

Used to return the smaller of 2 values. The inverse of **Min** is Max(), which returns the larger of 2 expressions. The first and second values to compare are defined in the parameters. The second value must be of the same type as the first value, except that numeric of different types are allowed. The value returned is the same type as the smaller value.

Parameters

Min(Value 1,Value 2)

Input Types: (any type, same type as Value 1)

Returns: (the smaller value)

Example

In these examples **Min** returns the smaller of 2 numbers:

Min(99, 100)

Returns: 99

Min(100, 99)

Returns: 99

In these examples **Min** compares date values:

Today()

Returns: 06/03/96

Min(Today(), Today() + 30)

Returns: 06/03/96

Related Functions

Also see the Max() function.

Rand()

Description

Used to return a random number between 0 and 1. **Rand** allows you to generate pseudo-random numbers. Multiple calls to **Rand** always return the same random number sequence, provided that they have the same start value (Seed) on the first call and that any subsequent calls do not specify the Seed.

Parameters

Rand(Seed)

where:

Seed = an optional start value. This is the point at which the random number generator is initialized. Subsequent random numbers are then influenced by the Seed.

If you first call **Rand** without Seed, it starts as though 100001 were specified. If you call the function with Seed as 100001, it allows you to restart the generator. Then, if you call the function several times without Seed, it returns the "standard sequence" of numbers. If Seed is less than or equal to 0, the system time is brought into the process.

Input Types: **(numeric)**

Returns: (numeric - between 0 and 1)

Example

Rand(1);

Rand() – returns for each run a random value between 0 and 1

First time: 0.748062729
Second time: 0.818540413
Third time: 0.084792103
… and so on.

Related Functions

Scient()

Description

Used to convert numeric values to scientific notation.

Parameters

Scient(numeric)

Input Types: (numeric)

Returns: (string)

Example

Scient(123.45)

Returns:1.2345E2

Related Functions

SetDecimal()

Description

Return and optionally change the setting that determines the number of decimal places used to display numbers.

SetDecimal() determines the number of decimal places displayed in the results of numeric functions and calculations. Its operation depends directly on the SetFixed() setting:

If SetFixed() is FALSE, SetDecimal() establishes the minimum number of decimal digits displayed by Exp(), Log(), SqRt(), and division operations. If SetFixed() is FALSE, SetDecimal() is still in effect.

If SetFixed() is TRUE, all numeric values are displayed with exactly the number of decimal places specified by SetDecimal().

Note that neither SetDecimal() nor SetFixed() affects the actual numeric precision of calculations — only the display format is affected.

Parameters

SetDecimal(NewSetting)

Input Types: (logical)

Returns: (logical)

NewSetting The number of decimal places to display. The initial default is 2. A negative value specifies that all significant digits are returned.

If NewSetting is not specified, SetDecimal() returns the current setting. If NewSetting is specified, the previous setting is returned.

Example

These examples show various results of the SetDecimal() function:

SetDecimal(2)

2/4 returns 0.50

1/3 returns 0.33

SetDecimal(4)

2/4 returns 0.5000

1/3 returns 0.3333

Related Functions

SetFixed()

Description

Return and optionally change the setting that fixes the number of decimal digits used to display numbers.

Parameters

SetFixed(NewSetting)

Input Types: (logical)

Returns: (logical)

NewSetting TRUE displays numeric output using the current SetDecimal() setting. FALSE ignores SetDecimal(), allowing the operation or function to determine the number of decimal places to display, according to the default rules for numeric display.

Example

SetFixed(.T.)

Returns: .F. if the previous setting was .F. or returns .T. if the previous setting was .T.

This is helpful when you want to perform some calculations and then return to the previous setting.

```
:DECLARE bSetting;
bSetting := SetFixed(.T.);
…. Some action that performs calculations…
SetFixed(bSetting);
```

Related Functions

SigFig() - Obsolete

Note:

This function has been superceded by the StdRound() Function

Description

Used to return a numeric value as a string after a rounding standard has been applied.

Parameters

SigFig(Rounding Standard, Number of Digits, Number)

Input Types: (string, numeric, numeric)

where:

Rounding Standard (String) is one of the following: ISO, EPA, FDA

Number of Digits (numeric) is a number

Number (numeric) is a number or variable containing a numeric variable.

Returns: (string)

Example

cAns := SigFig("FDA",5,nVal1);

Returns the value of nVal1 as a string with the rounding rule for FDA applied.

Related Functions

Also see the StdRound() function.

StdRound()

Description

Used to return a numeric value as a string after a rounding standard has been applied.

See the rounding rules in Appendix A: Rounding Rules

Parameters

StdRound(Rounding Standard, Number of Digits, Number)

Input Types: (string, numeric, numeric)

where:

Rounding Standard String is one of the following: ISO, EPA, FDA

Number of Digits numeric is a number

Number numeric is a number or variable containing a numeric variable.

Returns: (string)

EPA rounding should not be used with 0 (zero) as the number of digits because no significant values should be displayed.

When FDA is used and SetFixed is set to .T. (true) (which is not unlikely because during the calculations, all numeric values should be available) the rounding is adding 0000000(zero's) to the value.

Before the rounding the SetFixed should be set to .F. (false).

SetFixed(.F.);

Example

```
cAns := StdRound("FDA",5,nVal1);
```

Returns the value of nVal1 as a string with the rounding rule for FDA applied.

Related Functions

Val()

Description

Used to convert a string containing a numeric value to a numeric data type. If the string to convert is a valid numeric expression, **Val** processes it all. However, if the string is not entirely a valid numeric expression but contains decimal numeric, **Val** evaluates it until it reaches a second decimal point, the first non-numeric character, or the end of the expression. Leading spaces are ignored.

The Str() function and the Ltransform() function are closely related to **Val** since these functions convert numeric values to strings. The string to convert is defined in the parameters.

Parameters

Val(String)

where:

String = The string to convert. It can be in any of the compiler-supported base formats, such as binary, decimal, hex, or scientific.

Input Types: (string)

Returns: (numeric value)

Example

The following example demonstrates the use of Val:

```
RES := Lselect("SELECT NUMRES FROM RESULTS WHERE ANALYTE
= '<<ANALYTE>>' AND ORDNO = '<<ORDNO>>'");

Val(RES[1,1]) * 5
```

The variable RES contains the first row and column of the selected array from the NUMRES field for the current Analyte and order number. However, the NUMRES field is a character type and the values in this field, stored in the variable RES, are also character types. Therefore, there is a need to convert these character values to numeric values, using the Val function, before multiplying the variable by 5.

Related Functions

Also see the AllTrim() function, or the Left() function, or the Str() function, or the SubStr() function, or the Right() function.

ValidateNumeric()

Description

Used to check to see if a string is a valid number.

Parameters

ValidateNumeric(sNumber)

Input Types: (string)

Returns: (logical)

Example

ValidateNumeric("123")

Returns a .T. (true)

ValidateNumeric(".45")

Returns a .T. (true)

ValidateNumeric("123..45")

Returns an .F. (false)

Related Functions

LimsSetDigit()

Description

LimsSetDigit() and LimsSetDigitFixed() can be used together to control the way digits are displayed:

When LimsSetDigitFixed() is TRUE, display of numeric output is fixed according to the LimsSetDigit() value.

When LimsSetDigitFixed() is FALSE, numeric output displays according to the default rules for numeric display.
To provide finer control of numeric display, you can use the LTransform() function.
A -1 for <nNewSetting> implies that only significant whole digits to the left of the decimal are to be displayed (any leading zeros will be suppressed.)

Notes

LimsSetDigit() affects only the display format of numbers, not the actual numeric precision of calculations.

Purpose

Return and optionally change the setting that determines the number of digits that will be shown to the left of the decimal point when a number is displayed.

Syntax

LimsSetDigit([<nNewSetting>]) ---> dwCurrentSetting

Arguments

<nNewSetting> The number of digits to show. The initial default is 10. A negative value indicates that only the significant whole digits to the left of the decimal point (the mantissa) are returned. Any right padding is also suppressed.

Returns

If <nNewSetting> is not specified, LimsSetDigit() returns the current setting.
If <nNewSetting> is specified, the previous setting is returned.

Parameters

Syntax

LimsSetDigit([<nNewSetting>]) ---> dwCurrentSetting

Example

Examples

This example shows typical uses of LimsSetDigit():

Example 1:

```
:DECLARE nSaveDigit;

/* Initial setting is 10 digits;
/* Number is displayed right justified;
/* Save current setting, then reset to 5 digits;
```

```
nSaveDigit := LimsSetDigit(5);

Usrmes(,1234);   /*  "234";
Usrmes(,1234.567);   /*  "1234.567";
Usrmes(,123456);     /* "******";
LimsSetDigit(nSaveDigit);   /* Restore the old setting;

Usrmes(,1234);  /*  1234;

:RETURN ;
```

Example 2:

```
:DECLARE old_decSep, old_Dec, old_Digit;

old_Digit := LimsSetDigit(20);
old_Dec := SetDecimal(20);
old_decSep := SetDecimalSep(44);

UsrMes("", 999456/43);
UsrMes("", 9999456.566443*2342349993.887);
SetDecimalSep(old_decSep );
SetDecimal(old_Dec);
LimsSetDigit(old_Digit);

:RETURN;
```

Related Functions

LimsSetDigitFixed()

Description

Purpose

Return and optionally change the setting that fixes the number of digits used to display numeric output.

Description

LimsSetDigit() and LimsSetDigitFixed() can be used together to control the way digits are displayed:

When LimsSetDigitFixed() is TRUE, display of numeric output is fixed according to the LimsSetDigit() value.

When LimsSetDigitFixed() is FALSE, numeric output displays according to the default rules for numeric display.

Notes

LimsSetDigitFixed() affects only the display format of numbers, not the actual numeric precision of calculations.

Parameters

Syntax

LimsSetDigitFixed([<lNewSetting>]) ---> lCurrentSetting

Arguments

<lNewSetting> TRUE fixes the number of digits displayed. FALSE leaves the number of digits displayed unfixed. The initial default is FALSE.

Returns

If <lNewSetting> is not specified, LimsSetDigitFixed() returns the current setting. If <lNewSetting> is specified, the previous setting is returned.

Example

Examples

This example uses LimsSetDigitFixed() to start fixing the number of digits displayed at the beginning of a routine and stop fixing them at the end of the routine:

Example 1:

```
:DECLARE bSDFSetting;

bSDFSetting := LimsSetDigitFixed(TRUE);

/**** calls to other functions  ;

LimsSetDigitFixed(bSDFSetting);

:RETURN;
```

Example 2:

```
:DECLARE old_decSep, old_Dec, old_Digit;

old_Digit := LimsSetDigit(20);
old_Dec := SetDecimal(20);
old_decSep := SetDecimalSep(44);

UsrMes("", 999456/43);
UsrMes("", 9999456.566443*2342349993.887);

SetDecimalSep(old_decSep );
SetDecimal(old_Dec);
LimsSetDigit(old_Digit);

:RETURN;
```

Related Functions

FloatFormat()

Description

Purpose

Set the display format for a floating point numeric.

Description

FloatFormat() is used to format floating point numbers. Note that FloatFormat() affects only the display format of numbers and not the actual numeric precision of calculations.
This functiona cannot be called directly from StarLIMS. Use LimsAPICall().

Parameters

Syntax

FloatFormat(<fValue>, <iLen>, <iDec>) ---> fFormattedValue

Arguments

<fValue> Any numeric value.
<iLen> The desired length of the display of <fValue>, including decimal digits, decimal point, and sign. A value of -1 means that only significant digits to the left of the decimal point will be displayed (any left padding will be suppressed).
<iDec> The desired number of decimal digits in the display of <fValue>. A value of -1 means that only significant digits to the right of the decimal point will be displayed (any right padding will be suppressed).

Example

Examples

These examples use FloatFormat() to display the same number using three different formats. Note the number of leading spaces in each result:

```
LimsApiCall("FloatFormat", "FLOAT", "CAVORT20.DLL", 1234.546, 12,2);
 /*      1234.55;
LimsApiCall("FloatFormat", "FLOAT", "CAVORT20.DLL", 1234.546, -1,4);
 /*      1234.5460;
LimsApiCall("FloatFormat", "FLOAT", "CAVORT20.DLL", 1234.546, 12,-1);
 /*      1234.546;
```

Related Functions

Process Functions

Branch() – Obsolete

Note:

This function is obsolete. The :LABEL **statement is also obsolete. The use of this statements is not longer recommended.**

Description

Used to jump or branch to a specific label in the Action expression. The function is used with other LIMS and SQL functions. The specific label is defined in the parameters. Each label needs to be unique in the Action expression.

Parameters

Branch(Label)

Input Type: (string)

Returns: (empty string)

Example

```
:LABEL01;
    :IF nVar2 > nVar1;
        Branch("LABEL04");
    :ELSE;
        Branch("LABEL01");
    :ENDIF;
    Branch("LABELEND");

:LABEL04;
    StsMes("Successful completion");

:LABELEND
```

This example creates a loop as part of an action expression. When nVar1 is less than nVar2 the action loops or branches back to LABEL01. When nVar2 is greater than nVar1 the action branches to LABEL04 which will display a system message, "Successful Completion".

Branch("LABELEND") is used to branch to the end of the action.

Related Functions

DeleteInLineCode()

Description

Used to delete the block of code that is defined under the global variable "InLineCode_Name".

Parameters

DeleteInLineCode(Var)

Input Types: (string)

Returns: (logic - .T. = successful, .F. = variable doesn't exist)

Example

```
DeleteInLineCode("InLineCode_Name")
```

Related Functions

DisplayProperties()

Description

Used to display a list of all the available properties and methods that are available for a given OLE object.

Parameters

DisplayProperties(OLE Object)

Input Types: (object)

Returns: (empty string)

Example

```
DisplayProperties(OLE object);
```

Related Functions

DoProc()

Description

StarLIMS introduced for calling procedures a new internal executable function called: DoProc. The function is very flexible and allows exploiting all facilities provided by the local procedures in the same way as ExecFunction does with functions.

Parameters

x :=DoProc("<ProcName>", [{<parameters array>}]);

Input Types: (string, array)

The elements putted in square brackets are optional.

Returns:

In case the local procedure doesn't return any element you can call directly DoProc without using it's returned value.

(what ever the action returns via the :RETURN command.)

Example

Sample 1. Defining and calling a procedure that has no parameters and doesn't return values:

```
:DECLARE s1, s2, result;

s1 := "Star";

s2 := "LIMS";

DoProc("Concat"); /*:DO Concat;

:RETURN result;


:PROCEDURE Concat;

        result := s1 + s2;

:ENDPROC;
```

As you can see from this example, the code inside the procedure has unlimited access to variables defined outside. Both forms of calling a procedure can be used in this case.

Sample 2. Defining and calling a procedure that has formal parameters and returns values.

```
:RETURN DoProc("Sum", {5});


:PROCEDURE Sum;

:PARAMETERS n;
```

```
        :DECLARE sum, i;
        sum := 0;
        i := 0;
        :WHILE (i+=1) <= n;
        sum += i;
        :ENDWHILE;
        :RETURN sum;
:ENDPROC;
```

This form of procedure can only be called using the DoProc function.

Sample 3. Recursive procedures:

```
:RETURN DoProc("Prod", {5});

:PROCEDURE Prod;
:PARAMETERS n;
        :IF n <= 1;
                :RETURN 1;
        :ELSE;
                :RETURN n * DoProc("Prod", {n-1});
        :ENDIF;
:ENDPROC;
```

Related Functions

Also see the :PARAMETERS command, or the PrmCount() function, or the ExecUDF() function.

Also see the GetMethList() function, or the GetPropList() function, or the LimsOLEControl() function.

EndLimsOLEConnect()

Description

Used to shut down the connection to an OLE auto server from StarLIMS.

Parameters

EndLimsOLEConnect(VarOLEServerID)

Input Types: (variable OLE ServerID)

Returns: (Empty String).

Example

```
EndLimsOLEConnect(objCrystalOLE)
```

ObjCrystalOLE is the variable used for LimsOleConnect().

Related Functions

Also see the InsertOLEControl() function, or the LimsOLEConnect() function, or the LimsOLEControl() function.

ExecAction()

Description

Used to run another action from the current action. The action is identified by Window ID and Action ID. The Window ID is a string that identifies the application window. The Action ID is a string that identifies the specific action and can be assigned to the action when it's created. If needed, an Action ID can be added to the action from the Actions window.

If a Where Clause is specified, the data is filtered before the action is executed. Database is specified if the data is located in another database. Where Clause and Database are optional parameters.

Parameters

ExecAction(Window ID, Action ID, Where Clause, Database)

Input Types: (string, string, string, string)

Returns: (empty string)

Example

The following **ExecAction()** parameters perform the Audit Comment action, AUDCOM, in the Audit Trail (AUDTR) window.

```
ExecAction("AUDTR", "AUDCOM")
```

Related Functions

ExecFunction()

Description

Used to send an array of parameters to an ActionID which was written to receive these parameters via the :PARAMETERS command. You can us the PrmCount() function to check the number of parameters that were passed.

Parameters

ExecFunction(ActionID, {Array of Parameters})

Input Types: (string, array)

Returns: (what ever the action returns via the StartLimsTimer() function or the :RETURN command.)

Example

The following **ExecFunction()** passes the numeric variables "nA", and "nB" to the corresponding :PARAMETERS command in the ActionID "ACT001".

```
ExecFunction("ACT001", {nA,nB} );
```

Related Functions

Also see the :PARAMETERS command, or the PrmCount() function, or the ExecUDF() function.

ExecUDF()

Description

Used to send an array of parameters to an ActionID which was written to receive these parameters via the :PARAMETERS command. You can us the PrmCount() function to check the number of parameters that were passed.

Parameters

ExecUDF(Expression, {Array of Parameters})

Input Types: (Expr, array)

Returns: (what ever the expression returns via the StartLimsTimer() function or the :RETURN command.

Example

The following **ExecUDF** executes the expression stored in the character variable "cA" with the corresponding parameter “nOrdNo”.

```
:DECLARE ACT, nOrdNo, cA;

ACT := “Action111”;

nOrdNo := GetCurrent(“ORDERS”,””ORDNO);

cA := Lsearch("Select ACTION from ACTIONS where ACTIONID =
‘‘+ACT+"' "," DICTIONARY");

ExecUDF(cA, {nOrdNo});
```

Related Functions

Also see the ExecFunction() function, or the :PARAMETERS command, or the PrmCount() function.

FileWait()

Description

Used for synchronization, this function causes the system to wait for a specified amount of time (Delay Seconds) and then waits until the file name (File Name) is deleted or until the time out period (Time Out Period) is reached, whichever occurs first. The file name (File Name), the specified amount of time (Delay Seconds) and time out period (Time Out Period) are defined in the parameters.

From within an action it is possible to launch a DOS application, such as "Ver. 6 RUNDCU". However, Windows will not wait for the DOS application to finish, and continues processing. In this example, the continued process depends on what RUNDCU has generated. Therefore, in order to stop the process and allow RUNDCU to do it's job, it is recommended to use **FileWait**, as shown in the example below.

Parameters

FileWait(File Name, Delay Seconds, Time Out Period)

Input Types: (string, numeric(sec.), numeric(sec.))

Returns: (empty string)

Example

To execute RUNDCU, create the following action expression:

```
Lrename("STOP.XXX", "STOP.STP");

LimsExec(RUNDCU.BAT);

FileWait("STOP.STP");
```

The batch file, RUNDCU.BAT includes the command:

```
RUNDCU. . .

Rename   STOP.STP   STOP.XXX
```

The above action first creates a file called STOP.STP by renaming an existing file, STOP.XXX. The LimsExec command launches the batch file. The **FileWait** command causes subsequent commands to wait until the batch file is done. The last command in the batch file renames the file STOP.STP to STOP.XXX. This removes the block and enables subsequent commands to be executed.

Related Functions

GetAllInLineCode()

Description

Used to return an array of all inline code blocks defined.

Parameters

GetAllInLineCode()

Input Types: (empty string)

Returns: (array)

Example

```
GetAllInLineCode()
```

Related Functions

GetFieldObj()

Description

Used to return the Field Object Name.

Parameters

GetFieldObj(AppName, Field Name)

Input Types: (string, string)

Returns: (object)

Example

```
oVal1 := GetFieldObj( "RESOR","NUMRES"));
```

Related Functions

GetInLineCode()

Description

Used to store the block of code in variable VAR1, which later in the action will be called by ExecUDF(VAR1) function.

Parameters

GetInLineCode()

Input Types: (string)

Returns: (string)

Example

```
VAR1 := GetInLineCode("InLineCode_Name");

ExecUDF(VAR1);
```

The last line executes the code block.

Related Functions

GetMethList()

Description

Used to return an array of all the methods within an object.

Parameters

GetMethList(Object)

Input Types: (object)

Returns: (array)

Example

GetMethList(Object)

Related Functions

Also see the DeleteAllInLineCode() function, or the GetPropList() function, or the LimsOLEControl() function.

GetPropList()

Description

Used to return an array of all the properties within an object.

Parameters

GetPropList(Object)

Input Types: (object)

Returns: (Array)

Example

GetPropList(LimsApp("RESOR"))

Related Functions

Also see the DeleteAllInLineCode() function, or the GetMethList() function, or the LimsOLEControl() function.

InsertOLEControl()

Description

Used to create an OLE view with in a from view window in an established container window.

Parameters

InsertOLEControl(Window Object, OLE Control ID,Xorig,Yorig,Width, Height)

Input Types: (object, string, numeric, numeric, numeric, numeric)

Returns: (object)

Example

```
:DECLARE MO,APID,X;

MO := LimsApp("MATRL");

APID := "Shell.Explorer.2";

X := InsertOLEControl (MO,APID,20,20,300,300);

X:Show();

ExecInternal(X,"NAVIGATE","www.starlims.com");
```

The second and third lines establish variables to be used in the following lines of code. The fourth line will create the internet browser object and the sixth line will pass a method to the browser (Internet Explorer in this case) to go to the website www.starlims.com .

The fifth line X:Show(); makes the OLE control visible on the form.

X represents the object returned by **InsertOLEControl** , and Show() is a method of this objects, method that makes the OLE control visible on the form.

Related Functions

Also see the LimsOLEConnect() function, or the EndInLineCode() function, or the LimsOLEControl() function.

IsPath()

Note:

This function is obsolete, but it is still supported. It is recommended that you use the FileSupport() function instead.

Description

Used to check the validity of the specified directory defined in the parameters. If the directory exists, a True logic is returned. If it doesn't, a False logic is returned.

Parameters

IsPath(Specified Directory)

Input Types: (string)

Returns: (logic)

Example

```
IsPath("C:\WINDOWS")
```

If this directory exists, a True logic is returned.

Related Functions

Also see the FileSupport() function.

KeepGoing()

Description

Used for synchronization. After sending a DDE request to a DDE server, such as Microsoft Access or Excel, the action expression keeps going while the DDE request runs in the background. If a logical condition is used and returns a value of True, the expression process stops until a value of False is returned.

Parameters

KeepGoing(Logical Condition)

where:

.T. = stops the expression process until False is returned.

.F. = resumes the expression process

Input Types: (logical)

Returns: (empty string)

Example

Within an action expression that sends a DDE request to Microsoft Excel, the **KeepGoing** parameter is used as shown below.

```
DDE := RegDDEClient ("EXCEL");

:IF Empty(DDE);

        Branch ("LABEL001");

:ELSE

       Branch ("LABEL003");

:ENDIF;

KeepGoing();

:LABEL003;
    DDEServerExec(DDE, "SYSTEM", "Sheet1",'[CLOSE]');
    EndDDEClient(DDE);
```

Using a logical condition:

```
KeepGoing(.Not. Empty(LimsApp("My Window")).
```

KeepGoing stops processing the current expression until the user closes the window, "My Window". After the window is closed, **KeepGoing** resumes the process of the current action.

Related Functions

KillLimsTimer()

Description

Used to stop the timer TimerNo and rearranges the timer list. If, for example, we have 5 timers running on a window and we want to stop timer #4 using this function, then timer #4 is discarded and timer #5 becomes the new timer #4 .

Parameters

KillLimsTimer(Window Object, TimerNo)

Window Object The window object for which the timer was defined

TimerNo The unique identifier of the timer per window.

Input Types: (object,)

Returns: (empty string)

Example

```
KillLimsTimer(LimsApp("RENOR"),4);
```

- stops timer #4 defined on this window.

Related Functions

Labort()

Description

Used to stop the query creation of a LIMS Window. Implemented *only* in the window's SQL Select statement. The function cancels the completion of an SQL Select statement query. This can happen when the statement includes **If** or Lcase()- Obsolete conditions where the logical result activates the **Labort** function.

Parameters

Labort()

Input Types: (empty string)

Returns: (empty string)

Example

```
ORD := Lselect("Select * from ORDERS where Status = 'DONE'");

:IF Len(ORD)=0;

    Labort( );

:ENDIF;

"Select * from ORDERS"
```

This selects the all records in the Orders table with a status of "DONE". If there are no records with a status of "DONE" then the query is aborted.

Related Functions

LaunchApp()

Description

Used to launch a child application from a parent window by using an action button. The function opens the child window when the action button is clicked and will close the window when the parent window is closed. The parent and child window names are defined in the parameters. The reasons why this function is used instead of adding the child window as a menu option on the menu bar is to let the user open a window through an action instead of a menu option.

Parameters

LaunchApp(Child Window ID, Parent Window ID, Flag - .T. or .F.)

.F. (False) = action launches the child window and continues with the next statement in the action expression (default).

.T. (True) = action launches the child window and *then waits until the child window is closed* before continuing with the next statement in the action expression.

Input Types: (string, string, logic)

Returns: (empty string)

Example

The following **LaunchApp** parameters:

```
LaunchApp("ORDRS", "MATRL")
```

Opens the ORDERS Table window (ORDRS) from the MATERIALS Table window (MATRL).

Related Functions

LCommit()

Description

Used to make sure that the last changes that the user did to the data in a Lims Window reached the database. In other words, the function is used to save the modified data to the database. By default, when a user closes an application, the data that was modified is saved to the database. This function will only be used when the user doesn't close the window but desires to run an action for which he needs the newly modified data.

Parameters

LCommit(Window Object)

Input Types: (object)

Returns: (empty string)

Example

The parameter Window Object may be missing and then all the modifications from all the opened windows will be committed to the database. If the parameter is passed, only the modified data in that window will be committed.

```
LCommit();

LCommit(LimsApp("ORDERS"));
```

Related Functions

Let()

Description

Used to change the value of an *existing* named variable. It is recommended to use this function instead of adding another named variable with Lset. Adding extra, unnecessary variables reduces system performance and increases memory fragmentation.

Parameters

Let(Variable Name, New Expression)

Input Types: (string, any)

Returns: (empty string)

Example

To change the value for the named variable "V" from '0' to 'V + 1' the following **Let** parameters are used:

```
V := V + 1;
```

Related Functions

Also see the :DECLARE statement.

LimsCleanup()

Description

When you use StarLIMS (and any other application in general), you dynamically allocate memory as needed, and then release it. This process is transparent, Visual Objects takes care of the memory release. Visual Objects has a separate thread of execution, called a garbage collector. Whenever the used memory reaches a certain amount, and the available memory is low, VO starts the garbage collector. This collector releases the unused memory and performs a memory defragmentation (thus compacting both used and free memory).

This process can conflict sometimes with memory allocated for strings, arrays, and objects (especially big ones). You can force a memory collection by calling LimsCleanup(). Immediately after this call, the memory is optimized, and the garbage collector won't likely be invoked again very soon. So for the next few operations, you have big chances of not being disturbed by the garbage collector (unless you manipulate huge strings, arrays, or objects).

If there is the need to manipulate big strings and arrays or to perform operations with strings or arrays in a loop, to prevent memory fragmentation use the functions ArrayNew(), AevalA(), ArrayCalc() for the arrays and String Add(), StringCreate(), StringGet(), StringClean(), StringKill().

Parameters

LimsCleanup()

Input Type: (none)

Returns: (empty string)

Example

```
LimsCleanup();
```

Related Functions

LimsExec()

Description

Used to execute a windows or non-windows application. The function opens the application defined in the parameters. If the application defined in the parameters does not exist, an error message will appear on the screen.

Parameters

LimsExec(Application Name, Flag - .T. or .F.)

where:

.T. = default, opens the full application.

.F. = optional, opens the application in minimized mode (icon).

Input Type: (string, logic)

Returns: (empty string)

Example

By using the following **LimsExec** parameters:

LimsExec("c:\MSOFFICE\WINWORD.EXE BTEXDESC.DOC",.F.)

The BTEXDESC.DOC file in Microsoft Word opens in minimized mode (icon). If the file doesn't exist, the user will receive an error message.

Note:

LimsExec does *not* wait until the application is completed. After the application is loaded, control is returned immediately to the previous, or 'calling' action.

Related Functions

LimsOLEConnect()

Description

Used to establish a connection to an OLE auto server. Allows StarLIMS to use the properties and methods of the OLE Server.

Parameters

LimsOLEConnect(OLE ServerID, ErrorFlag, ForceNewServer)

Input Types: (**OLE ServerID** – string, **ErrorFlag** - logic, **ForceNewServer -** logic)

The ErrorFlag indicates the behavior in case of an error. If the flag is .T. (true – default) the error message is displayed in case of an error. If the flag is .F. (false) the error message is not displayed and a NULL_OBJECT is returned. When returning an OLE connection, if the OLE server is already instantiated, it's object is beeing returned – otherwise a new server is created. The **ForceNewServer** flag (default .F. - false) indicates if a new server is created regardless of the fact that a server with the same ServerID exists.

Returns: (Object) if connection is made otherwise returns (Empty Object).

Example

```
APID := "SHELL.EXPLORER.2";       /***  Internet Explorer OLE Server ID;
X := LimsOLEConnect(APID);
ExecInternal(X,"NAVIGATE","www.starlims.com");
 /*** executes the NAVIGATE method of this OLE Control.
```

Related Functions

Also see the InsertOLEControl() function, or the EndInLineCode() function, or the LimsOLEControl() function.

LimsOLEControl()

Description

Used to connect to the OLE Server specified by the 2nd parameter and create a container where the output of the server is to be displayed.

Parameters

LimsOLEControl(Owner, OLE ServerID, X-Orig, Y-Orig, Width, Height, Title)

Input Types: (object, string, numeric, numeric, numeric, numeric, string)

Returns: (object)

Example

```
X := LimsOLEControl(LimsApp("SHELL"),"SHELL.EXPLORER.2", 10, 10, 300,
300, "Browser Example");

ExecInternal( X, "NAVIGATE","www.starlims.com");
```

The first line will create the internet browser and the second line will pass a method to the browser (Internet Explorer in this case) to go to the website www.starlims.com .

Related Functions

Also see the InsertOLEControl() function, or the LimsOLEConnect() function, or the EndInLineCode() function.

Lkill()

Description

Used to delete named variables that are no longer needed. This helps to free up memory and resources.

Note:

You should always end your action expressions with Lkill statements that kill all named variables unless :DECLARE is used**.**

Parameters

Lkill(Variable Name)

Input Types: (string)

Returns: (empty string)

Example

A named variable was added to an action expression:

```
:DECLARE MYVAR;

MYVAR := "June"
```

To delete this variable from the system the following **Lkill** statement is used:

```
Lkill("MYVAR")
```

Related Functions

Also see the :DECLARE statement.

LockTable()

Description

Most SQL servers do not enable explicit locking of tables, where the decision to lock a table can be, for example, based upon internal rules. The **LockTable** function is used, for example, when the user needs to calculate the next order number where it is necessary to first find the maximum order number, increment it by 1, and insert this new value in the table. If multiple stations try accessing this information simultaneously, there could conceivably be a situation where 2 stations receive the same order number. In this example, this situation is unacceptable. If a uniqueness check is made, where the Order # is the primary key, a user trying to add an order will receive an engine message error, which is not desirable.

LockTable is used to prevent other users from accessing the same table on processes that require dedicated access. If another user tries to access a table that is locked, a 'False' logic will be returned and the user will need to wait until the first user is finished or the timeout period has expired, and the UnLock function is activated. (See UnLockTable() for more details.)

Important:

The locked table needs to be unlocked by the UnLockTable()UnLockTable_ function when the process is completed. Otherwise, the table will remain unavailable to all system users.

Note:

There MUST be an entry in the LIMSLOCK table for the table that you are trying to lock.

The table name and timeout period (seconds) are defined in the parameters.

Parameters

LockTable(Table Name, Time Out Period)

Input Types: (string, numeric(sec))

Returns: (logic)

Example

The following **LockTable** parameters used in an action expression:

```
:IF LockTable("ORDERS",20);
        Branch("Label02");
:ELSE
        Branch("Label01");
:ENDIF;

 :Label01;

  LockTable("ORDERS",20);

     ---
  UnLockTable("ORDERS");
  Branch("LabelEnd");

 :Label02;
```

```
UsrMes("Message", "Orders Table locked by another user.  Try again in a few seconds.");

:LabelEnd
```

First checks if the ORDERS table is locked. If it is (true), the user receives a message. If it's not true (false), the action branches to Label 01, locks the Orders table for 20 seconds, and continues the action expression. It then unlocks the Orders table and branches to Label End.

Related Functions

Also see the UnLockTable() function.

Lwait()

Description

Used for synchronization, this function causes the system to wait for the specified Time Out Seconds.

Parameters

Lwait(Time Out Period)

Input Types: (numeric(sec.) - can be a decimal value)

Returns: (empty string)

Example

```
Lwait(2.5)
```

Causes the system to wait 2.5 seconds.

Related Functions

LwSet() – No longer supported

Note:

This function is no longer supported.

Description

Used to set variables which are local to a given LIMS application window. Up to 100 variables can be set for a given window. This function is used together with **LwVar** described below. The Window ID, Variable Number (1 - 100) and Expression are defined in the parameters.

Parameters

LwSet(Window ID, Variable Number, Expression)

Input Types: (string, numeric, any)

Returns: (empty string)

Note:

For internal variables, single and two dimensional arrays can also be stored using LwSet.

Example

In the following example, the Window ID is DESMN, the variable number is 1, and the expression is a **LlookUp** function which Returns a Window ID.

```
LwSet("DESMN",1,LlookUp('WNDMAINT', 'WINDOW ID',
'*****',{'WINDOW ID'}))
```

Related Functions

PrmCount()

Description

Used to return the number of parameters that were passed by the ExecFunction() or the ExecUDF() functions. This function is very useful to check and ensure that the correct number of parameters were passed.

Parameters

PrmCount()

Input Types: (empty)

Returns: (numeric)

Example

NCnt := PrmCount();

This will return into the variable nCnt the number of variable that were passed to either the ExecFunction() or the ExecUDF() functions.

Related Functions

Also see the ExecFunction() function, or the :PARAMETERS command, or the ExecUDF() function.

RunApp()

Description

Used to execute a windows or non-windows application. The function opens the application defined in the parameters. If the application defined in the parameters does not exist, an error message will appear on the screen.

Parameters

RunApp(Application)

Input Type: (string)

Returns: (empty string)

Example

By using the following **RunApp** parameters:

RunApp("c:\MSOFFICE\WINWORD.EXE BTEXDESC.DOC")

The BTEXDESC.DOC file in Microsoft Word opens in minimized mode (icon). If the file doesn't exist, the user will receive an error message.

Note:

RunApp will place StarLIMS in the wait mode until the application is completed. After the application is loaded, control is returned immediately to the previous, or 'calling' action.

Related Functions

SendToPrinter()

Description

Used to print a text to the printer in text mode or RTF(Rich Text Format) mode. The text, printer and RTF indicator are defined in the parameters.

Parameters

SendToPrinter(sText, sPrinter, logicRtf)

Input Types: (string , string, logic)

sPrinter – If not specified, the user is prompted to select a printer. If it is specified, the string must be a string returned by GetPrinters(). To specify a printer, first call the function GetPrinters(), select the printer, and then use the return of the function. For more information, please see **GetPrinters()**.

logicRTF – The default value is .F. (false)

Returns: (empty string)

Example

SendToPrinter(“This is the text”, “HP LaserJet 4000 Series PS,winspool,SCC45561_P3” , .T.);

This string “HP LaserJet 4000 Series PS,winspool,SCC45561_P3” is returned by GetPrinters().

Related Functions

SetErrorHandler()

Description

This function enables the system error-handler.

Parameters

SetErrorHandler(bShowErrorFlag)

Where:

bShowErrorFlag – Indicates if the error is displayed or not on the screen. Defaults to .T. (True)

Input Types: (logic)

Returns: (empty string)

Example

```
SetErrorHandler();
```

Also see

Related Functions

SubmitToBatch()

Description

Used to execute heavy database routines in the background as a separate task.

This function sends the routine to an external program called LimsBtch.exe which executes the action against the database.

Parameters

SubmitToBatch(Expr, Sync)

Expr The name of the action or variable which stores the expression to execute or expression itself in double quotes.

Sync Flag that indicates if the function works synchronous or asynchronous.

Input Types: (string, logic)

Returns: (empty string)

The default value for Sync is .T. (true), but if the user wants to wait for the execution of the action submitted to the Batch, than .F. (false) should be used instead.

Example

```
:DECLARE sAction;

:BEGININLINECODE "ACT";
    :DECLARE a;
    a := "ZZZZ";
    DisplayOnconsole(a,"the value of a");
    :RETURN;
:ENDINLINECODE;

sAction:=GetInLineCode("ACT");

SubmitToBatch(sAction, .T.);

:RETURN;
```

In this example, sAction is passed to the batch and executed. Using the function DisplayOnConsole, the developer can debug the action that is passed to the batch processor. The batch processor does not have a visual debugging tool like the debugger in StarLIMS, and this function is used instead. If an error is produced in the action runs on the batch processor, an error log is created in the form of a text file in the current StarLIMS folder. The error log filename is the same as the current date (YYYYDDMM) and has the extension “log” (Ex: 20011008.log).

If a StarLIMS action needs to be passed to the batch, read into a variable the text of the action from the dictionary and pass the variable to SubmitToBatch().

Action := SQLExecute(“Select ACTION from ACTIONS where ACTIONID = ‘ACT_00120’”);

Action := Action[1,1];

The batch processor does not contain any graphical elements, it doesn’t have user prompts and was stripped of any user interaction mechanism but the DisplayOnConsole() function. This function exists only in the Batch Processor so it cannot be used from StarLIMS directly. The function will be displayed in Expression Edit as not recognized by StarLIMS (displayed in pink) but it will be recognized by the Batch Processor.
The developer should run only heavy database routines on the batch processor and not actions that are using graphical elements such as

windows, user messages and prompts.

Related Functions

SuspendLimsTimer()

Description

Used to deactivate a Timer associated with a particular window. To create a timer use StartLimsTimer(). To activate a timer use ResumeLimsTimer().

Parameters

SuspendLimsTimer(Window Object ,TimerID)

Input Types: (object, numeric)

Returns: (empty string)

TimerID The unique identifier of the timer per AppName window.

Window Object – the result of a LimsApp(Window ID) call

Example

```
SuspendLimsTimer(LimsApp("WND-000001"), 5 )
```

Related Functions

TraceOff()

Description

Used to turn off the Debugging Mode started by TraceOn().

If TraceOn() received a file as a parameter, TraceOff() ends the debugging and closes the file.

Parameters

TraceOff()

Input Types: (empty string)

Returns: (empty string)

Example

The following shows how Debugging Mode is turned off after being switched on with **TraceOn** in the middle of an action.

```
TraceOn()
    SQL code…..

TraceOff()
```

Related Functions

Also see the TraceOn() function.

TraceOn()

Description

Used to start the Debugging Mode for a portion of an action, when is encountered.

This function can receive as an optional parameter a file, and in this case the debug will be done in this file and the default debugger will not be shown.

Parameters

TraceOn(File)

Input Types: (string)

Returns: (empty string)

Example

The following shows how Debugging Mode is turned on in the middle of an action and stays on until TraceOff() is encountered.

```
TraceOn()
    SQL Code……

TraceOff()
```

or :

```
TraceOn("C:\debug.txt");
    SQL Code……

TraceOff()
```

Related Functions

UnlockTable()

Description

Used to unlock a table that has been previously locked by the LockTable() function.

Important:

The locked table needs to be unlocked when the process is completed. Otherwise, the table will remain unavailable to all system users.

Parameters

UnLockTable(Table Name)

Input Types: (string)

Returns: (string)

Example

The **UnLockTable** function is used with the LockTable() function as shown in the following section of an action expression:

```
:IF LockTable("Orders",20);

    Branch("Label02");

:ELSE;

    Branch("Label01");

:ENDIF;

:Label01;
    LockTable("Orders",20);
    ---
    ---
    UnLockTable("Orders");
    Branch("LabelEnd")

:Label02;
    UsrMes("Message", "Orders Table locked by another user.  Try again in a
    few seconds.");

:LabelEnd
```

This expression first checks if the Orders table is locked. If it is (true), the user receives a message. If it's not true (false), the action branches to Label 01, locks the Orders table for 20 seconds, and continues the action expression. It then unlocks the Orders table and branches to Label End.

Related Functions

Also see the LockTable() function.

UndeclaredVars()

Description

Used to enable or disable the use of undeclared variables in the current StarLIMS session.

Parameters

UndeclaredVars(Flag - .T. or .F.)

where

.T. = at the runtime, when the system encounters an undeclared variable, it creates it end continue with the execution (Default).

.F. = when an undeclared variable is encountered, a prompt will ask for the permission to create the variable or to raise an error. This setting is useful for debugging purposes.

Input Types: (logic)

Returns: (empty string)

Example

```
UndeclaredVars(.F.);
```

Related Functions

Related Functions

WinShellExec()

Description

This function is used to open the file passed in the parameters with the system's registered viewer.

Parameters

WinShellExec(FileName)

FileName – string – The complete path of the filename that the user wishes to open

Input Types: (string)

Returns: (empty string)

Example

WinShellExecute("C:\Clients.txt"**)**;

This function opens the specified text file in Notepad if this is the registered program for viewing text files.

Related Functions

Security Functions

ChkPassword()

Description

Used to check the validity of the user password for the given user name.

Parameters

ChkPassword(UserName, UserPassword)

UserName	The name of the user whose password you are checking.
UserPassword	The password you want to validate.
Input Types:	(string, string)
Returns:	(logic)

Example

```
ChkPassword(“John Doe”,”john123”);
```

Related Functions

Serial Communications Functions

BeginSerial()

Description

Used to initiate the COM port and to create a communication object.

The collection of the data from the COM port can be done in background in a buffer associated with this connection object. The SHELL window will handle the background activity.

This function sets the size of the I/O buffers for Transmit and Receive and initiates the port with values such as Baud Rate, Parity, Word Length, Stop Bits.

The function returns a connection object which can be further manipulated or an error message if the port could not be initialized.

Parameters

BeginSerial(sComNo, nBaud, nParity, nWordLen, nStop, bBckg, nRxSize, nTxSize, aReplyArr)

ComNo – the COM port.

nBaud – the baud rate of the COM port in Bits per Second.

nParity – the Parity of the COM port.

nWordLen – the length of the word for the COM port.

nStop – the number of Stop Bits.

bBckg – flag sets background collection from the port.

nRxSize – the size of the receive buffer.

nTxSize – the size of the transmission buffer.

aReplyArray – a two-dimensional array of possible strings read from the COM port and the actions that will be triggered by these values. The last array in this parameter is an array of two elements; one represents a timeout value in seconds and the second one an action to be executed when this timeout expires. The last array in this parameter is mandatory.

Input Types: (string, numeric, numeric, numeric, numeric, logic, numeric, numeric, array)

Returns: (object)

Example

```
:DECLARE oCOMM

oCOMM := BeginSerial("COM1",9600,0,8,1,.T., 512,512,
{ {“ERROR”, “ACT-111”} ,{15,”ACT-112”} } );
```

This function initializes COM1 with 9600 bits per second, Parity – None, word length – 8 and 1 stop bits, with background collection.

The last argument , the array tells this function that if the string “ERROR” is captured from the COM port to run the action “ACT - 111”. The second element of the array tells this function to run the specified action when the 15 seconds of timeout expires. This last parameter is needed to provide a failsafe mechanism in case the expected string values don’t arrive. This way the user has control over the communication and can stop the process if the expected values are not read from the COM port. In the timeout action the user can use GetBckgSerial() function to read the

Receive Buffer and to determine a suitable course of action .

Here is another example:

```
:PUBLIC OPENCOM,CAPT;

:DECLARE aCAPT,nPLATENO,ENQ,BUFF;

:DECLARE nRUNNO,cCOL,K,S;

:IF .not. Empty(CAPT) ;

      StopAction();

:ENDIF;

CAPT:= .t.;

OPENCOM := BeginSerial("COM1",9600,0,8,1,.F.);

:WHILE CAPT;

      BUFF := AllTrim(ReadSerial( OPENCOM,3,"CRLF"));

      :IF .not. Empty(BUFF) ;

            UpdCurrent("IF99_WD000002",{"BEFWEIGHT"},{BUFF});

            DN:=SetInternal(LIMSAPP("IF99_WD000002"),
"SHOULDJUMP", "DN");

      :ENDIF;

/*** enter a condition to reset the value of CAPT and to exit the loop;

….

:ENDWHILE;

EndSerial( OPENCOM );

OPENCOM := "";
```

Related Functions

CommNo()

Description

Used to return the COM port that owns the buffer number passed as a parameter.

Parameters

CommNo(nBuffer)

nBuffer The number of the buffer.

Input Types: (numeric)

Returns: (string)

Example

CommNo(1);

Returns “Com1” if the number of the buffer associated with COM1 is 1.

Related Functions

EndSerial()

Description

Used to terminate the conection established by the communication object returned by BeginSerial() .

Parameters

EndSerial(oCOMM)

oCOMM The communication object returned by BeginSerial().

Input Types: (object)

Returns: (empty string)

Example

EndSerial(oCOMM);

Returns “” if it was successful or an error message if it failed.

Related Functions

GetBckgSerial()

Description

Used return the string accumulated in the background collection buffer if the communication session was opened with the background flag set to .T. See BeginSerial() function.

If the flag was not set to true, this function will return a error message and an empty string.
Otherwise it will return the accumulated string from the Receive Buffer.

Parameters

GetBckgSerial(oCOMM)

oCOMM The communication object returned by BeginSerial().

Input Types: (object)

Returns: (string)

Example

```
GetBckgSerial( oCOMM );
```

Returns the string accumulated in the Receive Buffer, if the parameter bBckg is passed as (.T.) to the BeginSerial() function, enabling the background collection in the Receive Buffer.

Related Functions

GetComms()

Description

Used to identify all the available RS232 communication ports.

Parameters

GetComms()

Input Types: (empty string)

Returns: (array)

Example

```
GetComms();
```

May return {“Com1”,”Com2”} if the COM1 and COM2 ports are available.

Related Functions

GetCurrentBufferNo()

Description

Used to return the number of the last buffer which triggered an action.

Parameters

GetCurrentBufferNo()

Input Types: (empty)

Returns: (numeric)

Example

GetCurrentBufferNo();

Returns 1 if the buffer number that triggered the last action is 1.

Related Functions

GetSerial()

Description

Used to initialize a COM port to read data from the instruments until the Time Out expires or the Terminator is received. The communication settings, Time Out, and Terminator are defined in the parameters.

Parameters

GetSerial(COM Number, Baud Rate, Parity, Word Len, Stop Bits, Time Out, Terminator)

where:

COM Number = COM1 - COM8

Baud Rate = 19200, 9600, 4800, 2400, 1200, 600, 300, or 150

Parity = 0 for none, 1 for odd, 2 for even

Word Len = 7 or 8

Stop Bits = 1 or 2

Time Out = seconds

Terminator = any string. (see example)

Input Types: (string, numeric, numeric, numeric, numeric, numeric, string)

Returns: (string)

Example

This example sets a variable name for the commonly used terminator, XON/XOFF. It then starts COM3 with the following parameters and reads the data from the instruments for 6 seconds or until the variable XONXOFF is received.

```
XONXOFF := Chr(17) + Chr(19);

GetSerial("COM3",9600,0,8,1,6,XONXOFF);
```

Related Functions

Also see the PutSerial() function.

PhoneDial()

Description

Used to dial a stored telephone number.

Parameters

PhoneDial(COM, PHONE#)

Input Types: (String, String)

Returns: (empty string)

Example

```
PhoneDial("COM2", "19549648663")
```

- dials the phone number passed as a parameter using the modem connected on the specified COM port.

Related Functions

PutSerial()

Description

Used to initialize a COM port in order to communicate with the instrumentation and transmit a text string from StarLIMS to the instruments. The communication settings and text string are defined in the parameters. This functions is also associated with the GetSerial() function.

Parameters

PutSerial(COM Number, Baud Rate, Parity, Word Len, Stop Bits, Text)

where:

COM Number = COM1 - COM8

Baud Rate = 19200, 9600, 4800, 2400, 1200, 600, 300, or 150

Parity = 0 for none, 1 for odd, 2 for even

Word Len = 7 or 8

Stop Bits = 1 or 2

Text = any string up to 1k

Input Types: (string, numeric, numeric, numeric, numeric, string)

Returns: (logic - .T. if successful, .F. if not successful)

Example

This example starts COM3 with the following parameters and transmits the text "Initiate testing" to the instrument connected to this port.

```
PutSerial("COM3",9600,0,8,1,"Initiate testing.")
```

Related Functions

Also see the GetSerial() function.

ReadSerial()

Description

Used to read from the buffer associated with the communication object returned by BeginSerial().

Parameters

ReadSerial(oCOMM, TimeOut, Terminator)

oCOMM The communication object returned by BeginSerial().

TimeOut The number of timeout seconds. If not passed it is defaulted to 1000.

Terminator The string that acts as a terminator. When this string is encountered, the function stops reading from the COM port and exits, returning the string read including the terminator.

Input Types: (object, numeric, string)

Returns: (string)

Example

ReadSerial(oCOMM,9,"XX");

This reads from the COM until the 9 seconds timeout expires or the string "XX" is encountered.

Related Functions

SetBckgSerial()

Description

Changes or clears the buffer for the background collection of data from the COM port associated with the communication object returned by BeginSerial(), only if the background collection is allowed by the bBckg flag in BeginSerial().

Returns .T. or .F. depending upon the success of the function.

This function may return an error message if the background collection for this communication object is not allowed.

Parameters

SetBckgSerial(oCOMM, InitStr)

oCOMM The communication object returned by BeginSerial().

InitStr The initial string that is written in the Receive Buffer. If this parameter is not passed, it is defaulted to "" (empty string).

Input Types: (object, string)

Returns: (logic)

Example

SetBckgSerial(oCOMM, "START");

Returns .T. if the initialization of the buffer with the specified string was successful.

Related Functions

WriteSerial()

Description

Used to write a string to the COM port associated with the communication object returned by BeginSerial(). The functions returns .T. if the specified string was successfully written to the port .

Parameters

ReadSerial(oCOMM, sStr)

oCOMM	The communication object returned by BeginSerial().
sStr	The string that is written to the COM port.
Input Types:	(object, string)
Returns:	(logic)

Example

WriteSerial(oCOMM, "STARTRUN");

Transmits the string “STARTRUN” to the instrument connected to the COM port associated with the oCOMM object

Related Functions

String Functions

AllTrim()

Description

Used to remove leading and trailing spaces from a string. Most commonly used with Str() and SubStr() functions where numbers have been converted to strings. The string to be trimmed is defined in the parameters.

Parameters

AllTrim(String)

Input Types: (string)

Returns: (String with leading and trailing spaces removed)

Example

The following example of **AllTrim** is used in a DCU method.

```
RelPosition :=Position+26;

ANALYTE := AllTrim(SubStr(CurrentTable,RelPosition,12));
```

The function **AllTrim** is used in order for the value of the variable ANALYTE to match the value of RESULTS.ANALYTE.

Related Functions

Also see the Left() function, or the Str() function, or the SubStr() function, or the Right() function.

ASC()

Description

Used to convert an a character value to an ASCII code. The ASCII code (from 0 - 255) is defined in the parameters.

Parameters

ASC(Character)

Input Types: (character)

Returns: (numeric - ASCII value).

Example

ASC(H)

Returns: 72

Also see the ASCIIStr() function.

Related Functions

ASCIIStr()

Description

Used to calculate the checksum value for a string. The function returns the accumulated ASCII numeric values for the given string.

Parameters

ASCIIStr(string)

Input Types: **(string)**

Returns: (Accumulated ASCII value as a string).

Example

ASCIIStr("Barry") Returns: 512 (Chr(66) + Chr(97) + Chr(114) + Chr(114) + Chr(121))

Related Functions

Also see the ASC() function.

At()

Description

Used to return the position of the first occurrence of a sub-string within a string. The **At()** and Rat() functions are used with SubStr(), Left(), and Right() to extract substrings. The sub-string and string to be searched for are defined in the parameters.

Parameters

At(Sub String Search, String Search)

Input Types: (string, string)

Returns: (The position of the first occurrence of Sub String Search within String Search. If Sub String Search is not found, 0 is returned.)

Example

The following examples show how the position of the first occurrence of a substring within a string is returned using the At() function.

At("a", "abcde")

Returns: 1

At("bcd", "abcde")

Returns: 2

At("a", "bcde")

Returns: 0

Related Functions

Also see the Rat() function, or the SubStr() function, or the Left() function, or the Right() function.

Chr()

Description

Used to convert an ASCII code to a character value. The ASCII code (from 0 - 255) is defined in the parameters.

Parameters

Chr(ASCII Code)

Input Types: (numeric)

Returns: (a single character that corresponds to the ASCII Code).

Example

Chr(7)

Returns: Bell sounds

Chr(72)

Returns: H

Related Functions

Empty()

Description

Used to determine if the result of an expression is empty. The expression (any data type) is defined in the parameters. The criteria for determining whether a value is considered empty depends on the data type of the Expression, according to the following rules:

Data Type	Contents
Array	NULL_ARRAY or empty array
Code block	NULL_CODEBLOCK
Date	NULL_DATE
Logic	FALSE
NIL	NIL
VOID	TRUE
Numeric	0
Object	NULL_OBJECT
PSZ	NULL_PSZ
PTR	NULL_PTR
String	Spaces, tabs, carriage return/line feed, or NULL_STRING
Symbol	NULL_SYMBOL

Parameters

Empty(Expression)

Input Types: (any type)

Returns: (Logic, TRUE if the expression results in an empty value; otherwise, FALSE.)

Example

The following example shows how **Empty** is used:

```
NUMINDAT := Lselect("Select Max(NUMINDAT) from CALCEXER where
ADDEDDATE = Today()");

:IF Empty(NUMINDAT[1,1]);

    :RETURN .T.;

:ELSE;

     :RETURN NUMINDAT[1,1] +1;

:ENDIF;
```

If the first row and column of the variable NUMINDAT is empty, TRUE is returned. It will be empty, in this case, when there is no entry in the ADDEDDATE field for the current date.

Related Functions

Left()

Description

Used to extract a substring beginning with the first character in a string.

The string from which the substring is extracted from and the numbers of characters to extract, starting from the left side, are defined in the parameters.

Left is the same as SubStr()(<String>, 1, <Count>).

Left, Right(), and SubStr() are often used with both the At() and Rat() functions to locate the first and/or the last position of a substring before extracting it.

Parameters

Left(String, Count)

where:

String = the string from which to extract characters.

Count = the number of characters to extract.

Input Types: (string, numeric)

Returns: (the extracted substring)

If Count is negative or 0, a NULL_STRING is returned. If Count is larger than the length of the string, the entire string is returned.

Example

The following example shows how **Left** extracts the first three characters from the left of the target string:

Left("ABCDEF", 3) *Returns: ABC*

Related Functions

Also see the AllTrim() function, or the Str() function, or the SubStr() function, or the Right() function.

Len()

Description

Used to return the length of a string or the number of elements in an array. The array or string to be measured is defined in the parameters.

Parameters

Len(Array / String)

where:

Array / String = the string or array to measure. In a string, each byte counts as 1, including an embedded null character (Chr(0)). A NULL_STRING counts as 0. In an array, each element counts as 1.

Input Types: (array or string)

Returns: (numeric - length of String or number of elements in Array)

Example

The following examples show how **Len** is used in various ways:

```
Len("string of characters")
```

Returns: 20

```
Len(NULL_STRING)
```

Returns: 0

```
Len(Chr(0))
```

Returns: 1

```
Len(ArrayNew(5))
```

Returns: 5

Related Functions

LimsString()

Description

Used to return any type of input (except arrays, and objects) as a string.

Parameters

LimsString(String)

Input Types: (string)

Returns: (string)

Example

```
:DECLARE X;
X := CtoD("01/01/2001");
:RETURN LimsString(X);
```

The function returns the date as a string.

Related Functions

Llower()

Description

Used to convert the uppercase and mixed case characters in a string to lowercase. **Llower** is related to Upper(), which converts lowercase and mixed case strings to uppercase. **Llower** is generally used to format strings for display purposes. It can, however, be used to normalize strings for case-independent comparison or indexing purposes. The string to be converted into lowercase characters is defined in the parameters.

Parameters

Llower(String)

where:

String = the string to convert to lowercase.

Input Types: (string)

Returns: (String with all alphabetic characters converted to lowercase. All other characters remain the same as in the original string.)

Example

The following example demonstrates a result of **Llower**:

```
Llower("1234 CHARS ")
```

Returns: 1234 chars

Related Functions

Also see the Upper() function, or the Lower() function.

Ltransform()

Description

Used to convert any value into a formatted string.

This function is a conversion function that formats character, date, logical, and numeric values according to a specified picture string that includes a combination of picture function and template strings. It formats data for output to the screen or the printer. The value to be formatted and the picture are defined in the parameters.

Function string: A picture function string that specifies formatting rules for the return value as a whole, rather than to particular character positions. The function string consists of the @ character, followed by one or more additional characters as listed below. If a function string is present, the @ character must be the leftmost character of the picture string, and the function string must not contain spaces. A function string can be specified alone or with a template string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

Function	Action
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays date in SET DATE format
E	Displays date in British format
R	Non-template characters are inserted
X	Displays DB after negative numbers
Z	Displays zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

Template string: A picture template string specifies formatting rules on a character by character basis. The template string consists of a series of characters, some with a special meaning, as shown below. Each position in the template string corresponds to a position in the Value. Because Ltransform() uses a template, it can insert formatting characters such commas, dollar signs, and parentheses.

Characters in the template string that have no assigned meaning are copied into the return value. If the @R picture function is used, these characters are inserted between characters of the return value; otherwise, they overwrite the corresponding characters of the return value. A template string can be specified alone or with a function string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

Template	Action
A,N,X,9,#	Displays digits for any data type
L	Displays logical as "T" or "F"
Y	Displays logical as "Y" or "N"
!	Converts an alphabetic character to uppercase
$	Displays a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Specifies a decimal point position

,	Specifies a comma position

Parameters

Ltransform(Value, Picture)

Input Types: (any type except an array or NIL value, string)

Returns: (a formatted string as defined by Picture)

Example

This example formats a number into a currency format using a template:

Ltransform(123456, "$999,999")

Returns: $123,456

This example formats a string using a function:

Ltransform("to upper", "@!")

Returns: TO UPPER

Related Functions

Ltrim()

Description

Used to remove leading spaces from a string. Most commonly used with Str() function where numbers have been converted to strings. The string to be trimmed is defined in the parameters.

Parameters

Ltrim(String)

Input Types: (string)

Returns: (String with leading spaces removed)

Example

The following is an example of **Ltrim()**.

Ltrim(" AAA")

Returns "AAA"

Related Functions

Rat()

Description

Used to return the position of the last occurrence of a substring within a string.

Rat is like At(), which returns the position of the first occurrence of a substring within another string. The **Rat** and At() functions are used with SubStr(), Left(), and Right() to extract substrings.

The sub string and string to be searched for are defined in the parameters.

Parameters

Rat(Sub String Search, String Search)

Input Types: (string, string)

Returns: (The position of the last occurrence of Sub String Search within String Search. If Sub String Search is not found, 0 is returned.)

Example

The following examples show how the position of the last occurrence of a substring within a string is returned using the **Rat()** function.

Rat("a", "abcdeabcde")

Returns: 6

Rat("bcd", "abcdeabcde")

Returns: 7

Rat("a", "bcde")

Returns: 0

Related Functions

Also see the AllTrim() function, or the At() function, or the Left() function, or the Str() function, or the Right() function.

Right()

Description

Used to return a substring beginning with the rightmost character.

The string from which the substring is extracted from and the numbers of characters to extract, starting from the rightmost character, are defined in the parameters.

Right is the same as SubStr(String, Count). For example, Right("ABC",1) is the same as SubStr("ABC",-1). **Right** is related to Left(), which extracts a substring beginning with the leftmost character in the string.

The **Right**, Left(), and SubStr() functions are often used with both the At() and Rat() functions to locate either the first and/or the last position of a substring before extracting it.

Parameters

Right(String, Count)

where:

String = the string from which to extract characters.

Count = the number of characters to extract.

Input Types: (string, numeric)

Returns: (the extracted substring)

If the numbers of characters to extract (Count) is 0, a NULL_STRING is returned. A negative value is not allowed since a WORD cannot be negative. If Count is larger than the length of the string, the entire string is returned.

Example

This example shows the relationship between **Right** and SubStr():

Right("ABCDEF", 3)

Returns: DEF

SubStr("ABCDEF", 3)

Returns: CDEF

SubStr("ABCDEF", -3)

Returns: DEF

Related Functions

Also see the AllTrim() function, or the At() function, or the Left() function, or the Str() function, or the Val() function.

SEval()

Description

Execute a code block for each of the individual characters in a string.

SEval() is a character function that evaluates a code block once for each character in a string, passing the ASCII value and the character index as arguments. The return value of the code block is ignored. All characters in <cString> are processed unless either the <nStart> or the <nCount> argument is specified.

Parameters

SEval(<cString>, <cbBlock>, [<nStart>], [<nCount>]) ---> cString

<cString> The string to scan.

<cbBlock> The code block to execute for each character encountered.

<nStart> The starting character. A negative value starts from the end. The default value is 1 if **<nCount>** is positive and the length of **<**

cString> if **<nCount>** is negative.

<nCount> The number of characters to process from **<nStart>**. A negative value steps downward. The default is all characters to the end of the string.

Input Types: (string, string, numeric, numeric)

Returns: (string)

Example

This example uses SEval() to extract the street number from a string containing the complete street address:

```
:DECLARE cStreetNumber, cAddress;

cStreetNumber := “”;

cAddress := "1209 West Golden Lane";

SEval(cAddress, {|c| cStreetNumber += If(IsDigit(Chr(c)), Chr(c), “”)});
```

cStreetNumber is now “1209”

Related Functions

StrEval()

Description

Execute a code block for each of the individual characters in a string, changing the contents of the argument as well as the return value.

SEvalA() is identical to SEval() in that they both evaluate a code block once for each character of a string, passing the ASCII value and the character index as arguments. The only difference is that, while SEval() ignores the return value of the code block, SEvalA() assigns the return value to the original string. See SEval() for details.

Parameters

SEval(<cString>, <cbBlock>, [<nStart>], [<nCount>]) ---> cString

<cString> The string to scan.

<cbBlock> The code block to execute for each character encountered.

<nStart> The starting character. A negative value starts from the end. The default value is 1 if **<nCount>** is positive and the length of **<**

cString> if **<nCount>** is negative.

<nCount> The number of characters to process from **<nStart>**. A negative value steps downward. The default is all characters to the end of the string.

Input Types: (string, string, numeric, numeric)

Returns: (string) - a string of characters that have been processed by the code block.

Example

This example uses SEvalA() to change part of a string to uppercase:

```
:DECLARE cString;

cString := "He was doa.";

? SEvalA(cString,{|cChar| Asc(Upper(Chr(cChar)))}, 8, 3) // "He was DOA."

? cString          // "He was DOA."
```

Related Functions

Str()

Description

Used to convert a numeric expression to a string.

This function is often used to concatenate numbers to strings. Thus, it is useful for creating codes for items, such as part numbers, from numbers and for creating order keys that combine numeric and character data.

Str() is like Ltransform(), which formats numbers as strings using a mask instead of length and decimal specifications. The inverse of **Str()** is **Val()** which converts numbers formatted as strings to numeric values.

The numeric expression to be converted to a string (Numeric Expression), the string length to return (Length), and the number of decimal places to return (Decimals) are defined in the parameters.

Parameters

Str(Numeric Expression, Length, Decimals)

where:

Numeric Expression = the numeric expression to convert to a string.

Length = the length of the string to return, including decimal digits, decimal point, and sign. A value of -1 specifies that only the significant whole digits to the left of the decimal point are returned and suppresses any right padding. Decimal places, however, are still returned as specified in Decimals. If Length is not specified, the length returned is the actual length of the Numeric Expression.

Decimals = the number of decimal places in the return value. A value of -1 specifies that only the significant digits to the right of the decimal point are returned. The number of whole digits in the return value, however, are still determined by the Length argument. If Decimals is not specified, the decimals returned is the actual decimals (if used) of the Numeric Expression.

Input Types: (numeric, numeric, numeric)

Returns: (string.)

Rounding is determined as follows:

If Length is less than the number of decimal digits required for the decimal portion of the returned string, the return value is rounded to the available number of decimal places.

If Length is specified, but Decimals is omitted (no decimal places), the return value is rounded to an integer.

Example

These examples demonstrate the range of values returned by **Str**, depending on the parameters specified:

```
Str(123.45)
```

Returns: 123.45

```
Str(123.45, 4)
```

Returns: 123

```
Str(123.45 * 10, 7, 2)
```

Returns: 1234.50

Str(123.45, 10, 1)

Returns: 1234.5

Related Functions

Also see the AllTrim() function, or the Left() function, or the SubStr() function, or the Right() function.

StringAdd()

Description

This function adds (concatenates) a static string to another static string.

The first string should be long enough to accommodate the second one. Otherwise data is written past its border and memory corruption occurs.

This way if we will have a concatenation like A := A+B. The position where the second string will be inserted into the first is passed in the parameters. If the last parameter is not passed, the second string will be inserted into the first string at the beginning of the portion from the first string that is filled with NIL.

Parameters

StringAdd(String, SubString, Position)

Input Types: (string, string, numeric)

Returns: (empty string)

Example

The static string A contains : ”AAAAA ” length is 15, it has the last 10 characters filled with NIL.

The static string B contains : ”BBBBB ” length is 10, it has the last 5 characters filled with NIL.

The length of the string A is 15 and it can accommodate a 10 character string from the 6th character to the 15th.

After StringAdd(A, B, 6) the string A will contain: “AAAAABBBBB ”

Related Functions

StringClean()

Description

This function takes a static string (created with StringCreate()) of a certain length and initializes a certain portion of the string with NIL.

The string, the starting position and the count are passed in the parameters.

Parameters

StringClean(String, Position, Count)

Input Types: (string, numeric, numeric)

Returns: (empty string if Position > Len(String))

Example

We have the string sString := “AAAAAsssss”

After StringClean(sString, 6, 5) the new string will look like “AAAAA “, the last 5 characters containing NIL.

Related Functions

StringCreate()

Description

Creates a string with a given length in the static memory. This function is used mainly for operations on strings that are performed in a loop. For example a operation on strings can be the concatenation. By default when two strings A and B are concatenated as A = A+B the length of the two strings is summed, a new string is allocated with this length and the two initial strings are invalidated, A now pointing to the newly allocated string that results after the concatenation. The space that the two strings were occupying in the memory becomes free space. Especially on big strings and after a few of this operations the memory becomes fragmentized by these empty spaces. If we would want now to allocate another big string, we would need a continuous memory block to accommodate the size of the string. If such memory block is not found, Visual Objects starts a proceeds called “garbage collecting”. This process does a memory defragmentation relocating small empty memory blocks into a compact zone of free memory, creating space for the following memory allocations. This is a process that takes time and can negatively influence the performance of an action that performs operations on strings. Strings that are allocated in the static memory are not affected by the garbage collector so that the performance of any string operations is greatly improved. In order to use strings that are allocated in the static memory special functions were developed: StringAdd() for the concatenation, StringGet() that returns a certain static allocated string, StringClean() that empties a static allocated string and StringKill() that de-allocates the string.

Parameters

StringCreate(Length)

Input Types: (integer)

Returns: (string)

Example

StringCreate(10) returns a string with the length of 10, allocated in the static memory. The 10 elements of the string are null.

Instead of:

```
:DECLARE sMyStr;
sMyStr:="";
sMyStr:=sMyStr+"AAA";
sMyStr:=sMyStr+"BBB";
sMyStr:=sMyStr+"CCC";
```

for a better performance and to avoid memory fragmentation by relocating new strings with each operation use:

```
:DECLARE sMyStr;
sMyStr:=space(9);
StringAdd(sMyStr,"AAA",1);
StringAdd(sMyStr,"BBB",4);
StringAdd(sMyStr,"CCC",7);
```

Another way is to use the StringCreate,StringGet,StringClean,StringAdd and StringKill functions:

```
:DECLARE sMyStr,RetVal;
sMyStr:=StringCreate(1000);   /* should be the maximum size of the string;
StringClean(sMyStr);          /* into a loop, this can be necessary;
StringAdd(sMyStr,"AAA");
StringAdd(sMyStr,"BBB");
StringAdd(sMyStr,"CCC");
RetVal:=StringGet(sMyStr);
```

```
StringKill(sMyStr)
:RETURN RetVal;
```

Related Functions

Also see the StringAdd () function, the StringGet() function, the StringClean() function, and the StringKill() function.

StringGet()

Description

Returns the relevant portion of a static string. If for example the user has defined a static string like “AAA “ (length 10, first 3 characters are “A” and the next 7 are NIL) , this function will return the string as “AAA” (length 3).

Parameters

StringGet(String)

Input Types: (string)

Returns: (string)

Example

```
:DECLARE sString;
sString := StringCreate(10);
StringAdd(sString,”AAA”);
/*** the string is now represented in memory as “AAA “ ;
StringGet( sString);
Returns: “AAA”
```

Related Functions

StringKill()

Description

Frees the memory allocated for the static string that is passed in the parameters.

Parameters

StringKill(String)

Input Types: (string)

Returns: (empty)

Example

```
StringKill( sString );
```

Related Functions

StrSrch()

Description

Used to search for a substring within a string with the possibility (depending of the flag) to either look for the n-th occurrence or start the search for the first occurrence starting with the n-th (index) character of the string.

The string and substring to search in , the occurrence or starting point of the search and the flag are defined in the parameters.

The value of the Flag is by default false (.F.).

If the Flag is (.F.), the function will find the indicated occurrence of the substring in the string. If the flag is (.T.), the third parameter will be regarded as an index in the string from which to start the search for the first occurrence of the substring in the string.

The function returns an index in the string which indicates the position in the string where the substring was found.

If the last two parameters are not passed, the function acts like the At() function. If the third parameter is passed as 1 or 0 (zero) than the first occurrence of the string will be returned.

Parameters

StrSrch(SubString, String, Occurrence/Index, Flag)

where:

SubString = the substring for which to search.

String = the string in which to search.

Occurrence/Index = the occurrence number (1st, 2nd, …) or the index in the string from which to start looking for the first occurrence of the SubString in the String.

Flag = Flag that indicates whether the 3rd parameter indicates the number of the occurrence or an index in the string. Default is (.F.)

Input Types: (string, string, integer, logic)

Returns: (integer)

Example

This example uses **StrSrch** to search for the 2nd occurrence of the substring “compute”

```
cString := "To compute or not to compute?"

StrSrch("compute", cString, 2, .F.);
```

Returns: 22

This next example uses **StrSrch** to search for the 1st occurrence of the substring “compute” starting with the 10th character of the string.

```
cString := "To compute or not to compute?"

StrSrch("compute", cString, 10, .T.);
```

Returns: 22

Related Functions

StrTran()

Description

Used to search and replace characters within a string.

The string and substring to search in (String Name, Search for SubString), the substring that will replace the original substring (Replace with SubString) are defined in the parameters.

Parameters

StrTran(String, Search for SubString, Replace with SubString)

where:

String = the string in which to search.

Search for SubString = the substring for which to search. All occurrences of Search for SubString are replaced unless Start or Count is specified. Note that **StrTran** replaces substrings and, therefore, does not account for whole words.

Replace with SubString = the substring with which to replace Search. If this argument is not specified, Search is replaced with a NULL_STRING.

Input Types: (string, string, string)

Returns: (A new string with the specified occurrences of Search for SubString replaced by Replace with SubString.)

Example

This example uses **StrTran** to convert a post modern analog to a famous quotation:

```
cString := "To compute or not to compute?"

StrTran(cString, "compute", "be")
```

Returns: To be or not to be?

Related Functions

StrZero()

Description

Used to convert a numeric expression to a string and pad it with leading zeroes instead of blanks.

This function is useful in displaying numbers, creating codes such as part numbers from numeric values, and creating order keys that combine numeric and character data. **StrZero** is similar to the Str() function except that the padding character is a zero ("0") instead of a blank. For more information, see Str().

The numeric expression to be converted to a string (Numeric Expression), the string length to return including zeros (Length), and the number of decimal places to return (Decimals) are defined in the parameters.

Parameters

StrZero(Numeric Expression, Length, Decimals)

where:

Numeric Expression = the numeric expression to convert to a string.

Length = the length of the string to return, including zeroes, decimal digits, decimal point, and sign.

Decimals = the number of decimal places in the return value.

Input Types: (numeric, numeric, numeric)

Returns: (string)

Example

This example uses **StrZero()** to convert a 3-digit number to a 5-character string:

```
StrZero(987, 5, 0)
```

Returns: "00987"

Related Functions

SubStr()

Description

Used to extract a substring from a string.

SubStr is related to the Left() and Right() functions, which extract substrings beginning with leftmost and rightmost characters in the target string (String Name).

The **SubStr**, Right(), and Left() functions are often used with both the At() and Rat() functions to locate either the first and/or the last position of a substring before extracting it. They are also used to display or print only a portion of a string.

The string from which the substring is extracted from, the starting position, and the numbers of characters to extract are defined in the parameters.

Parameters

SubStr(String, Start, Count)

where:

String = the string from which to extract characters.

Start = the starting position in String. If Start is positive, it is relative to the leftmost character in String. If Start is negative, it is relative to the rightmost character in String. If Start is zero, a NULL_STRING is returned.

Count = the number of characters to extract. If omitted, the substring begins at Start and continues to the end of the string. If Count is greater than the number of characters from Start to the end of String, the extra is ignored.

Input Types: (string, numeric, numeric)

Returns: (the substring. If the substring is not present, a NULL_STRING is returned.)

Example

These examples extract the first and last name from a variable:

```
Name := "John Nobelsons"

SubStr(Name, 1, 4)
```

Returns: John

```
SubStr(Name, 6)
```

Returns: Nobelsons

```
SubStr(Name, -9, 3)
```

Returns: Nob

The next example shows how **SubStr** is used to extract the WHERE clause from a CURRENTSQL, such as: "Select * from ORDERS where ORDERS.STATUS = 'HOLD' order by ORDNO";

```
FRM := At('where', "CURRENTSQL")+6;
```

Returns: 27

```
ORD := At('order by ',"CURRENTSQL");
```

Returns: 48

```
:IF ORD=0;
        WHR := SubStr("CURRENTSQL", FRM);
:ENDIF;
:IF ORD#0;
        WHR := SubStr("CURRENTSQL", FRM, ORD-FRM);
:ENDIF;
```

Returns: ORDERS.STATUS = 'HOLD'

This example can be used to add a WHERE clause in the Lprint() function for printing reports.

In the above example, the variable FRM receives the value returned by the At() function, which is the position in CURRENTSQL after the word "Where" (Where…+6). ORD receives the value of the position in CURRENTSQL at the beginning of the word "Order By".

If there is no "Order By" statement (ORD=0), the WHERE clause begins at position FRM and continues to the end of CURRENTSQL. If there is an "Order By" statement, the WHERE clause begins at position FRM and ends at the position before "Order By" (Count = ORD-FRM).

Related Functions

Also see the AllTrim() function, or the Left() function, or the Str() function, or the Right() function.

Upper()

Description

Used to convert the lowercase and mixed case characters in a string to uppercase. All other characters remain the same as in the original string.

Upper is related to Lower(), which converts uppercase and mixed case strings to lowercase. **Upper** is often used to format strings for display purposes. It can also be used to normalize strings for case-independent comparison or indexing purposes.

The string to be converted into uppercase characters is defined in the parameters.

Parameters

Upper(String)

Input Types: (string)

Returns: (string)

Example

The following example demonstrates a result of **Upper**:

```
Upper("123 char = <>")          Returns: 123 CHAR = <>
```

Related Functions

Also see the Lower() function, or the Llower() function.

ValidateString()

Description

Used to check to see if a string contains only ASCII printable characters. (The valid range is 32-127).

Parameters

ValidateString(cStr)

cStr a regular string

Example

ValidateString("ABC") Returns a T (true).

Related Functions

Statements

:BEGINCASE

Description

Used to note the beginning of one or more CASE statements that will be evaluated sequentially in vertical order. You must also use the **:**ENDCASE statement after the last **:CASE** statement in the group.

Parameters

:BEGINCASE

Example

```
:BEGINCASE;

:CASE nVAL1 > 4;
     AB := “>4";

:CASE nVal1 =<4;
     AB := “=<4";

:ENDCASE;
```

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :EXITCASE statement, or the :OTHERWISE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ENDIF statement, or the :ELSE statement.

:BEGININLINECODE

Description

Used to start a block of code under the global variable.

Parameters

:BEGININLINECODE “Codeblock Name”

Codeblock Name - the name of codeblock defined between :BEGININLINECODE and :ENDINLINECODE statements

Example

```
:BEGININLINECODE "ACT";

:DECLARE TreeWhere;

:IF UserInput(200,200,500,300,'Select
',{"FD,120,200,200,25,SE,Field","FR,20,150,150,25,SE,From","TO,250,15
0,150,25,SE,To"});

   TreeWhere:=" "+FD+" between  "+FR +" and "+TO+" ";

:ELSE;

   TreeWhere:="**********";

:ENDIF;

:RETURN TreeWhere;

:ENDINLINECODE;

Action:=GetInLineCode("ACT");
```

Related Functions

:CASE

Description

Used to execute a one or more SQL statements if the expression for the CASE statement is true. If the statement is false the next CASE statement is validated. If no other CASE statements exist before an **:**ENDCASE statement, the action then drops to the **:**ENDCASE and then continues.

Parameters

:CASE Expr

Input Types: (SQL Expression)

In the statement above, if the value of nVAL1 is greater than 4 then the line following the first **:CASE** will be executed. If it is not, then the next **:CASE** statement will be evaluated. If the value is equal to or less than 4 then the statement following it would be executed. In either case, all **:CASE** statements would be evaluated until an **:**ENDCASE statement is detected.

Example

```
:CASE nVAL1 > 4;
     AB := ">4";

:CASE nVal1 =<4;
     AB := "=<4";

:ENDCASE;
```

Related Functions

Also see the Lcase()- Obsolete function, or the :ELSE statement, or the :BEGINCASE statement, or the :EXITCASE statement, or the:OTHERWISE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ENDIF statement.

:CHECKPARAM – Obsolete

Note:

This function is obsolete - See the PrmCount() **function**

Description

Used to verify if all requested values have been entered. If all data has not been entered, acts as a StartLimsTimer().

Parameters

:**CHECKPARAM list of parameters (fields) to check**

Input Types: (keyword)

Example

```
:CHECKPARAM RN1, RN2; <<RN1>>+<<RN2>>
```

If either RN1 or RN2 does not contain a value, the calculation is not performed. The script stops at the **:CHECKPARAM** line. The function ChkPrm()- Obsolete still works, but returns either **.T.** or **.F.**

Related Functions

Also see the PrmCount() function.

:DATABASE - Obsolete

Note:

This statement is obsolete - Use the SQLExecute() and Lselect() functions

Description

In the course of an expression, changes the target database. The default database is the main database.

Parameters

:DATABASE Database Name

Input Types: (keyword)

Example

```
:DATABASE DICTIONARY
```

This changes the target database to **DICTIONARY**. From this point forward in the expression, all SQL statements will target the database **DICTIONARY**

Related Functions

:DECLARE

Description

Enables you to list all the local variables for an action in one place that will be released from memory when the action is complete.

Parameters

:DECLARE Variable, Variable;

Example

```
:DECLARE cTEMP1, cTEMP2;
```

This will release the variables cTEMP1 and cTEMp2 from memory when the action is complete.

Related Functions

Also see the :PUBLIC statement.

:DEFAULT

Description

Sets default values for the parameters received by a StarLIMS action. This statement works in conjunction with the :PARAMETERS statement.

Parameters

:DEFAULT Parameter, Value;

Example

```
:PARAMETERS sFileName, sFolder, nFileCount;

:DEFAULT sFolder, 'C:\StarLIMS9\Temp';

:DEFAULT nFileCount, 0;
```

Let's assume this code exists at the beginning of a StarLIMS action. This action receives 3 parameters sFileName, sFolder, nFileCount. In case the last two parameters are not passed to the action, the :DEFAULT statement gives them default values. If these parameters are used before assigning values to them, an error will be raised, so this way we can make sure that they have valid valued before they are used.

Related Functions

Also see the :PARAMETERS statement.

:ELSE

Description

Used to actuate an alternate portion of SQL code if the condition in the If()- Obsolete statement was false.

Parameters

:ELSE

Input Types: (none)

Example

```
:IF n > nCTR;
    X := X + 1

:ELSE;
    X := 0;

:ENDIF;
```

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :BEGINCASE statement, or the :EXITCASE statement, or the:OTHERWISE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ENDIF statement.

:ENDCASE

Description

Used to terminate one or more **:**CASE statements.

Parameters

:ENDCASE

Input Types: (none)

Example

```
:CASE nVAL1 > 4;
    AB := ">4";

:CASE nVal1 =<4;
    AB := "+<4";

:ENDCASE;
```

In the statement above, if the value of nVAL1 is greater than 4 then the line following the first **:**CASE will be executed. If it is not, then the next **:**CASE statement will be evaluated. If the value is equal to or less than 4 then the statement following it would be executed. In either case, all **:**CASE statements would be evaluated until an **:ENDCASE** statement is detected.

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :BEGINCASE statement, or the :EXITCASE statement, or the:OTHERWISE statement, or the If()- Obsolete statement, or the :ENDIF statement, or the :ELSE statement.

:ENDIF

Description

Used to end a **:IF** statement.

Parameters

:ENDIF;

Input Types: (none)

Example

```
:IF n > nCTR;
    X := X + 1

:ELSE;
    X := 0;

:ENDIF;
```

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :BEGINCASE statement, or the :EXITCASE statement, or the:OTHERWISE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ELSE statement.

:ENDINLINECODE

Description

Used to end a block of code previously initiated by BeginInLineCode function. You must have the by BeginInLineCode function in your code to use the EndInLineCode function.

Parameters

:ENDINLINECODE

Input Types: (none)

Example

See :BEGININLINECODE

Related Functions

:ENDWHILE

Description

Used to end a **:**WHILE loop statement.

Parameters

:ENDWHILE

Input Types: (none)

Example

```
:WHILE nVAL1 < 4;
    nVal1 := nVal1 + .5;

:ENDWHILE;
```

In the statement above, the statement will be executed only if the value of nVal1 is less than 4. It will continue to loop until the statement becomes false. When it becomes false or if it is false to start with, the action will continue directly after the ENDWHILE statement.

Related Functions

See also the **:**EXITWHILE statement, or the **:**LOOP statement, or the **:**WHILE statement.

:ERROR

Description

Marks the beginning of the Error Handler. In case of an error in an action, instead of signaling the error, the action will jump to the error handling code and execute it. The error handling routine needs to be at the end of the action after the last **:**RETURN statement

Parameters

:ERROR;

Input Types: (none)

Example

```
/*** Sets the built-in error-handler not to display the default error message;
SetErrorHandler(.F.);
/*** This line will raise a syntax error message;
ArrayCalc({2,3,-4a},"MIN");
:RETURN;
/*** The Error Handling routine is executed when a error will be encountered;
:ERROR;
    UsrMes("Warning!”,” Error 1");
/*** This line resumes the execution of the action with the next line after the one that caused the error;
:RESUME;
```

Related Functions

See also the **:**RESUME statement.

:EXITCASE

Description

Enables you to exit from a **:CASE** statement immediately without dropping / evaluating the next **:CASE** statement. None of the CASE statements that follow the point where this statement appears will be evaluated. The control jumps immediately to the **:ENDCASE** point.

Parameters

:EXITCASE;

Example

```
:EXITCASE;
```

This will exit the **:CASE** statement immediately.

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :BEGINCASE statement, or the :OTHERWISE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ENDIF statement, or the :ELSE statement.

:EXITWHILE

Description

Enables you to exit from a **:**WHILE looping statement before the logical statement is false.

Parameters

:EXITWHILE;

Example

```
:EXITWHILE;
```

This will exit the **:**WHILE loop immediately.

Related Functions

See also the **:**ENDWHILE statement, or the **:LOOP** statement, or the **:**WHILE statement.

:IF

Description

Enables you to check if a condition exists and execute SQL code based upon the results of the comparison or condition. The ENDIF statement must also be used with the statement. You can also use the :ELSE statement if there is a alternative action or event to be executed if the logical condition is false.

Parameters

:IF [Logical Expression];

Example

```
:IF n > nCTR;
    X := X + 1

:ELSE;
    X := 0;

:ENDIF;
```

If the value of n is greater that the value stored in nCTR then the statement (X := X + 1) will be executed, otherwise the value of X will be set to 0.

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :BEGINCASE statement, or the :EXITCASE statement, or the:OTHERWISE statement, or the :ENDCASE statement, or the :ENDIF statement, or the :ELSE.

:INCLUDE

Description

The include statement can be used to include prior to execution the content of a server script inside other server script in the place where :INCLUDE statement appears. The include statements are processed before script execution. Using :INCLUDE statement the above script can be split in 2 scripts, one containing procedures and one the actual code.

Parameters

:INCLUDE [CategoryName.ScriptName];

Examples

Script05

```
:INCLUDE Research.Script06;


:DECLARE a, b;

:DECLARE sum, prod;


a := 2; b := 3;


sum := DoProc("MakeSum", {a,
b});

prod := DoProc("MakeProd",
{a, b});


:RETURN sum / prod;
```

Script06

```
:PROCEDURE MakeSum;

:PARAMETERS a, b;

      :DECLARE sum;

      sum := a + b;

      :RETURN sum;

:ENDPROC;


:PROCEDURE MakeProd;

:PARAMETERS a, b;

      :RETURN a * b;

:ENDPROC;
```

Sample 1. Defining and calling a procedure that has no parameters and doesn't return values:

```
:DECLARE s1, s2, result;

s1 := "Star";

s2 := "LIMS";

DoProc("Concat");      /*:DO Concat;

:RETURN result;


:PROCEDURE Concat;

      result := s1 + s2;

:ENDPROC;
```

As you can see from this example, the code inside the procedure has unlimited access to variables defined outside. Both forms of calling a procedure can be used in this case.

Sample 2. Defining and calling a procedure that has formal parameters and returns values.

```
:RETURN DoProc("Sum", {5});

:PROCEDURE Sum;

:PARAMETERS n;

    :DECLARE sum, i;

    sum := 0; i := 0;

    :WHILE (i+=1) <= n;

        sum += i;

    :ENDWHILE;

    :RETURN sum;

:ENDPROC;
```

This form of procedure can only be called using the *DoProc* function.

Sample 3. Recursive procedures:

```
:RETURN  DoProc("Prod", {5});

:PROCEDURE Prod;

:PARAMETERS n;

    :IF n <= 1;

        :RETURN 1;

    :ELSE;

        :RETURN n * DoProc("Prod", {n-1});

    :ENDIF;

:ENDPROC;
```

Sample 4. Complete script for testing local procedures functionality.

```
:DECLARE r;

:DECLARE s;

s := "";

DoProc("HelloWorld");

s := s + r + Chr(13);

s := s + LimsString(DoProc("Sum", {5})) + Chr(13);

s := s + LimsString(DoProc("Sum2", {5})) + Chr(13);

s := s + LimsString(DoProc("Prod", {5})) + Chr(13);


/*WriteText("C:\ret.txt", s);  /* For testing with LimsBatch;

:RETURN s;


:PROCEDURE HelloWorld;

    r := "Hello, World!";

:ENDPROC;


:PROCEDURE Sum;

:PARAMETERS n;

    :DECLARE sum, i;

    sum := 0; i := 0;

    :WHILE (i+=1) <= n;

        sum += i;

    :ENDWHILE;

    :RETURN sum;

:ENDPROC;


:PROCEDURE Sum2;

:PARAMETERS n;

   :RETURN DoProc("Sum",{n});

:ENDPROC;
```

```
:PROCEDURE Prod;
:PARAMETERS n;
    :IF n <= 1;
        :RETURN 1;
    :ELSE;
        :RETURN n * DoProc("Prod", {n-1});
    :ENDIF;
:ENDPROC;
```

Sample 5. Testing *:INCLUDE* functionality.

In order to run this sample we need to setup the execution environment:

- Move HelloWorld and Sum procedures from previous script to a new action called: *Local Procedures Lib 02*;
- Move Sum2 and Prod procedures to a new action called: *Local Procedures Lib 01* and add in front of this action a new like:
  :INCLUDE Local Procedures Lib 02;
- In front of remaining lines from the initial script add the following line:
  :INCLUDE Local Procedures Lib 01;

If you followed these steps, you should have the following actions:

Local Procedures Lib 02

```
:PROCEDURE HelloWorld;

        r := "Hello, World!";

:ENDPROC;



:PROCEDURE Sum;

:PARAMETERS n;

        :DECLARE sum, i;

        sum := 0; i := 0;

        :WHILE (i+=1) <= n;

                sum += i;

        :ENDWHILE;

        :RETURN sum;

:ENDPROC;
```

Local Procedures Lib 01

```
:INCLUDE Local Procedures Lib 02;



:PROCEDURE Sum2;

:PARAMETERS n;

      :RETURN DoProc("Sum",{n});

:ENDPROC;



:PROCEDURE Prod;

:PARAMETERS n;

       :IF n <= 1;

                :RETURN 1;

       :ELSE;

                :RETURN n *
DoProc("Prod", {n-1});

       :ENDIF;

:ENDPROC;
```

Test Local Produres 02

```
:INCLUDE Local Procedures
Lib 01;



:DECLARE r;

:DECLARE s;



s := "";

DoProc("HelloWorld");

s := s + r + Chr(13);

s := s +
LimsString(DoProc("Sum",
{5})) + Chr(13);

s := s +
LimsString(DoProc("Sum2",
{5})) + Chr(13);

s := s +
LimsString(DoProc("Prod",
{5})) + Chr(13);



/*WriteText("C:\ret.txt",
s);  /* For testing with
LimsBatch;

:RETURN s;
```

Now run the script: *Test Local Produres 02*.

You'll see that the result is the same as the result obtained with Sample 4. You can notice that the *:INCLUDE* works recursively, including all actions referred in the current running action or in any of it's children.

The :INCLUDE keyword is very powerful allowing developers to write modular code. Some of it's features are listed here:

- The statement can be used to include any action inside other action but is very useful when is used together with local procedures for building libraries of local procedures (as showed in current example);
- The statement can appear in any position inside of an action, StarLIMS processor evaluating all :INCLUDE statements before starting to execute the current action;
- The :INCLUDE keyword takes only one literal parameter, the name of the action that is intended to be included. Variables are not accepted as parameters for :INCLUDE.
- If the action is not found, the :INCLUDE statement is ignored.

Features and limitations

- From inside a procedure you can access/ override the values of variables declared outside the procedure if you don't declare them again locally;
- A variable declared inside the body of a procedure has local visibility and is automatically disposed when the procedure ends. Declaring a variable with the same name as an existing external variable will not override the external variable;
- Procedures have local visibility inside the action where are declared. You can call any procedure from the current action but you cannot call a procedure located in other action (for calling procedures located in other actions you should first include that action in current action using the *:INCLUDE* keyword);
- Procedures have only one level and cannot be nested;

:LABEL – Obsolete

Note:

This statement is obsolete.

Description

Used as a target for the Branch function. In the statement, the word **:LABEL** is suffixed by a *unique* label ID. A label ID can only be used once in the expression.

Parameters

:LABEL Label ID

Input Types: (keyword)

Example

```
:LABEL01;
    If(nVar2 > nVar1, Branch("LABEL04"), Branch("LABEL01"));
    Branch("LABELEND");

:LABEL04;
    StsMes("Successful completion");

:LABELEND
```

This example creates a loop as part of an action expression. When nVar1 is less than nVar2 the action loops or branches back to Label 01. When nVar2 is greater than nVar1 the action branches to Label 04 which will display a system message, "Successful Completion".
Branch("LABELEND") is used to branch to the end of the action.

Related Functions

Also see the Branch() function.

:LOOP

Description

Enables you to redirect to the top of the **:**WHILE loop from within the middle of the loop.

Parameters

No parameters

Example

```
:LOOP;
```

This will cause the action to jump to the top of the **:**WHILE before executing the **:**ENDWHILE statement.

Related Functions

See also the :ENDWHILE statement, or the :EXITWHILE statement, or the :WHILE statement.

:OTHERWISE

Description

Enables you to execute SQL code in a **:CASE** group whether or not any of the **:CASE** statements are evaluated as true, unless an :EXITCASE is encountered.

Parameters

:OTHERWISE;

Example

```
:OTHERWISE;
```

This will cause the SQL code following the statement to be executed whether or not any of the **:CASE** statements were evaluated as true, unless an :EXITCASE statement is encountered.

Related Functions

Also see the Lcase()- Obsolete function, or the :CASE statement, or the :BEGINCASE statement, or the :EXITCASE statement, or the :ENDCASE statement, or the If()- Obsolete statement, or the :ENDIF statement, or the :ELSE statement.

:PARAMETERS

Description

Used as to receive a the parameters set to it by the ExecFunction()**()** and the ExecUDF()**()** functions. The function receives a comma delimited variable list.

Parameters

:PARAMETERS X,Y,Z;

Input Types: (any)

Note:

The values passed by the associated functions are an array of values, with each value corresponding with a specific element in the comma delimited list. When the comma delimited list has 5 values, the array of values passed to it MUST also have the same number of values.

Example

Where ExecFunction("ACT123",{"Barry", "Grace", "Gelu"})

```
:PARAMETERS PARA1, PARA2, PARA3;
    UsrMes("PARA1", PARA1);
    UsrMes("PARA2", PARA2);
    UsrMes("PARA3", PARA3);

:LABELEND
```

This example will open up three user messages boxes and display the values contained in PARA1, PARA2, and PARA3. Notice that there are three values passed in the **ExecFunction()** and there are three values expected in the PARAMETERS command.

Related Functions

Also see the ExecFunction() function, or the PrmCount() function, or the ExecUDF() function.

:PROCEDURE

Description

A procedure can be declared in any point inside of a StarLIMS action, but it's a good practice to declare procedures either at the beginning or at the end of the code and not inside the flow of a code. Local procedures can have only one level and cannot be nested. If your application requires nesting code you can use a combination of StarLIMS actions and *procedures*.

Parameters

For declaring procedures you should use the following syntax if the procedure takes no parameters:

:PROCEDURE Sub1;

…
:RETURN …;

:ENDPROC;

…or the following form, if the procedure accepts input parameters:

:PROCEDURE Sub;

:PARAMETERS a,b;

…
:RETURN …;

:ENDPROC;

Example

```
:PROCEDURE Sum;
:PARAMETERS a,b;
    :DECLARE c;
    c := a + b;
     :RETURN c;
:ENDPROC;
```

Receives two numeric parameters a and b. Declares a variable c. Sums up the values of a and b, assigns the sum to c and returns it.

To call this procedure use:

```
DoProc("Sum", {a, b});
```

Related Functions

:PUBLIC

Description

Enables you to declare public variables that will not be released from memory when the action is complete.

Parameters

:DECLARE Variable, Variable;

Example

:DECLARE cTEMP1, cTEMP2;

Related Functions

Also see the :DECLARE statement.

:REGION

Description

Enables you to declare a segment of text, all in one section having a name, and retrieve it throught GetRegion later.

Parameters

:REGION regionName;

….

:ENDREGION;

Example

```
/* Set some variables for client...;
:REGION form_OnLoad;
   navigator.Variables.Set("MYUSERNAME",
                           "$MYUSERNAME$");
   navigator.Variables.Set("MYUSERLANG",
                           "$MYUSERLANG$");
:ENDREGION;



:REGION ImportScript;

      ExecFunction( "TestsImpExp.Imp_Import", {
       "$importID$", $testsInfo$, $syncTests$ } );


:ENDREGION;
```

Related Functions

Also see the GetRegion statement.

:REPEAT - Obsolete

Note:

This statement is obsolete.

Description

Used to create a loop that repeats an action. The keyword used as the label name is defined in the parameters.

Note:

StopRepeat must be used with :REPEAT to stop the loop and continue with the next statement. See the example below.

Parameters

:REPEAT (Label Name)

Input Types: (keyword)

Example

The **:REPEAT** statement used within an action expression to create a loop is shown below:

```
I := 0;

:REPEAT LABELOOP;
    UsrMes("I",1);
    I := I+1;
    :IF I > 3;
        StopRepeat( );
    :ENDIF;

:LABELOOP
    UsrMes("END OF REPEAT", I)
```

In this example, the variable "I" is set to 0. Then the **:REPEAT** statement sets the label to **:LABELOOP** and a user message displays the value of "I". After this, the value of "I" is increased by 1. The expression then checks if "I" is greater than 3. If it is (true) the repeat loop is stopped and the user message "End of Repeat" is shown. If "I" is not greater than 3 (false), then the **:LABELOOP** repeats the loop.

Related Functions

Also see the StopRepeat() - Obsolete function.

:RESUME

Description

Used as part of the Error Handler routine marked by the :ERROR statement. This statement resumes the execution of the action to the line that is after the one which raised the error handled by the error handler .

Parameters

:RESUME;

Input Types: (none)

Example

```
:ERROR;
    UsrMes("Warning!”,” Error 1");
/*** This line resumes the execution of the action with the next line after the one that caused the error;
:RESUME;
```

Related Functions

Also see the :ERROR statement.

:RETURN

Description

Used to stop the current section and return the expression or variable.

Parameters

:RETURN Expr (Optional)

Example

The **:RETURN** command used within an action expression to return an expression or value.

```
:LABEL_START;
    nA := 25 ;
    nB := 150 ;
    nANS := nA + nB ;
:RETURN ANS
```

In this example, the numeric variable "nA" is set to 25 and the numeric variable "nB" is set to 150. We then take the two numeric variables and add them together and store them into a third numeric variable "nANS" Then the **:RETURN** statement take the value of "nANS" and returns the value back to the calling action.

Related Functions

:WHILE

Description

Used to start a loop that will continue until the statement becomes false or an **:**EXITWHILE statement is encountered. It is ended by using the **:**ENDWHILE statement.

Parameters

:WHILE SQL logical Expr

Input Types: (Logical SQL Expr)

Example

```
:WHILE nVAL1 < 4;
    nVal1 := nVal1 + .5;

:ENDWHILE;
```

In the statement above, the statement will be executed only if the value of nVal1 is less than 4. It will continue to loop until the statement becomes false. When it becomes false or if it is false to start with, the action will continue directly after the **:**ENDWHILE statement. *This type of a statement should be used cautiously, because you **must** control the incrementation of the variable used in the logical statement for the WHILE. If not, you will get stuck in an endless loop*. You can also exit the loop by using an :EXITWHILE statement within the loop to exit the loop if certain conditions exist, but you can not branch into this loop from outside the WHILE statement.

Related Functions

See also the :ENDWHILE statement, or the :EXITWHILE statement, or the :LOOP statement.

UDP Functions

EnableRemoteAccess()

Description

Used to enable or disable the remote access on the server (the computer to which you wish to send a script to be executed). The IP and Port on which you want to enable or disable the remote requests are sent in the parameters. This function opens a socket on this computer used for communication with the clients.

Parameters

EnableRemoteAccess(sIP, nPort, nTimeOut, bFlag, bConfirm)

Where:

sIP – The IP address of the computer you wish to use as a server for UDP requests from other users.

nPort – The Port where the server will listen for remote access requests from clients.

nTimeOut – The number of seconds the socket opened on the server will listen for data after a communication has been established.

bFlag – Enables or disables the remote access.

bConfirm – When a user tries to connect to this computer a prompt will appear on the screen so that the user working on that machine will be able to accept or reject the request.

Input Types: (string, numeric, numeric, logic, logic)

Returns: (empty string)

Example

EnableRemoteAccess("111.111.111.111", 7899, 1, .T., .F.);

This statement enables (the flag is true) the remote access on the computer with the IP address: "111.111.111.111" , on the port 7899 and disables the confirmation (last parameter is false).

Related Functions

GetFromUDP()

Description

Used to read whatever the function sent to a remote machine for execution, returns. This function reads from the socket data that was sent back to the caller through a :RETURN statement in the action that was executed remotely.

Parameters

GetFromUDP(oSocket)

Where:

oSocket – Returned by the InitUDPClient() function, the socket is passed as a parameter to identify the source of the UDP requests.

Input Types: (socket object)

Returns: (array)

This function returns an array of 3 elements:

{IP, Port, Data}

IP – The IP of the machine where the function was executed
Port – The Port on the remote machine.
Data – The return of the action that was sent for remote execution.

Example

See the example for the SendUDPRequest() function.

Related Functions

InitUDPClnt()

Description

Creates the local socket to be used in a UDP communication with another computer. On the remote computer the function EnableRemoteAccess() has to be run, enabling the access to that computer's resources.

This function, InitUDPClient() enables the UDP communication on the client side and the function EnableRemoteAccess() enables the communication on the server side.

Parameters

InitUDPClnt(nTimeOut)

Where:

nTmeOut – The number of seconds the socket opened on the local machine will listen for data after a communication has been established.

Input Types: (numeric)

Returns: (socket object)

Example

```
:DECLARE oSocket;

oSocket := InitUDPClnt(2);
```

This statement returns a socket object used to communicate with a remote computer using the UDP functions. This function enables the current computer to communicate to others.

Related Functions

SendUDPRequest()

Description

Used to send a request to the server (the computer to which you wish to send a script to be executed). The IP and Port to which you want to send the remote requests are sent in the parameters. This function uses the socket returned by InitUDPClnt() for communication with the remote computer.

The function returns the length of the action to be executed remotely.

Parameters

SendUDPRequest(oSocket, sIP, nPort, sData, sName, sPassword)

Where:

oSocket – Returned by the InitUDPClient() function, the socket is passed as a parameter to identify the source of the UDP requests.

sIP – The IP address of the computer you wish to use as a server for UDP requests from other users.

nPort – The Port where the server will listen for remote access requests from clients.

The IP address and the port identify the destination of the UDP requests. These are the same parameters used in the EnableRemoteAccess() function call on the server.

sData – The request to be sent. This parameter can contain an entire action that will be executed on the server machine with or without the knowledge of the user that is using the server at that moment.

sName – The username of the user that tries to send the action sData to the other machine for execution.

sPassword – The password of the user.

These parameters (sName, sPassword) are sent to authenticate the user (sender) to verify that he has the right to log in to the remote computer.

Input Types: (object, string, numeric, string, string, string)

Returns: (numeric)

The function returns the length of the string sData or 0 (zero) in case of an error.

Example

```
/* The inline code variable “TEST” will contain the action to be sent
remotely;

:BEGININLINECODE "TEST";
     :DECLARE TmpStr;
     TmpStr:=lConfirm("This action is about to run on your system. Allow it?
","From SYSADM","YESNO","QUESTIONMARK");
      :RETURN TmpStr;
:ENDINLINECODE;

:DECLARE udpsrv,arr,Delay,SendTo;

/*This following enables the communication on the server side. If the
parameters are not passed it is assumed that the function refers to the
current machine ;

EnableRemoteAccess(,,,,.F.);
Delay:=Seconds();

/* The following initializes the client and returns a socket or empty if an error
has occurred;
```

```
 udpsrv:=InitUDPClnt();
SendTo:="111.111.111.125";
:IF Empty(udpsrv);
      :RETURN;
:ENDIF;

/* The following sends the action contained in the TEST variable to the
defined SendTo destination;

 :IF SendUDPRequest(udpsrv, SendTo, 8999, GetInlineCode("TEST"))=0;
      :RETURN "";
:ENDIF;
ProgressEnable(.F.);
:WHILE Empty(Arr) .and. seconds()-delay < 5;
    KeepGoing();
    /* The following gets the return of the action sent to the remote machine.
    Arr:=GetFromUDP(udpsrv);
:ENDWHILE;
/* Disables the remote access;
EnableRemoteAccess(,,,.F., );
ProgressEnable(.T.);
/* Kills the socket object;
endSocket(udpsrv,.T.);
DeleteInlinecode("TEST");
/* returns…;
:IF Len(arr)=3;
      :RETURN arr[3];
:ELSE;
      :RETURN "ERROR";
:ENDIF;
```

Related Functions

Internal Functions

ArgInternal()

Description

Used to pass named arguments to an OLE Server. Should be used in conjunction with the ExecInternal function.

Parameters

ArgInternal(ArgName,ArgValue)

where:

ArgName = Name of the argument.

ArgValue = Value to be passed.

Input Types: (string, any type)

Returns: Named Argument.

Also see the ExecInternal

Example

OLE supports the concept of named arguments. On method invocation, an argument name that is defined by the OLE automation server can be attached to the actual parameter value. This allows you to pass parameters in any order and to omit parameters because the server can uniquely identify the parameters using their names. The NamedArg class is used to model this way of parameter passing

```
:DECLARE o ;
o: = LimsOLEConnect("Word.Basic")
o:FormatFont(ArgInternal(“Points”, 18))
```

Related Functions

CreateInternal()

Description

Used to create an object which inherits the properties of the Visual Object class name specified in the parameters.

Important Note:

A good knowledge of Visual Objects is strongly recommended.

Parameters

CreateInternal(Visual Object Class Name, argument1 … argument6)

Input Types: (string, string)

Returns: The newly created object

Example

```
:DECLARE obj;
obj := CreateInternal("DataWindow",LimsApp("SHELL"));
obj:Show();
Lwait(5);
obj:EndWindow();
:RETURN"";
```

Related Functions

ExecInternal()

Description

Used to execute a method for the specified application (object).

The user can get a list of methods of a certain object using

GetMethList(Object)

Parameters

ExecInternal(Object, Method Name, Param1, ... Param6)

where:

LimsApplication = application reference returned by the LimsApp function.

Method Name = name of method to be executed.

Param1 - Param6 = method's parameters, if needed.

Input Types: (object, string, any type - depending on the method used)

Returns: What the invoked method returns.

Example

```
ExecInternal(LimsApp("SHELL"),"SHELLEXIT")
```

Executes the method FileExit for the Shell Window.

Related Functions

Also see the ArgInternal().

GetInternal()

Description

Used to retrieve a property of an object.

The user can get a list of properties of a certain object using

GetPropList(Object)

Parameters

GetInternal(Object, Property Name)

where:

LimsApplication = application reference returned by the LimsApp function.

Property Name = name of property to be retrieved.

Input Types: (object, string)

Returns: The property

Example

GetInternal(LimsApp("SHELL"),"SIZE"):WIDTH

Retrieves the width of the Shell Window.

The expression GetInternal(LimsApp("SHELL"),"SIZE") returns an object - Dimension object that has 2 properties: WIDTH and HEIGHT. In our example we are retrieving the value of WIDTH.

Related Functions

LimsApp()

Description

Used to return the object associated with a LIMS Window.

Parameters

LimsApp(WindowID)

where:

WindowID = LIMS Window ID or the word SHELL for the Shell Window. Also referred to as 'object'.

Input Types: (string)

Returns: A reference to the application (object).

Example

LimsApp("SHELL")

Returns: reference to the Shell Window (object).

Related Functions

SetInternal()

Description

Used to change a value of a property.

The user can get a list of properties of a certain object using GetPropList(Object)

Parameters

SetInternal(Object, Property Name, value)

where:

Object = application reference returned by the LimsApp function.

Property Name = name of property to be retrieved.

Value = the new value of the property for the LIMS Application.

Input Types: (object, string, string)

Returns: (empty string)

Example

Sets the Audit Trail property to "Y" (Yes) for the TESTS window.

```
SetInternal(LimsApp("TESTS"),"AUDITTRAIL", "Y")
```

Related Functions

Error Handling

GetLastSSLError ()

Description

Returns an Error object with the last SSL error (VO error)

Parameters

Input Types: (none)

Returns: an Error object containing the last SSL error

Example

```
ExecFunction("SystemInit.OnServerStart");

:RETURN 0;

:ERROR;
      error := GetLastSSLError();
      msg := "";
      :IF error != NIL;
            msg := FormatErrorMessage( error );
      :ELSE;
            error := ReturnLastSQLError();
            :IF error != NIL;
                  msg := "DBMS error code: " +
LimsString(error:NativeError) + Chr(13) + Chr(10);
                  msg += "ODBC error code: " +
error:SQLState + Chr(13) + Chr(10);
                  msg += "Description:     " +
error:ErrorMessage + Chr(13) + Chr(10);
            :ENDIF;
      :ENDIF;

      usrmes( "Error executing
Runtime_Support.OnServerStart", msg );
      :RETURN 1;
```

Try to execute the function OnServerStart. If no error occurs then 0 is returned.

If any error occurs then try to retrieve the last SSL error. If GetLastSSLError returns an Error object, then you can format the error message. Otherwise, see if the error was SQL generated and compose a different message. In the end, if there was an error, return 1.

Related Functions

ReturnLastSqlError ()

Description

Returns a SQLErrorInfo object with the last SQL error

Parameters

Input Types: (none)

Returns: an SQLErrorInfo object containing the last SQL error

Example

```
ExecFunction("SystemInit.OnServerStart");

:RETURN 0;

:ERROR;
      error := GetLastSSLError();
      msg := "";
      :IF error != NIL;
            msg := FormatErrorMessage( error );
      :ELSE;
            error := ReturnLastSQLError();
            :IF error != NIL;
                  msg := "DBMS error code: " +
LimsString(error:NativeError) + Chr(13) + Chr(10);
                  msg += "ODBC error code: " +
error:SQLState + Chr(13) + Chr(10);
                  msg += "Description:     " +
error:ErrorMessage + Chr(13) + Chr(10);
            :ENDIF;
      :ENDIF;

      usrmes( "Error executing
Runtime_Support.OnServerStart", msg );
      :RETURN 1;
```

Try to execute the function OnServerStart. If no error occurs then 0 is returned.

If any error occurs then try to retrieve the last SSL error. If GetLastSSLError returns an Error object, then you can format the error message.

Otherwise, see if the error was SQL generated and compose a different message. In the end, if there was an error, return 1.

Related Functions

ReturnLastSqlErrorStatement()

Description

Returns the SQL statement that generated the last SQL error

Parameters

Input Types: (none)

Returns: an SQL statement string that generated the last SQL error

Example

```
ExecFunction("SystemInit.OnServerStart");

:RETURN 0;

:ERROR;
      error := GetLastSSLError();
      msg := "";
      :IF error != NIL;
            msg := FormatErrorMessage( error );
      :ELSE;
            error := ReturnLastSQLError();
            :IF error != NIL;
                  sqlStatement :=
ReturnLastSQLErrorStatement();
                  msg := "DBMS error code: " +
LimsString(error:NativeError) + Chr(13) + Chr(10);
                  msg += "ODBC error code: " +
error:SQLState + Chr(13) + Chr(10);
                  msg += "Description:     " +
error:ErrorMessage + Chr(13) + Chr(10);
                  msg += "Statement:     " +
sqlStatement + Chr(13) + Chr(10);

            :ENDIF;
      :ENDIF;

      usrmes( "Error executing
Runtime_Support.OnServerStart", msg );
      :RETURN 1;
```

Try to execute the function OnServerStart. If no error occurs then 0 is returned.

If any error occurs then try to retrieve the last SSL error. If GetLastSSLError returns an Error object, then you can format the error message.

Otherwise, see if the error was SQL generated and compose a different message, containing the last SQL statement that generated the last error.

In the end, if there was an error, return 1.

Related Functions

FormatErrorMessage ()

Description

Returns a formatted string with information about error

Parameters

FormatErrorMessage(oError)

Input Types: (oError)

oError – an VO Error object

Returns: a formatted string with information about the error

Example

```
ExecFunction("SystemInit.OnServerStart");

:RETURN 0;

:ERROR;
      error := GetLastSSLError();
      msg := "";
      :IF error != NIL;
            msg := FormatErrorMessage( error );
      :ELSE;
            error := ReturnLastSQLError();
            :IF error != NIL;
                  sqlStatement :=
ReturnLastSQLErrorStatement();
                  msg := "DBMS error code: " +
LimsString(error:NativeError) + Chr(13) + Chr(10);
                  msg += "ODBC error code: " +
error:SQLState + Chr(13) + Chr(10);
                  msg += "Description:     " +
error:ErrorMessage + Chr(13) + Chr(10);
                  msg += "Statement:     " +
sqlStatement + Chr(13) + Chr(10);

            :ENDIF;
      :ENDIF;

      usrmes( "Error executing
Runtime_Support.OnServerStart", msg );
      :RETURN 1;
```

Try to execute the function OnServerStart. If no error occurs then 0 is returned.

If any error occurs then try to retrieve the last SSL error. If GetLastSSLError returns an Error object, then you can format the error message.

Otherwise, see if the error was SQL generated and compose a different message, containing the last SQL statement that generated the last error.

In the end, if there was an error, return 1.

Related Functions

RaiseError()

Description

Creates and throws an Error object

Parameters

RaiseError(strDescription, strFunction, nErrorCode)

Input Types: (string, string, numeric)

strDescription – error description -> saved in Error:Description

strFunction – any meaningful description for the location of error; saved in Error:Operation

nErrorCode – the error's code, saved in Error:GenCode.

The user should use error codes > 10000.

Example

…

```
errDataAccessError := {20102, "DataAccessError",
"Data Access Error"};
…

bDBResponse :=
RunSQL(sSQL,"DICTIONARY",{sNewCategoryID,Upper(categ
oryName),categoryName,categoryDescription,applicatio
nFlag});

      oErr := ReturnLastSQLError();

      :IF !bDBResponse;
            sNewCategoryID :=  "";
            :IF .not. Empty(oErr);
                  RaiseError(oErr:ErrorMessage,
errDataAccessError[2], errDataAccessError[1]);
            :ENDIF;

      :ENDIF;
```

Try to run a SQL statement. If it fails, the ReturnLastSQLError function will return a valid object, and you can programmatically raise an error that will be handled later on, in the calling code.

The RaiseError function can be used in any other context.

Related Functions

ShowSQLErrors ()

Description

Enables (show=TRUE)/disables the RaiseError when executing sql statements.

Implicit, all sql operations do RaiseError if an error happened while executing the sql statement. By doing this, all errors go to client application.

But there are situations when the developer wants to handle the error. In this case he/she can use :ERROR command or ShowSQLErrors function.

If error mechanism is disabled than the developer has to check the result returned by the SQL functions and to decide what to do next.

In case of error he/she can use ReturnLastSqlError to get the SQLErrorInfo object that encapsulates the sql error.

Parameters

ShowSQLErrors(show)

Input Types: (boolean)

show – enables (show=TRUE)/disables the RaiseError when executing sql statements

Example

Related Functions

Web Specific Functions

AddToApplication()

Description

Adds a pair {name, value} into the Application array. If the pair exists, then the value for the corresponding name is updated.

Parameters

AddToApplication(varName, varValue)

Input Types: (string, object)

Returns: none

Example

```
/*---------------------------------------------------
  Registers a new session for the specified user in
the "STARLIMSUsers" application variable
---------------------------------------------------;
:PROCEDURE RegisterSession;
:PARAMETERS strUserName;
      :DECLARE strLoggedUsers;

      strLoggedUsers :=
       AllTrim(GetFromApplication("STARLIMSUsers"));

      :IF Len(strLoggedUsers) > 0;
            strLoggedUsers := strLoggedUsers + ",";
      :ENDIF;

      strLoggedUsers:= strLoggedUsers + strUserName;
      AddToApplication("STARLIMSUsers",
                       strLoggedUsers);

:ENDPROC;
```

Related Functions

AddToSession()

Description

Adds a pair {name, value} into the session array. If the pair exists, then the value for the corresponding name is updated.

Parameters

AddToSession(varName, varValue)

Input Types: (string, object)

Returns: none

Example

```
/* Gets the current user's role identifier;
:PROCEDURE GetUserRole;
:DECLARE userRole;

userRole := GetFromSession( "MYUSERROLE" );

:IF Empty(userRole);
      userRole := lSearch( "SELECT treeauth from
       users where USRNAM = ?", "", "DATABASE",
      {MYUSERNAME} );

      :IF Empty(userRole);
            RaiseError("User does not exist or no
role defined for user!");
      :ENDIF;

      AddToSession( "MYUSERROLE", userRole );
:ENDIF;

:RETURN userRole;

:ENDPROC;
```

Related Functions

ClearApplication()

Description

Clears the Application array and sets its size to 0.

Parameters

ClearApplication()

Input Types: (none)

Returns: none

Example

```
ClearApplication();
```

Related Functions

ClearSession()

Description

Clears the Session array and sets its size to 0.

Parameters

ClearSession()

Input Types: (none)

Returns: none

Example

```
ClearSession();
```

Related Functions

GetFromApplication()

Description

Retrieves a value from the Application using the name key.

Parameters

GetFromApplication(varName)

Input Types: (string)

Returns: value for corresponding name, as it was stored in the Application array.

Example

```
/*-----------------------------------------------------
  Registers a new session for the specified user in
the "STARLIMSUsers" application variable
-----------------------------------------------------;
:PROCEDURE RegisterSession;
:PARAMETERS strUserName;
      :DECLARE strLoggedUsers;

      strLoggedUsers :=
       AllTrim(GetFromApplication("STARLIMSUsers"));

      :IF Len(strLoggedUsers) > 0;
            strLoggedUsers := strLoggedUsers + ",";
      :ENDIF;

      strLoggedUsers:= strLoggedUsers + strUserName;
      AddToApplication("STARLIMSUsers",
                       strLoggedUsers);

:ENDPROC;
```

Related Functions

GetFromSession()

Description

Retrieves a value from the session using the name key.

Parameters

GetFromSession(varName)

Input Types: (string)

Returns: value for corresponding name, as it was stored in the Session array.

Example

```
STARLIMSDEPT := GetFromSession("STARLIMSDEPT");
STARLIMSSITECODE := GetFromSession("STARLIMSSITECODE");
```

Related Functions

RemoveFromApplication()

Description

Deletes an item from Application array if found by name.

Parameters

RemoveFromApplication(varName)

Input Types: (string)

Returns: none

Example

```
RemoveFromApplication("STARLIMSUsers");
```

Related Functions

RemoveFromSession()

Description

Deletes an item from Session array if found by name.

Parameters

RemoveFromSession(varName)

Input Types: (string)

Returns: none

Example

```
RemoveFromSession("STARLIMSDEPT");
```

Related Functions

Appendix A: Rounding Rules

EPA - Rounding Rules

Given a number with r digits to the left of the decimal and d+t digits in the fraction part, with d being the place to which the number is to be rounded and t being the remaining digits which will be truncated, this number is rounded to r+d digits by adding 5 to the (r+d+1)th digit when the number is positive or by subtracting 5 when the number is negative. The t digits are then truncated at the (r+d+1)th digit. The symbol for a rounded number truncated to zero is (*).

Example:

StdRound("EPA",3,Val(Input)));

Number (Input)	Expected Result
0	0.00
0.0	0.00
0.00	0.00
0.000	0.00
0.01243	0.0124
0.01257	0.0126
0.01299	0.0130
0.03999	0.0400
0.09999	0.100
0.001234	0.00123
0.001235	0.00124
0.001299	0.00130
0.003999	0.00400
0.009999	0.0100
0.0004551	0.000455
0.0004555	0.000456
0.0001299	0.000130
0.0001999	0.000200
.1	0.100
.01	0.0100

.001	0.00100
.0001	0.000100
.00001	0.0000100
.000001	0.00000100
.0001999	0.000200
.00019	0.000190
.0019	0.00190
.4	0.400
.40	0.400
.400	0.400
.45	0.450
.455	0.455
.4555	0.456
.4565	0.457
.4999	0.500
.9	0.900
.90	0.900
.91	0.910
.915	0.915
.900	0.900
.901	0.901
.910	0.910
.911	0.911
.9001	0.900
.9005	0.901
.0990	0.0990
.99	0.990
.990	0.990
.9900	0.990

.99000	0.990
.9995	1.00
.9999	1.00
9.943	9.94
9.945	9.95
9.949	9.95
9.955	9.96
9.999	10.0
0.99	0.990
0.955	0.955
0.99425	0.994
0.99451	0.995
0.999	0.999
0.9995	1.00
0.9999	1.00
1	1.00
1.0	1.00
1.00	1.00
1.000	1.00
1.0000	1.00
1.231	1.23
1.235	1.24
1.239	1.24
99.43	99.4
99.45	99.5
99.46	99.5
99.91	99.9
99.95	100
99.99	100

455.45	455
455.55	456
999.55	1000
999.1	999
9931000	9930000
9935000	9940000
9937000	9940000
9961254	9960000
9965569	9970000
9969782	9970000
9999236	10000000

FDA - Rounding Rules

Table 1. FDA Rounding Rules

When The First Digit Dropped is:	The Last Digit Retained is:	Examples
Less than 5	Unchanged	2.44 to 2.4 2.429 to 2.4
More than 5, or 5 followed by at least 1 digit other than 0	Increased by 1	2.46 to 2.5 2.51 to 2.5
5 followed by zeros	Unchanged if Even, or Increased by 1 if Odd	2.450 to 2.4 2.550 to 2.6

(a) When the first digit discarded is less than five, the last digit retained should not be changed. For example, if the quantity 984.3 is to be declared to three significant digits, the figure 3 to the right of the decimal point must be discarded since it is less than 5 and the last digit to be retained (the figure "4") will remain unchanged. The rounded number will read 984. The same rationale applies to numbers declared to two significant digits (for example 68.4 and 7.34); again the final digit is dropped and the last digit retained remains unchanged so that the "rounded-off" numbers become 68 and 7.3 respectively.

(b) When the first digit to be discarded is greater than five, or it is a five followed by at least one digit other than zero, the last digit to be retained should be increased by one unit.

Examples:

984.7 becomes 985

984.51 becomes 985

6.86 becomes 6.9

6.88 becomes 6.9

(c) When the first digit to be discarded is exactly five, followed only by zeros, the final digit to be retained should be rounded up if it is an odd number (1,3,5,7, or 9), but no adjustment should be made if it is an even number (2,4,6, or 8).

Examples:

984.50 becomes 984

985.50 becomes 986

68.50 becomes 68

7.450 becomes 7.4

7.550 becomes 7.6

ISO - Rounding Rules

Table 2. ISO Rounding Rules

When The First Digit Dropped is:	The Last Digit Retained is:	Examples
Less than 5	Unchanged	2.44 to 2.4 2.429 to 2.4
More than 5	Increased by 1	2.46 to 2.5 2.51 to 2.5
5 followed by zeros	Increased by 1	2.450 to 2.5 2.550 to 2.6

(a) When the first digit discarded is less than five, the last digit retained should not be changed. For example, if the quantity 984.3 is to be declared to three significant digits, the figure 3 to the right of the decimal point must be discarded since it is less than 5 and the last digit to be retained (the figure "4") will remain unchanged. The rounded number will read 984. The same rationale applies

to numbers declared to two significant digits (for example 68.4 and 7.34); again the final digit is dropped and the last digit retained remains unchanged so that the "rounded-off" numbers become 68 and 7.3 respectively.

(b) When the first digit to be discarded is greater than five, or it is a five followed by the zero, the last digit to be retained should be increased by one unit.

Examples:

984.7 becomes 985

984.51 becomes 985

6.86 becomes 6.9

6.88 becomes 6.9

984.50 becomes 985

985.50 becomes 986

68.50 becomes 69

7.450 becomes 7.5

7.550 becomes 7.6

Index