LibreOffice Basic Primitives

Author: Jean-François Nifenecker
jean-francois.nifenecker@laposte.net

Version: 0ae

Date: 2018-06-17

Summary

Presentation

This document comes with a bunch of subprogram "primitives" aimed at filling some programming gaps in LibreOffice Basic. These primitives cover much of LibreOffice functionality and are meant to be extended by any contributor over time. They complement the usual `Tools` library made available under LibreOffice packages since the OOo years.

I aim at "correcting" some of the drawbacks that the `Tools` library suffers: would we see them under a result point of view or a thematic theme, the `Tools` library mixes different primitives (its `Strings` module has some maths functions); too few are documented and, when they are, it's quite poorly; the development standards aren't clear (there's no pedagogic incentive towards the naming of items, code structure and the like); some of these primitives call UI items (ex. the `MsgBox()` function) which imply they can't be used "as-is" in a non-UI environment; error handling isn't standardized and often non effective; test tools are not available.

In other words, LibO Basic users need reinventing the wheel each and every time they have a new set of tools to develop.

Covered areas

The primitive subprograms proposed here are thematically grouped. The current edition covers the following areas:

- Operating System information,
- IO helpers,
- Logging tool,
- Strings management,
- Arrays management,
- Mathematical tools,
- LibreOffice application suite tools,
- Calc module management,
- Writer module tools,
- Forms helpers,
- Email sending tool.

LibreOffice version and other software

The macros are being developed using LibreOffice v.4.3.x and above. They passed their tests under a few operating systems (read on) but they have **not** been tested under lower versions nor with ApacheOpenOffice (AOO). For instance, it is known that dates management has changed starting with LibO version 4.1.1. and differ from AOO.

Vocabulary

FQN	Fully qualified name. Applies to files and folders names. More information here: https://en.wikipedia.org/wiki/Fully_qualified_name#Filenames_and_paths

Primitives Programming Guidelines

In order to fix some of the problems cited above, an effort has been put into the application of a set of guidelines to all the primitives proposed here:

- no user interaction (ex. calling the `MsgBox()` function for error reporting) within the subprograms, unless a module is specifically developed for UI use (ex. folder and file pickers),
- error handling using common error codes (global constants), or by returning specific error values,
- full documentation as a header of all subprograms (input, output, restrictions, usage examples),
- all modules come with corresponding test modules and subs,
- as much as possible, setting the return value of a function only takes place once,
- when necessary, ancillary subprogram names are prefixed using an underscore character "_". These subprograms are reserved for internal use and therefore should not be called from the outside,
- uniform naming conventions are applied,
- uniform structure for all modules.

No User Interaction

For obvious reasons, most of the primitives never visually communicate with the user. This means that such visual functions like `MsgBox()` are forbidden within the primitives. This allows to use the primitives for non-visual uses, the visual part and error managing burdens being transferred to the caller.

Note that some parts *may* use GUI when appropriate, as this is the case for the folder and file pickers. This is clearly documented anytime.

Error Handling

For a better inter-process communication, the primitives generally handle errors by use of error codes. Most of the time, these are stored within global constants named as `ERR_xxx`, as specified in the details of the libraries below. It is up to the caller to make appropriate use of the return values.

The `ERR_xxx` constants are declared in the modules where they are used. This means that none is available if the module is not used.

These constants naming follows the `ERR_<library>_<name>` mask,
ex: `ERR_IOPRIM_NOSUCHFOLDER`

Full Documentation

Modules

Subprograms

Naming Conventions

Modules Structure

Modules structure should be consistent. Here's my take on this, which I'm applying everywhere (See the actual code as an example).

1. A header

   The header is a set of comment lines which introduce the code and give basic information. A header should always specify at least the following:

   1. The module name and summarized purpose.
   2. The dependencies with other modules or libraries.
   3. The author name.
   4. The module version and date.
   5. The license under which the code is released.

2. Execution options

   In LibreOffice Basic, any execution option must appear first.

3. Declares

   This is a very old habit of mine, from my Pascal days, that declares always come before the code that uses the identifiers. They are listed in the following order:

   1. Custom types.
   2. Constants

      Globals, then Public, then private to the module.

   3. Variables

      Globals, then Public, then private to the module.

4. Sub-programs

   In alphabetical order (_ prefixes are ignored for ordering).

5. A footer

   A single End-of-file comment line.

   No line of text is allowed past the footer.

Mode details below.

Dependencies

I've tried to avoid dependencies between modules as much as possible. At times, though, in order to minimize code duplication, some dependencies may be present. These are specified in the modules headers and in the following descriptive pages.

In other situations, though, you'll find internal subprograms, noted with a _ (underscore) prefix, that may replicate code found elsewhere. I've adopted this format to avoid the said dependencies.

Subprograms structure

Subs and Functions follow this structure:

1. Declaration
2. Comments
    1. A quick overview.
    2. Input parameters details.
    3. Output value (functions only).
    4. Various details (optional).
3. Declares

    Constants and variable declares, strictly separated from the subprogram body.

    This is a heritage from my Pascal days, where all declares must be done at the top of the subprogram. Anyway, I find this way of doing to be much clearer than some of the intricacies one can see sometimes in Basic code.
4. Subprogram body
5. Function exit (functions only)

    I always set the function exit at a single (final) place to ensure readability. Only on very exceptional occasions, where code would be overly complicated, a function may have more than one exit point or not have its exit point at the end.
6. Subprogram end

    Shown using a comment mentioning the subprogram name as a reminder.

Example (function `ArrayExists()` in `ArrayPrim.Arrays`)

```
Function ArrayExists(ByRef pArray As Variant) As Boolean
'Checks whether an array exists (is dimensioned) and can be manipulated.
'Input:
'-- pArray: the array to check
'Output: True if pArray is a valid array variable otherwise False

   Dim l_Exists As Boolean

   'is the variable defined?
   l_Exists = Not IsNull(pArray)
   If l_Exists Then
      'is the variable an array variable?
      l_Exists = IsArray(pArray)
      If l_Exists Then
         'is the array initialized?
         l_Exists = (UBound(pArray) >= LBound(pArray))
      End If
   End If

   ArrayExists = l_Exists
End Function 'ArrayExists
```

Test Modules

All modules and subprograms come with a test routine. The test routines are stored in the same module as the routines they test. The test routines are named from the tested routines names. For instance, to test `Function GetSomeResult()` there is a `Sub _GetSomeResult_test()`. As the primitives are listed alphabetically, this means that both the primitive routine and its test sub come together in the code.

Note that classes are not tested that way. (WIP)

In case of changes within the primitives, the corresponding tests must be carried again.

Examples

Ideally, any piece of code should come with a real-life example. Unfortunately, such is not currently the case here. When time permits, I add code samples that show how to use some of the primitives together in order to achieve some common goal.

Installation (version 0ae

(Does nothing – WIP!)

To install the primitives libraries in the MyMacros container,
just click the **Install now!** button.
This button can also update previous installed versions, if any.

Install now!

Alternatively, you may manually import the libraries, like this:

1. **Tools > Macros > Manage macros > LibreOffice Basic**.
2. In the new **LibreOffice Basic Macros** window, click the **Manage** button.
3. A new **LibreOffice Basic Macro manager** window comes up.
4. Go to the **Libraries** tab and click the **Import** button.
5.

Using the Primitives

Subs and Functions

Classes

Very few primitives are provided under the form of classes. This is the case for ini file and log file management or for mailing facilities.

(from the defunct CereusAPIs)

Classes can just be instantiated using one the three ways shown below:

```
 1   Sub Main
 2
 3     '1 - Using the Dim statement
 4     Dim obj as new clsBase
 5     ’MsgBox "I’ll popup before class initialization"
 6
 7     '2 - Using Set
 8     Dim obj as Object
 9     Set obj = new clsBase
10     'MsgBox "I’ll popup after class initialization"
11
12     '3 - Using createObject (according to documentation only on Window systems)
13     Dim obj as Object
14     Set obj = CreateObject("clsBase")
15     'MsgBox "I’ll popup before class initialization"
16
17   End Sub
```

There is a slight difference between the second approach and the first and third approaches. Following the second approach (`Set`) the class instance is initialized after assigning it to the `obj` variable. In the other two approaches the class is initialized as soon as a field, property or method is referenced/called.

Author & License

Author

The first version of these primitives collection is

© 2016-2018 – Jean-François Nifenecker, jean-francois.nifenecker@laposte.net.

Most of the macros are from the author above. Nevertheless, some may come from different sources, such as forums or mailing lists. When it is the case, the original author is credited in these lines and in the code. Here's a list of the cited authors:

(list)

CereusAPIs	(website seems to have vanished by now)
Heertsch	

If you are one of the authors and do not want your work to be collected in this set, just let me know so that I remove the corresponding macros from the set.

License

The primitives

The following pages list the global constants, global variables, subs and functions that are part of the primitives set.

One chapter is devoted to each library, the modules being listed in sub-chapters. While constants and variables are listed in apparent disorder, the primitives are listed in alphabetical order.

OSPrim

Operating system primitives.

OSPrim.OS

Dependencies: (none)

Constants

Constant name	Value	Description
OSPRIM_OSWIN	`Windows`	
OSPRIM_OSLINUX	`Linux`	
OSPRIM_OSMAC	`MacOS`	
OSPRIM_OSOSX	`OSX`	

Primitives

GetEnvString()

TBD

GetOSName()

```
Function GetOSName() As String
```

Returns the OS name (see `OSPRIM_OSxxx` constants above).
If the OS is not one of the three known ones, returns an empty string.
Adapted from Heertsch (https://forum.openoffice.org/en/forum/viewtopic.php?f=45&t=26 280&p=176 535&hilit=operating#p137 780)

IsLinux()

```
Function IsLinux() As Boolean
```

Returns `True` if the underlying operating system is Linux, otherwise `False`.

IsOSX()

```
Function IsOSX() As Boolean
```

Returns `True` if the underlying operating system is OS X, otherwise `False`.

IsWindows()

```
Function IsWindows() As Boolean
```

Returns `True` if the underlying operating system is Windows, otherwise `False`.

SetEnvString()

TBD

ErrorPrim

Error handling library.

An effort is being put in all primitives in order to provide error checking every time it is possible or useful. While many functions returns error values (`-1`, zero-length string, `False`, `Null` and so on), some functions and some subs just set an error value. It's then up to the caller to test for these errors values.

Whenever error values are set, you'll find the corresponding constants listed at the top of each chapter and at the top of each code module. The following syntax convention was adopted: the constant name mask is `ERR_<LIBNAME>_<ERRORTYPE>`.

For instance, you'll find `ERR_IOPRIM_NOSUCHFOLDER`.

TBD

IOPrim

This library offers folder and file access modules.

Note that internally, because LibreOffice is multi-platform, files, folders and paths names are converted to the URL format. This format should always be preferred and the OS native format used only when communicating with the user.

IOPrim.Globals

Dependencies: (none)

Global Constants

Constant name	Value	Description
Error Constants (longs)		
ERR_IOPRIM_NOERR	0	No error.
ERR_IOPRIM	1000	Base value for all files and folders error results. The other error codes are added to ERR_IOPRIM value.
ERR_IOPRIM_NOSUCHFOLDER	ERR_IOPRIM + 1	Folder not found.
ERR_IOPRIM_FOLDERDELETE	ERR_IOPRIM + 2	Folder deletion error.
ERR_IOPRIM_FOLDERMOVE	ERR_IOPRIM + 3	Folder move error.
ERR_IOPRIM_FOLDERCOPY	ERR_IOPRIM + 4	Folder copy error.
ERR_IOPRIM_FOLDEREXISTS	ERR_IOPRIM + 5	The target folder already exists.
ERR_IOPRIM_FOLDERLIST	ERR_IOPRIM + 6	Folder contents listing error.
ERR_IOPRIM_FOLDERTOSELF	ERR_IOPRIM + 7	Can't copy to self.
ERR_IOPRIM_NOSUCHFILE	ERR_IOPRIM + 101	File not found.
ERR_IOPRIM_UNKNOWNMODE	ERR_IOPRIM + 102	File opening mode is unknown.
ERR_IOPRIM_FILEEXISTS	ERR_IOPRIM + 103	File already exists.
ERR_IOPRIM_FILETOSELF	ERR_IOPRIM + 104	Can't copy file to self.
ERR_IOPRIM_FILEREADONLY	ERR_IOPRIM + 105	Can't set file R/O.
ERR_IOPRIM_FILEHIDE	ERR_IOPRIM + 106	Can't hide file.
ERR_IOPRIM_CANTCLOSETEXT	ERR_IOPRIM + 201	Text file can't be closed.
ERR_IOPRIM_NOSPACE	ERR_IOPRIM + 301	No space left on drive.
ERR_IOPRIM_INISECUNKNOWN	ERR_IOPRIM + 401	Ini Section not found.
ERR_IOPRIM_INIKEYUNKNOWN	ERR_IOPRIM + 402	Ini Key not found.
File Access Mode Constants (longs)		
IOPRIM_READMODE	1	Open a file for reading.
IOPRIM_WRITEMODE	2	Open a file for writing.
IOPRIM_APPENDMODE	8	Open a file for appending (read and write).
File Locking Modes		(long)
IOPRIM_NOLOCK	0	No file locking.
IOPRIM_LOCKREAD	1	Lock a file in read mode.
IOPRIM_LOCKWRITE	2	Lock a file in write mode.
IOPRIM_LOCKREADWRITE	4	Lock a file in read and write modes.
Folder contents filtering		
IOPRIM_FOLDERFILTER_ALL	0	(no filtering)
IOPRIM_FOLDERFILTER_FILESONLY	1	Return files only.
IOPRIM_FOLDERFILTER_FOLDERSONLY	2	Return folders only.

Constant name	Value	Description
Other constants		
IOPRIM_PATHSEPCHAR	/	(string) Path separator character in URLs.
IOPRIM_EXTSEPCHAR	.	(string) Extension separator character.

IOPrim.Folders

Dependencies: ArrayPrim.Arrays, IOPrim.Files

Primitives

`CheckPathStr()`

```
Function CheckPathStr(ByRef pPath As String) As String
```

Checks that a path string terminates with a `/` (slash).

Input

- `pPath`: the path string to be checked.

Output

The validated path string, with a `/` (slash) as last character.

`CopyFolder()`

```
Function CopyFolder(ByRef pSourceFolder As String, pTargetFolder As
String) As Long
```

Copies a folder into another.

Input

- `pSourceFolder`: the source folder.
- `pTargetFolder`: the target folder.

If the target folder exists, the source folder is copied within the existent; if it doesn't exist, it is created.

Note: `pSourcefolder` and `pTargetFolder` may be either in URL or OS form.

Output

Returns `0` if the folder was copied, or the copy error:

- `ERR_IOPRIM_NOSUCHFOLDER:` source folder not found.
- `ERR_IOPRIM_NOSPACE`: can't create target folder.
- `ERR_IOPRIM_COPYTOSELF`: can't copy to same folder.
- `ERR_IOPRIM_FOLDERCOPY`: can't copy folder.

`CreateFolder()`

```
Function CreateFolder(ByRef pFolderName As String) As Long
```

Creates a folder.

Input

- `pFolderName`: the FQN of the folder to create (in URL or OS form).

Output

Returns `0` if the folder was created, or the creation error:

- `ERR_IOPRIM_FOLDEREXISTS`: the folder already exists.
- `ERR_IOPRIM_NOSPACE`: no space left on drive.

`DeleteFolder()`

```
Function DeleteFolder(ByRef pFolderName As String) As Long
```

Deletes a folder.

Input

- `pFolderName`: the FQN of the folder to delete (in URL or OS form).

Output

Returns `0` if the folder was deleted, or the deletion error:

- `ERR_IOPRIM_NOSUCHFOLDER`: folder not found.
- `ERR_IOPRIM_FOLDERDELETE`: can't delete folder.

FolderExists()

```
Function FolderExists(ByRef pFolderName As String) As Boolean
```

Checks if a folder exists.

Input

- `pFolderName`: the FQN of the folder to check (in URL or OS form).

Output

Returns `True` if the folder exists, otherwise `False`.

FolderIsEmpty()

```
Function FolderIsEmpty(ByRef pFolderName As String) As Boolean
```

Checks if a folder is empty.

Input

- `pFolderName`: the FQN of the folder to check (in URL or OS form).

Output

Returns `True` if the folder is empty, otherwise `False`.

GetFolderContents()

```
Function GetFolderContents(ByRef pFolderName As String, pFilter As
Byte, pContents As Variant) As Long
```

Returns a folder contents.

Input

- `pFolderName`: the FQN of the folder to be scanned (in URL or OS form).
- `pFilter`: the output filter (all/files only/folders only) (See the `IOPRIM_FOLDERFILTER_XXX` constants).
- `pContents`: the one-dimension array to be filled with the file and folder names (strings).
  The file names are FQN in URL form.

Output

Returns `0` if the folder contents was listed, or the access error:

- `ERR_IOPRIM_NOSUCHFOLDER`: folder not found.
- `ERR_IOPRIM_FOLDERLIST`: can't list folder contents.

GetParentFolder()

```
Function GetParentFolder(ByRef pFolderName As String) As String
```

Returns the parent of a given folder.

Input

- `pFolderName`: the FQN of the folder to process (in URL or OS form).

Output

The name of the parent of `pFolderName` or a zero-length string if an error occurred.

Note: this function does **not** check for the child path existence.

IsFolder()

```
Function IsFolder(ByRef pName As String) As Boolean
```

Checks whether an item name is a folder one.

Input

- `pName`: the item name to check.

Output

Returns `True` if `pName` is a folder name (in URL or OS form), otherwise `False`.

IsSubFolder()

```
Function IsSubFolder(ByRef pParentName As String, pChildName As String)
As Boolean
```

Checks whether an item name is a folder one.

Input

- `pParentName`: the parent item name to check (in URL or OS form).
- `pChildName`: the child item name to check (in URL or OS form).

Output

Returns `True` if `pChildName` is a child of `pParentName`, otherwise `False`.

Note: This function does **not** check if either folder actually exists.

IOPrim.Files

Dependencies: StringsPrim.Strings

Primitives

`ChangeFileExt()`

```
Function ChangeFileExt(ByRef pFileName As String, pNewExt As String)
As String
```

Changes the extension part of a *file* name.

Input

- `pFileName`: the file name to be processed (in URL or OS form).
- `pNewExt`: the new extension, without leading dot.

Output

The new file name with the changed extension (in URL form). If any, the path part is stripped out.

Note: This does **not** check for the file/path actual existence.

Usage

```
MyFile = "C:\Path\To\SomeFile.ods"
Result =  ChangeFileExt(MyFile, "bak")
```

returns "file:///C:/Path/To/SomeFile.bak"

Depends on

StringsPrim.Strings

`ExtractFileExt()`

```
Function ExtractFileExt(ByRef pFileName As String) As String
```

Retrieves the extension part of a file name.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

The file extension, without leading dot.

Note: This function does **not** check for the file/path existence.

Usage

```
FileExt = ExtractFileExt("C:\Path\To\MyFile.odt")
```

returns `odt`

`ExtractFileName()`

```
Function ExtractFileName(ByRef pFileName As String) As String
```

Retrieves a filename without access path from a FQN file name.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

The file name without the path part.

Note: this does **not** check for the file/path existence.

See also `JustFileName()`.

Usage

```
FileName = ExtractFileName("C:\Path\To\MyFile.odt")
```

returns `MyFile.odt`

ExtractFilePath()

```
Function ExtractFilePath(ByRef pFileName As String) As String
```

Extracts the path from a given FQN file name.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

The path part of `pFileName`, in URL form and terminated with a `/` (slash).

Note: this does **not** check for the file/path existence.

Usage

```
Path = ExtractFilePath("C:\Path\To\MyFile.odt")
```

returns `"file:///C:/Path/To/"`

GetFileContents()

```
Function GetFileContents(ByRef pFileName As String) As String
```

(from `Tools.GetRealFileContent`)
Returns the content type of a document.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

Returns the content type of a document by extracting the content from the header of the document or an empty string if the content couldn't be determined.
Returns:

- odt: `writer8`
- ods: `calc8`
- odp: `impress8`
- odg: `draw8`
- odb: `StarBase`
- txt: `writer_T602_Document`
- rtf: `writer_Rich_Text_Format`
- pdf: `pdf_Portable_Document_Format`
- zip: `writer_FictionBook_2`

etc.

GetFileDateTimeModified()

```
Function GetFileDateTimeModified(ByRef pFileName As String) As Date
```

Returns a file date of last modification.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

Returns the date, or `0` if the operation couldn't be executed.

GetFileSize()

```
Function GetFileSize(ByRef pFileName As String) As Long
```

Returns a file name size in bytes.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

Returns the file size in bytes or `-1` if an error was encountered.

GetSafeDateTimeStr()

```
Function GetSafeDateTimeStr(ByRef pDate As Date, Optional pWithTime
As Boolean) As String
```

Returns a string representation of a date for use as part of a valid filename.

Input

- `pDate`: the date to be used.
- `pWithTime`: (optional) specifies if the result string should have a time part or not.
  Defaults to `True`.

Output

The string representation of the date/time, following the ISO masks:

- `YYYYMMDD_HHmmSS` (date and time),
- `YYYYMMDD` (date only; `pWithTime` is `False`).

IsHidden()

```
Function IsHidden(ByRef pFileName As String) As Boolean
```

Checks whether a file is hidden or not.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

Returns `True` if `pFilename` is hidden, otherwise `False`.
If the file doesn't exist, returns `True`.

IsReadOnly()

```
Function IsReadOnly(ByRef pFileName As String) As Boolean
```

Checks whether a file is read-only or not.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

Returns `True` if `pFilename` is read-only, otherwise `False`.
If the file doesn't exist, returns `True`.

JustFileName()

```
Function JustFileName(ByRef pFileName As String) As String
```

Retrieves a filename without its access path nor its extension from a FQN file name.

Input

- `pFileName`: the file name to be processed (in URL or OS form).

Output

The file name without the path and extension parts.

Note: this does **not** check for the file/path existence.

See also `ExtractFileName()`.

Usage

```
JustFile = JustFileName("C:\Path\To\MyFile.odt")
```

returns `MyFile`

NTFSFileNameString()

```
Function NTFSFileNameString(ByRef pFileName As String, Optional
ByRef pExtended As Boolean) As String
```

Returns a NTFS compliant file name by stripping out some potentially problematic characters or names and replacing others with more secure chars.

Input

- `pFileName`: the original file name.
  Caution: The file name must be passed *without* the path part.
- `pExtended`: (optional, defaults to `False`) implies
  – a more extended char set to be stripped out of the file name and
  – accented chars are replaced with their non-accented counterparts.

Output

The transformed file name, if necessary.

Details about the file naming checks:

- non-printable chars (Ascii < 32 decimal) are stripped out,
- spaces are replaced with '_' (underscores),
- accented chars are replaced with their unaccented counterparts,
- some characters are stripped out (see the `NTFS_CHARSTRIP` constant),
- optionally some more stripping is done for the additional `NTFS_CHARSTRIPEX` characters.
- if the file name starts with a system-reserved one (see the `NTFS_CONFLICT` const), an '_' (underscore) is prepended to the file name in order to avoid any system conflict (see https://en.wikipedia.org/wiki/Filename).

SetHidden()

```
Function SetHidden(ByRef pFileName As String, pHide As Boolean) As
Long
```

Sets a file as hidden or not.

Input

- `pFileName`: the FQN of the file to process (in URL or OS form).
- `pHide`: the hiding flag (`True` to hide, `False` to show).

Output

Returns `0` if the operation was executed, or an error code:

- `ERR_IOPRIM_NOSUCHFILE`: file not found.
- `ERR_IOPRIM_FILEHIDE`: can't set the hidden flag.

SetReadOnly()

```
Function SetReadOnly(ByRef pFileName As String, pRO As Boolean) As
Long
```

Sets a file as read-only or not.

Input

- `pFileName`: the FQN of the file to process (in URL or OS form).
- `pRO`: the read-only flag (`True` to set, `False` to unset).

Output

Returns `0` if the operation was executed, or an error code:

- `ERR_IOPRIM_NOSUCHFILE`: file not found,
- `ERR_IOPRIM_FILEREADONLY`: can't set read-only flag.

IOPrim.TextFiles

Dependencies: (none)

This module's subprograms allow for text files control using the standard LibreOffice Basic tools.

As streams were introduced lately, another set of primitives has been developed for the same purpose in another module (see `IOPrim.Streams`)

Primitives

`CloseTextFile()`

```
Function CloseTextFile(ByRef pFileH As Integer) As Long
```

Closes a given file text handle.
Returns `0` if the file handle was closed, otherwise an error value:
- `ERR_IOPRIM_CANTCLOSETEXT`: Can't close.

`LoadTextFileAsString()`

```
Function LoadTextFileAsString(ByRef pFileName As String, pLineSep As
String) As String
```

Reads a text file and returns it as a string.
The source text file is opened, read then closed by this function.

Input
- `pFileName`: the file name (FQN, in URL or OS mode)
- `pLineSep`: the line separators to add after each line read (except the last one).

Output
The text file as a string.
Note: line separators are part of the returned string.

See also
ReadTextAsString()

`LoadTextFileAsVector()`

```
Function LoadTextFileAsVector(ByRef pFileName As String) As Variant
```

Reads a text file and returns it as a 1D array (vector) of strings.
The source text file is opened, read then closed by this function.

Input
- `pFileName`: the file name (FQN, in URL or OS mode)

Output
The text file as a string vector.

See also
ReadTextAsVector()

`OpenTextFile()`

```
Function OpenTextFile(ByRef pFileName As String, Optional pMode As
Long) As Integer
```

Opens a text file named `pFileName` under a specified mode.

Input
- `pFileName`: the file name to open.
- `pMode`: (optional) one of the following values:
  - `IOPRIM_READMODE (1)`: Read.
  - `IOPRIM_WRITEMODE (2)`: Write.
    (An existing file is overwritten)
  - `IOPRIM_APPENDMODE (8)`: Read/Write.

  `pMode` defaults to `IOPRIM_READMODE`.

Output
Returns the file handle (integer value, must be `> 0`). If an error occurred, returns `0`.

`ReadTextAsString()`

```
Function ReadTextAsString(ByRef pHFile As Integer, pLineSep As String)
As String
```

Reads an opened text file and returns its contents as a string.
The source text file is not closed when the function returns.

Input

- `pHFile`: the text file handler.
- `pLineSep`: the line separators to add after each line read (except the last one).

Output

The text file as a string.

Note: line separators are part of the returned string.

See also

LoadTextFileAsString()

`ReadTextAsVector()`

```
Function ReadTextAsVector(ByRef pHFile As Integer) As Variant
```

Reads an opened text file and returns its contents as a 1D array (vector).
The source text file is not closed when the function returns.

Input

- `pHFile`: the text file handler

Output

The text file as a string vector.

See also

LoadTextFileAsVector()

`ReadTextLine()`

```
Function ReadTextLine() As String
```

Reads a line of text from a text file, at the current cursor location.

`StoreFromArray()`

```
Function StoreFromArray() As Long
```

Stores all the strings within a one-dimension array into a text file.

`StoreText()`

```
Function StoreText() As Long
```

Stores the string into a text file.

`WriteTextLine()`

```
Function WriteTextLine() As Long
```

Writes a line of text within a text file, at the current cursor location.

IOPrim.Streams

(work-in-progress)

Dependencies: (none)

The primitives in this module access to files using the API `SimpleFileAccess` service (aka streams). When multi-platform development is in order, they should be preferred to the classic primitives in `IOPrim.TextFiles`.

For a demonstration of these routines use, see the ini-file management (`Ini` module). (not yet done)

Stream encoding methods: the stream read/write functions may set the source or target text encoding method. If not explicitly set, the encoding is in UTF-8 by default. The encoding is set using a description string (ex: `"UTF-8"`). The complete list of available encoding descriptors can be found here:

https://www.iana.org/assignments/character-sets/character-sets.xhtml (`name` column).

Globals

Constant name	Value	Description
`IOPRIM_SERV_SFA`	`"com.sun.star.ucb.SimpleFileAccess"`	(string) SimpleFileAccess service name.

Primitives

`CloseStream()`

```
Function CloseStream(ByRef pStream As Object) As Long
```

Closes a stream.

Input

- `pStream`: the stream to close.

Output

Returns `0` if the operation was executed, or an error code:

- `pStream` is `Null`.
- `pStream` doesn't support any stream service.
- Can't close the stream ().

`LoadTextStreamAsString()`

```
Function LoadTextStreamAsString(ByRef pFileName As String, pEncoding
As String) As String
```

Reads a text file and returns its contents as a string.
This function opens, reads then closes the source text file.

Input

- `pFileName`: the file name (FQN, in URL or OS mode)
- `pEncoding`: the text file contents encoding, as a string (defaults to `UTF-8` when not specified).

Output

The text contents as a string.

Note: line separators (Ascii 10 and/or Ascii 13) are part of the returned string.

`LoadTextStreamAsVector()`

```
Function LoadTextStreamAsVector(ByRef pFileName As String, pEncoding
As String) As Variant
```

Reads a text file and returns its contents as a vector (1D array).
This function opens, reads then closes the source text file.

Input

- `pFileName`: the file name (FQN, in URL or OS mode)
- `pEncoding`: the text file contents encoding, as a string (defaults to `UTF-8` when not specified).

Output

The text contents as a 1D array.

`OpenStream()`

```
Function OpenStream(ByRef pFileName As String, pMode As Long,
Optional pEncoding As String) As Object
```

Opens a `pFileName` (text stream) using `pMode`. An encoding method may be specified.

Input

- `pFileName`: the input file name (in URL or OS mode)
- `pMode`: the file opening mode.
  `pMode` values are:
  - `1`: Read (`IOPRIM_READMODE`)
  - `2`: Write (`IOPRIM_WRITEMODE`)
  - `8`: Append (Read/Write) (`IOPRIM_APPENDMODE`)
- `pEncoding`: (optional) The stream encoding method.
  Defaults to UTF-8.

Output

Returns the stream object if the operation was executed, or `NULL` if an error was encountered.

`ReadTextFromStream()`

```
Function ReadTextFromStream(ByRef pStream As Object, pString As
String) As Long
```

`StoreToStream()`

```
Function StoreToStream(ByRef pFileName As String, pStr As String) As
Long
```

Writes a string into a text file.

Input

- `pFileName`: the target file name (FQN, in URL or OS form).
- `pStr`: the string to write.

Output

Returns `0` if the operation was executed, or an error code:
- can't write to stream (),
- stream is `NULL` ().

WriteTextToStream()

```
Function WriteTextToStream(ByRef pStream As Object, pStr As String)
As Long
```

Writes a string to a stream.

Input

- `pStream`: the target stream object.
- `pStr`: the string to write.

Output

Returns `0` if the operation was executed, or an error code:

- can't write to stream (),
- stream is `NULL` ().

IOPrim.Ini

(work-in-progress)

Dependencies: (none)

This module offers ini file handling (based upon `IOPrim.TextFile` primitives above).

Primitives

`DeleteIniKey()`

```
Function DeleteIniKey(ByRef pFileName As String, pSection As String,
pKey As String) As Long
```

Deletes key from a given section in an ini file.

Input

- `pFileName`: the target ini file name (FQN, in URL or OS form).
- `pSection`: the section name, with or without the enclosing brackets.
- `pKey`: the key name.

Output

Returns `0` if successfully completed, otherwise an error code:
- File not found ().
- Section not found ().
- Key not found ().
- File can't be written ().

`DeleteIniSection()`

```
Function DeleteIniSection(ByRef pFileName As String, pSection As
String) As Long
```

Deletes a `pSection` for the ini file `pFilename`.
The `pSection` name is passed without bracketting.
Returns `0` if successfully completed, otherwise an error code:
- File not found (),
- Section not found ().

`IniKeyExists()`

```
Function IniKeyExists(ByRef pFileName As String, pSection As String,
pKey As String) As Boolean
```

Returns `True` if `pKey` exists within `pSection` of `pFilename` ini file.
If either file, section or key don't exist, returns `False`.

`IniSectionExists()`

```
Function IniSectionExists(ByRef pFileName As String, pSection As
String) As Boolean
```

Returns `True` if `pSection` exists within `pFilename` ini file.
If either file, or section don't exist, returns `False`.

`ReadIniKey()`

```
Function ReadIniKey(ByRef pFileName As String, pSection As String, pKey
As String, pValue As String) As Long
```

Reads a value (`pValue`) read in `pFilename`, for `pSection` and `pKey`.
Returns `0` if successfully completed, otherwise an error code:
- File not found (),
- Section not found (),
- Key not found ().

IniSectionStr()	`Function IniSectionStr(ByRef pSName As String) As String` Returns a section name for ini file writing or reading, that is with surrounding square brackets. Note that spaces are left untouched, except starting and ending ones which are stripped off the input string.
WriteIniKey()	`Function WriteIniKey(ByRef pFileName As String, pSection As String, pKey As String, pValue As String) As Long` Writes a value (`pValue`) to `pFilename`, for `pSection` and `pKey`. If `pFileName` doesn't exist, it is created. If `pSection` doesn't exist, the section is created. If `pKey` exists, `pValue` replaces the previous value, otherwise the key is created. Returns `0` if successfully completed, otherwise an error code: • Can't create file (), • File can't be written ().

IOPrim.Log

Dependencies: (none)

The subprograms in this module handle writing to log files.

A log file is a text file which holds lines of messages. A log message/line structure follows this scheme:

```
20160325 153412 [ERR ] File not found
^^^^^^^^ ^^^^^^ ^^^^^^ ^^^.......
date     time   type   message...
(iso)
```

You may log three message types: error messages `[ERR ]`, information messages `[INFO]` and uncategorized messages (see the `IOPRIM_LOGUNK`, `IOPRIM_LOGERROR` and `IOPRIM_LOGINFO` constants). Basically, logging a message is done using either `LogIt()`, `LogError()` or `LogInfo()` functions.

Logging strategy:

0. Define the logging context (file name and overwrite mode) (`SetLogging()`).
1. Enable logging (`EnableLogging()`) to start the logging process.
2. Log events by choosing the event type (error or information, or nothing) (`LogIt(), LogError()` or `LogInfo()`)

   Note that you don't have to explicitly open the log file as it is automatically opened whenever necessary.
3. End logging (`CloseLog()`), which actually closes the log file.

Notes:

- Logging may be suspended and resumed at any time, using `DisableLogging()` and `EnableLogging()`.
- If you want to log again after explicitly closing the log file, just re-enable the logging (`EnableLogging()`) before logging new messages.
- This module has been made as independent as possible from any other in the set.
- Error management is possible through the `ERR_IOPRIM_LOGxxx` constants.

Constants

Constant name	Value	Description
General		
IOPRIM_LOGUNK	0	
IOPRIM_LOGERROR	1	
IOPRIM_LOGINFO	2	
IOPRIM_LOGERRORSTR	0	Error flag (notice the space as 4th char) for log lines.
IOPRIM_LOGINFOSTR	0	Information flag for log lines.
Error codes		
ERR_IOPRIMLOGNONE	0	No error.

Constant name	Value	Description
ERR_IOPRIM_LOGCANTCLOSE	1	Can't close the log file.
ERR_IOPRIM_LOGSUSPENDED	2	Logging suspended, can't log.
ERR_IOPRIM_LOGFILECLOSED	3	Log file is closed.
ERR_IOPRIM_LOGCANTWRITE	4	Can't write to log file.
ERR_IOPRIM_LOGCANTOPEN	5	Can't open log file.
ERR_IOPRIM_LOGSET	6	Undefined log file settings.

Primitives

CloseLog()

```
Function CloseLog() As Long
```

Closes the log file.
Returns `0` if successful, otherwise an error code:
- `ERR_IOPRIM_LOGCANTCLOSE`: Can't close.

DisableLogging()

```
Sub DisableLogging()
```

Suspends the logging.

EnableLogging()

```
Sub EnableLogging()
```

Starts logging.

LogError()

```
Function LogError(ByRef pMsg As String) As Long
```

Logs an error message
Acts as a facade to `LogIt()`

LogInfo()

```
Function LogInfo(ByRef pMsg As String) As Long
```

Logs an information message
Acts as a facade to `LogIt()`

LogIt()

```
Function LogIt(ByRef pMsg As String, Optional pType As Integer) As Long
```

Writes `pMsg` to the log file, with an optional `pType` flag.
`pType` defaults to `IOPRIM_LOGUNK (0)`.
The values for `pType` are integers in (see above):
- `0 (IOPRIM_LOGUNK)`
- `1 (IOPRIM_LOGERROR)`
- `2 (IOPRIM_LOGINFO)`

All output to the log file is timestamped.
The output line looks like:

```
20160325 153412 [ERR ] File not found
```

The log file is written only if the logging is running.
Returns `0` if successful, otherwise an error code:
- `ERR_IOPRIM_LOGSTOPPED`: Logging is stopped.
- `ERR_IOPRIM_LOGFILECLOSED`: Log file is closed.
- `ERR_IOPRIM_LOGCANTWRITE`: Can't write to log file.

`OpenLog()`	`Function OpenLog() As Long` Opens a log file (path and name in global vars). If `g_OverWrite` is `True`, the log file replaces any previous log file with the same name, otherwise the log is appended to the existing log file. Returns `0` if successfully opened, otherwise an error code: • `ERR_IOPRIM_LOGSET`: Undefined settings. • `ERR_IOPRIM_LOGCANTOPEN`: Can't open the log file.
`SetLogging()`	`Sub SetLogging(ByRef pLogFileName As String, Optional pOvw As Boolean)` Sets the logging context: file name and overwrite mode.

IOPrim.ZipClass

Dependencies: (none)

This class manages zip files (compressed files): zip container creation, adding, deleting, suppressing or extracting files or folders. It can be used to explore LibreOffice documents.

Directories

Zip containers can store directories. Accessing a directory is straightforward: just specify its full name wherever required. The root directory name usually is `""` (null-length string) and subdirectories are separated with the `"/"` (slash) character.

Note that the root directory name can also be written as `"/"` (slash). Thus, both `/Path/To/MyFile.odt` and `Path/To/MyFile.odt` are valid and point to the same file (`MyFile.odt`).

Error Management

Errors are reported internally by the `ZipClass`. The last error code can be accessed using the `ErrorStatus` read-only property. The `ResetError()` method resets the error status to the default.

The available error values are described through the `ERR_ZIP_XXX` values below.

Constants

Constant name	Value	Description
Error Constants		
ERR_ZIP_NONE	0	No error (default error value)
ERR_ZIP_PACKAGEINIT	1	The zip package container couldn't be initialized.
ERR_ZIP_NOSUCHZIP	2	The zip file doesn't exist.
ERR_ZIP_NOSUCHFILE	3	The specified file cannot be found on the device.
ERR_ZIP_DIREXISTSINZIP	4	The specified directory already exists in the zip container. This can possibly not be an error.
ERR_ZIP_NOSUCHFILEINZIP	5	The specified file cannot be found in the zip directory.
ERR_ZIP_NOSUCHDIRINZIP	6	The directory doesn't exist within the zip container.
ERR_ZIP_NOTAZIP	7	The container is not a zip.
ERR_ZIP_NOTACHILDDIR	8	The specified subdirectory is not a child of the parent.
ERR_ZIP_NOPARENTDIR	9	The specified directory is not a parent of the subdir.
ERR_ZIP_NOSUBDIR	10	The specified directory has no subdirs.
ERR_ZIP_NONEMPTYDIR	11	The directory is not empty, thus can't be deleted.

Properties

ErrorStatus	`Integer [RO]`	Returns the current error status. See `ERR_ZIP_XXX` constants above.
IsInitialized	`Boolean [RO]`	Returns `True` if the zip container is initialized, `False` otherwise.
ZipContainer	`Object [RO]`	Returns the internal zip container object.

Methods

Public Methods

AddDirectory()

```
Public Sub AddDirectory(ByRef pZipDir As String)
```

Adds a directory within the zip container.

Input

- `pZipDir`: the directory name to add to the container.

Note: the root directory is `""` (null-length string). Subsequent directories are separated using `"/"`.

AddFile()

```
Public Sub AddFile(ByRef pFileURL As String, Optional ByRef pZipDir As
String)
```

Adds a file to an existing zip container.

Input

- `pFileURL`: the file name (URL form) to add to the package
  If the file is present already, it is replaced.
- `pZipDir`: (optional) The storage directory within the zip container.
  Defaults to `""` (root).
  If the directory doesn't exist, it is created.

Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHFILE`: `pFileURL` file not found.

AddTree()

```
Public Sub AddTree(ByRef pTreeURL As String, pParentURL As String)
```

(TBD)

CloseZip()

```
Public Sub CloseZip()
```

Closes the zip file.
This method resets the error status value and initialization flag.

DeleteDirectory()

```
Public Sub DeleteDirectory(ByRef pDirName As String, pParentURL As
String)
```

Deletes a directory and its contents from the zip container.

Input

- `pDirName`: the full directory path within the container
  ex: `Path/To/SomeDir/`
  Root dir cannot be deleted.
- `pParentURL`: the parent directory of the one to delete.

Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHDIRINZIP`: directory not found in zip.
- `ERR_ZIP_NOTACHILDDIR`: the specified subdirectory is not a child of the parent.
- `ERR_ZIP_NOPARENTDIR`: the specified directory is not a parent of the subdir.
- `ERR_ZIP_NOSUBDIR`: the specified directory has no subdirs.

Notes:

- The process stops if an error occurs at some stage.
- Directories are recursively deleted bottom-up until the parent directory is reached.

DeleteFile()

```
Public Sub DeleteFile(ByRef pFileName As String)
```

Deletes a file from the zip container.

Input

- `pFileName`: the full path to the file in the container
  ex: `Path/To/SomeFile.odt`

Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHFILEINZIP`: file not found in zip.

DirectoryContents()

```
Public Function DirectoryContents(ByRef pDirURL As String) As Variant
```

Returns a directory contents as an array of strings.

Input

- `pDirURL`: the directory name in URL form

Output

An array of strings.
Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHDIRINZIP`: directory not found in zip.

DirectoryExistsInZip()

```
Public Function DirectoryExistsInZip(ByRef pPathURL As String) As
Boolean
```

Checks whether a directory exists within the container.

Input

- `pPathURL`: the path to check.
  Ex: `/Path/To/SomeSubdir`

Output

`True` if the directory exists otherwise `False`.
Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHDIRINZIP`: directory not found in zip.

ExtractAll()

```
Public Sub ExtractAll(ByRef pTargetDirURL As String, Optional
pIgnoreEmpty As Boolean)
```

Extracts all the contents of a container to some target device.

Input

- `pTargetDirURL`: the target device (in URL form) where to extract directories and files.
  If the target doesn't exist, it is created if possible.
- `pIgnoreEmpty`: (optional) empty directories are not recreated.
  Defaults to `True`

Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHZIP`: zip file not found.
- `ERR_ZIP_PACKAGEINIT`: zip container initialization problem.

ExtractFile()

```
Public Sub ExtractFile(ByRef pFileName As String, pTargetURL As String)
```

Extracts a file from the zip container.

Input

- `pFileName`: the full path to the file in the container
  Eg: `Path/To/SomeFile.odt`
- `pTargetURL`: the target file name, in URL form.

Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHFILEINZIP`: file not found in zip.

ExtractTree()

```
Public Sub ExtractTree(ByRef pTreeURL As String, pTargetDirURL As
String)
```

Extracts a container directory tree and its contents to some directory on a device.

Input

- `pTreeURL`: the internal tree to extract, in URL form (all dirs and subdirs).
- `pTargetDirURL`: the target device (in URL form) where to extract directories and files.
  If the target doesn't exist, it is created if possible.
- `pIgnoreEmpty`: (optional) empty directories are not recreated.
  Defaults to `True`

Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHZIP`: zip file not found.
- `ERR_ZIP_PACKAGEINIT`: zip container initialization problem.
- `ERR_ZIP_NOSUCHDIRINZIP`: directory not found in zip.

FileExistsInZip()

```
Public Function FileExistsInZip(ByRef pPathURL As String) As Boolean
```

Checks whether a filename exists within the container.

Input

- `pPathURL`: the file path to check, in URL form.
  Ex: `/Path/To/SomeFile.odt`

Output

`True` if the file exists otherwise `False`.
Possible error results:

- `ERR_ZIP_NONE`: everything went OK.
- `ERR_ZIP_NOSUCHDIRINZIP`: directory not found in zip.

FileStream()	`Public Function FileStream(ByRef pFileName As String) As Object` Reads a file stream from the zip container and returns it. **Input** • `pFileName`: the full path to the file in the container (FQN) Ex: `Path/To/SomeFile.odt` **Output** The file stream read or `Null` if an error occurred. Possible error results: • `ERR_ZIP_NONE`: everything went OK • `ERR_ZIP_NOSUCHFILEINZIP`: file not found in zip
OpenZip()	`Public Sub OpenZip(ByRef pZipURL As String)` Opens a zip package, either existing or new. Opening a non-existing zip container creates it. Opening an existing one updates it. **Input** • `pZipURL`: the zip file name in URL form Possible error results: • `ERR_ZIP_NONE`: everything went OK. • `ERR_ZIP_PACKAGEINIT`: zip container initialization problem.
ResetErrorStatus()	`Public Sub ResetErrorStatus()` Resets the error value to its default (`ERR_ZIP_NONE`).

Usage

See the `ZipClass_Test` module.

IniPrim

(under development – not working yet)

IniPrim.Globals

Dependencies: (none)

All constants have a `INIPRIM` or `INICLASS` prefix. As always, error constants are `ERR_` prefixed.

Constant name	Value	Description
Error Constants		
`ERR_INICLASS_NOTHINGTODO`	`-2`	Nothing to do.
`ERR_INICLASS_EMPTYVALUE`	`-1`	The wanted value is empty (the key exists). This is not really an “error”.
`ERR_INICLASS_NONE`	`0`	No error.
`ERR_INICLASS_CANTCREATEFILE`	`1`	Impossible to create or update the ini file.
`ERR_INICLASS_CANTOPENFILE`	`2`	The file can’t be opened.
`ERR_INICLASS_FILENOTFOUND`	`3`	The ini file specified was not found.
`ERR_INICLASS_CANTWRITEFILE`	`4`	The ini file is set as read-only. Can’t write to the file.
`ERR_INICLASS_SECTIONNOTFOUND`	`11`	The specified section is not defined in the ini file.
`ERR_INICLASS_KEYNOTFOUND`	`12`	The specified key is not defined in the section.

IniPrim.IniStreamClass

(not yet available)

Dependencies: StringsPrim.Strings

This module defines an Ini file management class. Ini files can be managed either in memory or on-disk.

Properties

BackupExtension	`String [R/W]`	The ini file backup extension. Defaults to ".bak"
CacheUpdates	`Boolean [R/W]`	Defines the way the ini file is handled: • if `CacheUpdates` is `True`, the ini file is managed in memory. The file is written on disk on object destruction or on a call to the `UpdateFile()` method. • if `CacheUpdates` is False, then every change in the ini file is immediately written to disk. Defaults to `False`: all changes are immediately sent to device.
FileName	`String [R/W]`	Sets or gets the ini file name (FQN). Note: the file name **must** specify the extension which is not set by the class.
LastError	`Long [RO]`	Returns the last error number. See the error constants list above for more information.
RemChars	`String [R/W]`	Sets and reads the comment chars values. By default `RemChars` is set to # only.
Stream	`Object [RO]`	Returns the stream associated with the ini file.

Methods

CloseFile()

```
Sub CloseFile()
```

Closes the ini file.
If `CacheUpdates` is `True` then the file is updated at that moment.

DeleteKey()

```
Function DeleteKey(ByRef pSection As String, pKey As String) As Long
```

Deletes a key from a section in the ini file.

Input

- `pSection`: the section name from which to delete the key.
  The `pSection` name is passed without bracketing.
- `pKey`: the key name to delete.

Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- File not found (`ERR_INICLASS_FILENOTFOUND`),
- Section not found (`ERR_INICLASS_SECTIONNOTFOUND`),
- Key not found (`ERR_INICLASS_KEYNOTFOUND`).

DeleteSection()

```
Function DeleteSection(ByRef pSection As String) As Long
```

Deletes a section from the ini file.

Input

- `pSection`: the section name from which to delete the key.
  The `pSection` name is passed without bracketing.

Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- File not found (`ERR_INICLASS_FILENOTFOUND`),
- Section not found (`ERR_INICLASS_SECTIONNOTFOUND`).

KeyExists()

```
Function KeyExists(ByRef pSection As String, pKey As String) As Boolean
```

Returns `True` if `pKey` exists within `pSection` of the ini file.
The `pSection` name is passed without bracketing.
If either file, section or key don't exist, returns `False`.

Init()

```
Function Init(ByRef pFileName As String, Optional pCacheUpdates As Boolean) As Long
```

This method initializes both `FileName` and `CacheUpdates` properties at once, then loads the specified ini file if it exists (see `Load()` method).
`CacheUpdates` defaults to `False`.
Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- File not found (`ERR_INICLASS_FILENOTFOUND`),
- Can't open file (`ERR_INICLASS_CANTOPENFILE`)

Load()

```
Function Load() As Long
```

Loads the ini file.
This method resets all ini class internals. When `CachedUpdates` is `True`, this allows to abandon all changes since the previous call to `Load()`.
Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- File not found (`ERR_INICLASS_FILENOTFOUND`),
- Can't open file (`ERR_INICLASS_CANTOPENFILE`)

ReadKey()

```
Function ReadKey(ByRef pSection As String, pKey As String, pValue As String) As Long
```

Reads a value (in `pValue`) for `pSection` and `pKey`.
The `pSection` name is passed without bracketing.
Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- File not found (`ERR_INICLASS_FILENOTFOUND`),
- Can't open file (`ERR_INICLASS_CANTOPENFILE`),
- Section not found (`ERR_INICLASS_SECTIONNOTFOUND`),
- Key not found (`ERR_INICLASS_KEYNOTFOUND`).

SectionExists()

```
Function SectionExists(ByRef pSection As String) As Long
```

Returns `ERR_INICLASS_NONE` if `pSection` exists within the ini file.
If either file or section don't exist, returns an error code:

SectionKeys()

Returns an array of all key names present within a section in the ini file.

Sections()

Returns an array of all section names present within the ini file.

SectionValues()

Returns an array of all key-value pairs in a section of the ini file.

UpdateFile()

```
Function UpdateFile() As Long
```

Updates the file on disk.
This method has to be called to force file update when the property `CacheUpdates` is True.
When `CacheUpdates` is `False`, a call to `UpdateFile()` does nothing.
Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- Nothing to do (`ERR_INICLASS_NOTHINGTODO`)
- Can't create file (`ERR_INICLASS_CANTCREATEFILE`).

WriteKey()

```
Function WriteKey(ByRef pSection As String, pKey As String, pValue As
String) As Long
```

Writes a value (`pValue`) for `pKey` in `pSection`.
If the file doesn't exist, then it is created.
If `pSection` doesn't exist, the section is created.
If `pKey` exists, `pValue` replaces the previous value, otherwise the key is created.
Returns `ERR_INICLASS_NONE` if successfully completed, otherwise an error code:

- Can't create file (`ERR_INICLASS_CANTCREATEFILE`).

LogPrim

This library offers logging means through a dedicated class.

LogPrim.LogTextClass

Dependencies: (none)

This module defines a log file management class using standard text file management. Log messages are immediately written to disk.

Constants

All constants have a `LOGCLASS` prefix. Moreover, error constants are `ERR_` prefixed.

Constant name	Value	Description
Error Constants		
`ERR_LOGCLASS_NONE`	`0`	No error.
`ERR_LOGCLASS_CANTCREATEFILE`	`1`	Impossible to create or update the ini file.
`ERR_LOGCLASS_CANTOPENFILE`	`2`	The file can't be opened.
`ERR_LOGCLASS_FILENOTFOUND`	`3`	The ini file specified was not found.
`ERR_LOGCLASS_CANTWRITEFILE`	`4`	The ini file is set as read-only. Can't write to the file.
Other constants		

Properties

`Active`	Boolean [R/W]	`Active` determines whether the log mechanism is active. If set to `True`, the object opens the log file if needed. Any attempt to log a message while the log is not active will try to set this property to `True`. Closing the log file is done by setting the `Active` property to `False`. If the log file cannot be opened, this property results in an `ERR_LOGCLASS_CANTOPENFILE` error (see the `LastError` property).
`DefaultType`	String [R/W]	Defines the default logging type, among the ones in the `LOGCLASS_LOGxxx` constants (see above).
`Filename`	String [R/W]	`FileName` is the name of the log file used to log messages. If none is specified, then the name of the document is used, with the extension replaced by `.log`. The file is then located in the `/tmp` directory on unix-like systems, or in the application directory for Dos/Windows systems.
`LastError`	Long [RO]	Returns the last error number. See log error codes above. The `ClearError` method resets the internal flag to `ERR_LOGCLASS_NONE`.
`LogMask`	String [R/W]	Sets the log lines mask. Available mask codes are: • `%E%`: Event string (`"Error"`, `"Info"`, `"Warning"` or `"Debug"`) • `%M%`: Message part (no formatting) • `%T%`: Timestamp (formatted using the `TimeStampFormat` property). Defaults to ISO: `yyyymmdd hhnnss`. Defaults to `%T% [%E%] %M%` Example message: `20160801 012345 [Info] Opening Myfile.odt`

Paused	Boolean [RO]	`Paused` indicates whether the sending of messages is temporarily suspended or not. See `PauseLog()` and `ResumeLog()` methods.
TimeStampFormat	String [R/W]	Defines the timestamp format. Defaults to ISO (`yyyymmdd hhnnss`). The format should be compatible with the `Format()` function in LibreOffice Basic.

Methods

Public Methods	
ClearError()	`ClearError()` Clears the internal error flag and resets it to `ERR_LOGCLASS_NONE`
Log()	`Log()` Logs a message. The optional `pType` argument is used as the message event type. `pType` defaults to `"info"`.
LogDebug()	`LogDebug()` Logs a debug event message.
LogError()	`LogError()` Logs an error event message.
LogInfo()	`LogInfo()` Logs an information event message.
LogWarning()	`LogWarning()` Logs a warning event message.
PauseLog()	`PauseLog()` Temporarily suspends the sending of log messages. Subsequent calls to `log()` will simply eat the log message and return as if the message was sent. The sending can be resumed by calling `Resume()` The `Paused` property (R/O) returns the pause status.
ResumeLog()	`ResumeLog()` Resumes the sending of log messages if sending was paused through `Pause()`. The `Paused` property (R/O) returns the pause status.

LogPrim.LogStreamClass

Dependencies: (none)

This class is the same as LogTextclass, except it uses streams as a text file output. The overall methods and properties are identical.

(under development)

StringsPrim

StringsPrim.Strings

Dependencies: (none)

Global Constants

Constant name	Value	Description
STRPRIM_SGLQUOTECHAR	'	Single quote char.
STRPRIM_DBLQUOTECHAR	"	Double quote char.

Primitives

DelChar()

```
Function DelChar(ByRef pStr As String, pChar As String) As String
```

Strips a character out of a string and returns the result.

Input

- `pStr`: the string to process.
- `pChar`: the character to be stripped out of `pStr` (1 char only)

Output

The processed string, without the `pChar` character.

Usage

```
DelChar("LibreOffice", "e")
```

returns `"LibrOffic"`

DelSpaces()

```
Function DelSpaces(ByRef pStr As String, Optional pUnBreak As Boolean)
As String
```

Deletes all spacing characters in a string and returns the result.

Input

- `pStr`: the string to process.
- `pUnbreak`: (optional) specifies if the unbreakable space must be processed or not. Defaults to `False`.

Spaces can be:

- `Chr(32)`: ordinary space
- `Chr(160)`: unbreakable space

Output

The processed string, without the spacing characters.
If `pUnBreak` is set to `True` (defaults to `False`), unbreakable spaces (`Chr(160)`) are deleted as well as ordinary spaces (`Chr(32)`).

Usage

```
DelSpaces(" LibreOffice is the best office suite")
```

returns `"LibreOfficeisthebestofficesuite"`

FilterNonPrintableStr()

```
Function FilterNonPrintableStr(ByRef pStr As String) As String
```

Strips-out any non-printable char from the original string.

Input

- `pStr`: the string to process

Output

The input string without non-printable chars (Ascii code < 32 decimal)

`LeftPad()`

```
Function LeftPad(ByRef pStr As String, pPadChar As String, pLength As
Long) As String
```

Pads a string on the left to a given length using a padding character and returns the resulting string.

Input

- `pStr`: the string to process
- `pPadChar`: the character used for padding
- `pLength`: the output string length

Output

The result string.
If `pLength` is less than `pStr` actual length, the resulting string is *shortened* accordingly on the left.
See `RightPad()`.

Usage

`LeftPad("LibreOffice", "-", 15)`
returns `"----LibreOffice"`

`NoAccentStr()`

```
Function NoAccentStr(ByRef pStr As String) As String
```

Replaces accented chars with not accented ones.

Input

- `pStr`: the string to process.

Output

The input string with accented chars replaced with their unaccented counterparts (see the two constants in the code).

Usage

`NoAccentStr("énumération")`
returns `"enumeration"`

`QuoteStr()`

```
Function QuoteStr(ByRef pStr As String, pQuoteChar As String) As String
```

Adds quotes to a string and returns the quoted string.

Input

- `pStr`: the string to process.
- `pQuoteChar`: the quoting character.

Output

Returns the quoted string using `pQuoteChar` to both ends of `pStr`.
If `pStr` already has such quotation marks **at both ends**, `pStr` is left untouched.
See `UnQuoteStr()`.

Usage

`QuoteStr("My Text", Chr(29))` ⟶ `"'My Text'"`
`QuoteStr("'My Text", Chr(29))` ⟶ `"''My Text'"`
`QuoteStr("'My Text'", Chr(29))` ⟶ `"'My Text'"`

ReplaceStr()

```
Function ReplaceStr(ByRef pStr As String, pSearchStr As String,
pReplStr As String) As String
```

Replaces a string portion in a given string.

Input
- `pStr`: the string to process.
- `pSearchStr`: the searched string.
- `pReplStr`: the replacement string.

Output
The modified input string.
Usage
`ReplaceStr("LibreOffice is an office suite", "an", "the best")`
returns `"LibreOffice is the best office suite"`

RightPad()

```
Function RightPad(ByRef pStr As String, pPadChar As String, pLength As
Long) As String
```

Pads a string on the right to a given length using a padding character and returns the resulting string.

Input
- `pStr`: the string to process
- `pPadChar`: the character used for padding
- `pLength`: the output string length

Output
The result string.
If `pLength` is less than `pStr` actual length, the resulting string is *shortened* accordingly on the right.
See `LeftPad()`.
Usage
`RightPad("1234", 7, "x")` ⟶ `"1234xxx"`
`RightPad("1234", 3, "x")` ⟶ `"123"`

StripChars()

```
Function StripChars(ByRef pStr As String, pStripChars As String) As
String
```

Strips a given set of characters from a string.

Input
- `pStr`: the string to process.
- `pStripChars`: the chars to strip from `pStr`.

Output
The stripped string.
Usage
`MyString = StripChars(MyString, "+-*/")`
strips out numeric operators from `MyString`

SuppressMultipleChars()

```
Function SuppressMultipleChars(ByRef pStr As String, pChar As String)
As String
```

Suppresses multiple occurrences of a given character in a string, leaving one only, and returns the result.

Input
- `pStr`: the string to process.
- `pChar`: the char to filter (one character only).

Output
The string, stripped-out of extraneous characters. Multiple `pChar` characters are suppressed, leaving one only.

`TitleCase()`

```
Function TitleCase(ByRef pStr As String) As String
```

Converts a string into Title Case.

Input

- `pStr`: the string to process.

Output

The processed string.

Usage

```
Str = "test the o'connors and the mac-addamses"
TitleCase(Str) ⟶ "Test The O'Connors And The Mac-Addamses"
```

`TrimEx()`

```
Function TrimEx(ByRef pStr As String) As String
```

Enhanced version of the Basic `Trim()` function that suppresses surrounding spaces from a given string. This function suppresses both standard spaces (Ascii `032`) and **unbreakable spaces** (Ascii `160`).

Input

- `pStr`: The string to process.

Output

The process result.

`UnQuoteStr()`

```
Function UnQuoteStr(ByRef pStr As String, pQuoteChar As String) As
String
```

Removes quotes around a string and returns the unquoted string.

Input

- `pStr`: the string to process.
- `pQuoteChar`: the quoting character to remove.

Output

Returns the unquoted string using `pQuoteChar` only when `pStr` already has such quotation marks *at both ends*, otherwise `pStr` is left untouched.
See `QuoteStr()`.

ArrayPrim

Array primitives.

ArrayPrim.Arrays

Dependencies: (none)

Vocabulary:

- Vector: a 1-dimension array.
- Array: any array with more than one dimension.
- Nested array: an array with another array in it.

Primitives

`AddToVector()`

```
Sub AddToVector(ByRef pVector As Variant, pItem as Variant)
```

Adds an item to a single-dimension array without data loss.

Input

- `pVector`: a vector array (1-D array).
- `pItem`: the item to add to the array.

`Array2DToDataArray()`

```
Function Array2DToDataArray(ByRef pArray2D As Variant) As Variant
```

Creates a 2-dimensions nested array (compatible with the Calc ranges `.DataArray` property) from a 2D array.

Input

- `pArray2D`: the input array. Must be 1 or 2D otherwise it is ignored.
  The input array is supposed to be an existing array.

Output

A data array compatible with Calc ranges `.DataArray`. The output may be a 1D or 2D array, or `Null` according to the input array characteristics.
See also: `DataArrayToArray2D()`

`ArrayDimCount()`

```
Function ArrayDimCount(ByRef pArray As Variant) As Integer
```

Returns the number of dimensions of an array.

Input

- `pArray`: the array to be tested.

Output

The dimension number.

Note: An unallocated dynamic array has 0 dimension. In this situation, the function returns `-1`.

`ArrayExists()`

```
Function ArrayExists(ByRef pArray As Variant) As Boolean
```

Returns `True` If the array exists and can be manipulated.

Input

- `pArray`: the array to check

Output

`True` if `pArray` is a valid variable otherwise `False`

Note: An array is considered as existing if the following constraints are all checked:
– `pArray` is not `Null`.
– `pArray` is an array-type variable.
– `pArray` upper bound is greater or equal than its lower bound.

`ArrayIsEmpty()`

```
Function ArrayIsEmpty(ByRef pArray() As Variant, Optional pDim As Byte)
As Boolean
```

Checks whether an array dimension is empty (has no item) or not.

Input

- `pArray`: the array to be checked.
- `pDim`: (optional) the dimension to test. `pDim` must be `1` or greater.
  Defaults to `1`.

Output

Returns `True` if the `pDim`th dimension of `pArray()` is empty (has no element), otherwise `False`.

`ConcatVectors()`

```
Function ConcatVectors(ByRef pVectors As Variant) As Variant
```

Returns a unique vector (1D array) from an array of vectors.
Vectors are concatenated in the input order.

Input

- `pVectors`: an array of vectors.

Output

The vector resulting from the concatenation of all input vectors.

`DataArrayToArray2D()`

```
Function DataArrayToArray2D(ByRef pDataArray() As Variant, Optional
pIgnoreEmpty As Boolean) As Variant
```

Returns a 2-dimensions array from a 2D nested array (typically a Calc range `.DataArray` property).

Input

- `pDataArray`: the nested 2D array to convert.
- `pIgnoreEmpty`: (optional) a flag stating whether empty cells will be ignored or not.
  Defaults to `True`.

Output

A 2-dimensions array with the original array data.
See also: `Array2DToDataArray()`

`QuickSort1D()`

```
Sub QuickSort1D(ByRef pArray() As Variant, Optional pFrom As Long,
Optional pTo as Long)
```

Sorts a one dimension array in ascending order using the Quicksort algorithm.
starting at the `pFrom` item ans ending with the `pTo` item (`pFrom` and `pTo` are included in the sorted set).
Default value for `pFrom` is `LBound(pArray)`.
Default value for `pTo` is `UBound(pArray)`.

Note: the sub is recursive, which means that very big sets of data will probably cause memory overflow.

Adapted from VBScript in http://www.4guysfromrolla.com/webtech/012799-2.shtml

---

`ReverseVector()`

```
Function ReverseVector(ByRef pVector As Variant) As Variant
```

Reverses the items order of a vector (1-D array).

Input

- `pVector`: a 1-D array.
  This array is supposed to exist.

Output

The same array, with its items in reverse order (case sensitive).

`ShellSort()`

```
Sub ShellSort(ByRef pArray)
```

Sorts a one dimension array in ascending order using the Shellsort algorithm.
Adapted from VBScript in http://www.4guysfromrolla.com/webtech/012799-2.shtml

`SortVectorBubble()`

```
Function SortVectorBubble(ByRef pVector As Variant, Optional ByRef pAsc
As Boolean)
```

Sort a vector (1-D array) data using the bubble sort algorithm (slow for large arrays).

Input

- `pVector`: the vector to sort.
  The input array must exist.
- `pAsc`: (optional) `True` if the sort must be in ascending order, otherwise `False`.
  Defaults to `True`.

Output

The sorted vector. The sort is case-sensitive.
Adapted from https://helloacm.com/bubble-sort-in-vbscript/

`StringPosInArray()`

```
Function StringPosInArray(ByRef pArray As String, pStr as String,
Optional pCompare as Integer) as Long
```

Returns the position of a string in an array.

Input

- `pArray`: the array to be searched into.
- `pStr`: the string to search.
- `pCompare`: (optional) the comparison mode.
  Defaults to `1`.

`pCompare` values:

- `0` = Binary comparison
- `1` = Text comparison

Output

Returns the position of `pStr` in `pArray()` or `-1` if not found.
Adapted from Alain de la Chaume in
https://forum.openoffice.org/fr/forum/download/file.php?id=5018&sid=fc90d6ba94b035b746fede1326c8d907

`VectorFromStringNums()`

```
Function VectorFromStringNums(ByRef pInputStr As String, pSepChar As
String, pRangeChar As String, pAsLongs As Boolean) As Variant
```

Converts a string of numbers into an array of discrete values. This mimics the page selection behavior in the UI print dialog.

Input

- `pInputStr`: the string to convert.
  The string is made of integer numbers (longs) with separators.
- `pSepChar`: the item separator character (one only).
- `pRangeChar`: the range separator character (one only).
- `pAsLongs`: convert the output array items into Longs.

Output

A vector (1-D array) containing each item in the input string, separated as specified with `pSepChar` and `pRangeChar`.
If an error occurred, or if the input string is a zero-length one, the output is `Null`.

Usage

```
Str = "-1, 3-6, 10"
VectorFromStringNums(Str, ",", "-", True)
```

returns the array: `(-1, 3, 4, 5, 6, 10)`

```
VectorFromStringNums(Str, ",", "-", False)
```

returns the array: `("-1", "3", "4", "5", "6", "10")`

`VectorToDataArray()`

```
Function VectorToDataArray(ByRef pVector As Variant, Optional pRaw As
Boolean) As Variant
```

Converts a vector (1D array) into a DataArray that can be used to feed a Calc range DataArray property.
A Calc DataArray is a 2D nested array.

Input

- `pVector`: the vector to convert.
- `pRaw`: (optional) Defines what should be done with empty vector items.

If set to `True` (the default), data is left as-is. That is, copying the data into a Calc range may result in cells with error values for empty values. If set to `False`, the empty value is replaced with a zero-length string.

Output

The DataArray. Its first array index is `0`.

MathPrim

MathPrim.Math

Dependencies: (none)

Basic math calculations.

Module Variables

Variable name	Type	Description
`mEpsilon`	`Double`	Stores the comparator value for equality tests. The `Init()` sub sets this variable to `0.000001` (10^{-6}). See also `MATH_DELTA` and `SetEpsilon()`.

Global Constants

Constant name	Value	Description
`MATH_DELTA`	`0.000001`	10^{-6} default value for `mEpsilon`.

Primitives

`Average()`

```
Function Average(ByRef pArray As Variant) As Double
```

Returns the average value from an array.

Input

- `pArray`: the array to process.
  The array is supposed to be populated.

Output

The average value of all the items in the array dimension 1.

Usage

`Average(Array(1, 10, 100))` ⟶ `37`

`BankersRound()`

```
Function BankersRound(ByRef pValue As Double, pDecimals As Integer) As
Double
```

Rounds a value to a given number of decimals using the “bankers rounding”, that is rounding to the nearest *even* number.

Input

- `pValue`: the value to round.
- `pDecimals`: the number of decimals for the rounding.

Output

The rounded number.

Usage

`BankersRound(1.5, 0)` ⟶ `2`
`BankersRound(2.5, 0)` ⟶ `2`
See also `Round()`.

`Init()`

```
Sub Init()
```

Sets the initial values for the `Math` module.
Currently, this Sub sets the value for `mEpsilon` which is used for numerical comparisons purposes.

`IsDifferent()`

```
Function IsDifferent(ByRef pValue1 As Double, pValue2 As Double) As
Boolean
```

Checks whether two numbers are different.

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns `True` if `pValue1` is different from `pValue2`, above the `mEpsilon` error margin.
See also `SetEpsilon()`, `IsEqual()`, `IsGreater()`, `IsLower()`.

`IsEqual()`

```
Function IsEqual(ByRef pValue1 As Double, pValue2 As Double) As Boolean
```

Checks whether two numbers are equal.

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns `True` if `pValue1` is equal to `pValue2`, within the `mEpsilon` error margin.
See also `SetEpsilon()`, `IsDifferent()`, `IsGreater()`, `IsLower()`.

`IsGreater()`

```
Function IsGreater(ByRef pValue1 As Double, pValue2 As Double) As
Boolean
```

Checks whether a number is greater than another.

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns `True` if `pValue1` is greater than `pValue2`, above the `mEpsilon` error margin.
See also `SetEpsilon()`, `IsEqual()`, `IsLower()`, `IsEqual()`, `IsDifferent()`.

`IsGreaterEqual()`

```
Function IsGreaterEqual(ByRef pValue1 As Double, pValue2 As Double) As
Boolean
```

Checks whether a number is greater or equal to another.

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns `True` if `pValue1` is greater or equal to `pValue2`, above the `mEpsilon` error margin.

`IsInRange()`

```
Function IsInRange(ByRef pValue As Double, pMin As Double, pMax As
Double, Optional pExclusive As Boolean) As Boolean
```

Checks whether a value is in a range bounds.

Input

- `pValue`: the number to check.
- `pMin`: the range lower bound.
- `pMax`: the range upper bound.
- `pExclusive`: (optional) specifies if the check must exclude the bounds or not. Defaults to `False`.

Output

Returns `True` if `pValue` is in the `pMin..pMax` range

IsLower()

```
Function IsLower(ByRef pValue1 As Double, pValue2 As Double) As Boolean
```

Checks whether a number is lower than another.

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns `True` if `pValue1` is lower than `pValue2`, above the `mEpsilon` error margin.
See also: `SetEpsilon()`, `IsGreater()`, `IsEqual()`, `IsEqual()`, `IsDifferent()`.

IsLowerEqual()

```
Function IsLowerEqual(ByRef pValue1 As Double, pValue2 As Double) As
Boolean
```

Checks whether a number is lower or equal to another.

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns `True` if `pValue1` is lower or equal to `pValue2`, above the `mEpsilon` error margin.

Max()

```
Function Max(ByRef pValue1 As Double, pValue2 As Double) As Double
```

Returns the larger of two numeric values (doubles).

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns the greatest of two decimal numbers, within the `mEpsilon` error margin.
See also: `MaxInt()`.

MaxInArray()

```
Function MaxInArray(ByRef pArray As Variant) As Double
```

Returns the greatest of the values in an array.

Input

- `pArray`: The array to process. The array is supposed to be populated.

Output

The largest value within the array within `mEpsilon` margin.
See also: `MinInArray()`.

MaxLng()

```
Function MaxLng(ByRef pNum1 As Long, pNum2 As Long) As Long
```

Returns the larger of two numeric values (longs).

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns the greatest of two integers
See also: `Max()`.

`Median()`

```
Function Median(ByRef pArray As Variant) As Double
```

Returns the median value from the items in an array.

Input

- `pArray`: The array to process. The array is supposed to be populated.

Output

The median value of the ones stored in the array.

`Min()`

```
Function Min(ByRef pValue1 As Double, pValue2 As Double) As Double
```

Returns the smaller of two numeric values (doubles).

Input

- `pValue1`: the first number.
- `pValue2`: the second number.

Output

Returns the smaller of two decimal numbers, within the `mEpsilon` error margin.

`MinInArray()`

```
Function MinInArray(ByRef pArray As Variant) As Variant
```

Returns the lowest of the values in an array.

Input

- `pArray`: The array to process. The array is supposed to be populated.

Output

The lowest value within the array within `mEpsilon` margin.
See also: `MaxInArray()`.

`Round()`

```
Function Round(ByRef pValue As Double, pDecimals As Integer) As Double
```

Rounds a value to a given number of decimals.

Input

- `pValue`: the value to round.
- `pDecimals`: the number of decimals for the rounding.

Output

The rounded number.
Usage
`Round(1.5, 0)` ⟶ `2`
`Round(2.5, 0)` ⟶ `3`
See also: `BankersRound()`.

`SetEpsilon()`

```
Sub SetEpsilon(ByRef pEpsilon As Double)
```

Sets the global `mEpsilon` value for equality calculations.
`mEpsilon` defaults to `0.000001` (10^{-6}).
See also: `Init()`, `IsEqual()`, `IsDifferent()`, `IsGreater()`, `IsLower()`.

`SwapValues()`

```
Sub SwapValues(ByRef pValue1 As Variant, ByRef pValue2 As Variant)
```

Swaps the two values `pVal1` and `pVal2`.

LibOPrim

LibreOffice application primitives.

LibOPrim.App

Dependencies: (none)

This module is related to the LibreOffice application as a whole.

Global Constants

Constant name	Value	Description
Error Constants		
`ERR_LOPRIM_CMDUNK`	`3`	Unknown command.
Misc.		
`LOPRIM_SERV_DISPATCH`	`"com.sun.star.frame.DispatchHelper"`	The dispatcher call.

Primitives

`SetFullScreen()`

```
Sub SetFullScreen(Optional ByRef pShowToolbar As Boolean)
```

Sets the current window to full screen.

Input

- `pShowToolbar`: allow or not to simultaneously hide the **FullScreen** toolbar that displays when in full screen mode.
  Defaults to `True` (the FullScreen toolbar is left on screen).

Note: the **FullScreen** toolbar can be manipulated using the `HideToolbar()` and `DisplayToolbar()` subs with `LOPRIM_TB_FULLSCREEN` as a parameter.

`ShowDocumentProperties()`

```
Sub ShowDocumentProperties()
```

Calls LibO document properties dialog.

`ShowNavigator()`

```
Sub ShowNavigator(ByRef pShow As Boolean)
```

Shows or hides the **Navigator**.

Input

- `pShow`: display (`True`) or hide (`False`) the **Navigator**.

`ShowPrinterDialog()`

```
Sub ShowPrinterDialog()
```

Calls the LibO printer dialog.

`ShowPrintPreview()`

```
Sub ShowDocumentPreview()
```

Calls the LibO document preview dialog.

`ShowSideBar()`

```
Sub ShowSidebar(ByRef pShow As Boolean)
```

Shows or hides the **Sidebar**.

Input

- `pShow`: display (`True`) or hide (`False`) the **Sidebar**.

Note: be aware that the **SideBar** may include the **Navigator**. This means that you might have to hide that one as well.

LibOPrim.DocCheckerClass

Dependencies: none.

A LibreOffice document checking class.

This class allow to create a set of checks to carry against a given document, making easier to find if this document satisfies the requirements.

Currently, this class offers checks against:

- The document type (Writer, Calc, etc.).
- The presence and value of custom properties.

This class may be used stand-alone. It is also used internally by the CalcImportClass (optionnally) for which it was initially intended.

Global Constants

Constant name	Value	Description
Error Constants		
ERR_DOCCHKCLASS_NORUN	`-1`	Process did not run.
ERR_DOCCHKCLASS_NONE	`0`	No error (OK).
ERR_DOCCHKCLASS_CANTOPENSOURCE	`1`	Source document cant be opened.
ERR_DOCCHKCLASS_SOURCENOTFOUND	`2`	Document file wasnt found.
ERR_DOCCHKCLASS_INCORRECTTYPE	`3`	Document is not a spreadsheet.
ERR_DOCCHKCLASS_CUSTOMPROPNAME	`4`	Custom property not found.
ERR_DOCCHKCLASS_CUSTOMPROPVALUE	`5`	Custom property value not found.
Misc.		
DOCCHKCLASS_DOCTYPE_NONE	`0`	(none/unknown).
DOCCHKCLASS_DOCTYPE_WRITER	`1`	Writer.
DOCCHKCLASS_DOCTYPE_WRITERWEB	`2`	Writer/Web.
DOCCHKCLASS_DOCTYPE_WRITERMASTER	`3`	Writer/Master document.
DOCCHKCLASS_DOCTYPE_CALC	`4`	Calc.
DOCCHKCLASS_DOCTYPE_IMPRESS	`5`	Impress.
DOCCHKCLASS_DOCTYPE_DRAW	`6`	Draw.
DOCCHKCLASS_DOCTYPE_MATH	`7`	Math.
DOCCHKCLASS_DOCTYPE_CHART	`8`	Chart.
DOCCHKCLASS_DOCTYPE_XFORMS	`9`	XML / XForms.
DOCCHKCLASS_DOCTYPE_BASEAPP	`10`	Base main application.

Properties

Property	Type	Description
`LastError`	Long [RO]	The last error that occurred.
`DocType`	Byte [R/W]	The document type to check. The document type value is one of the `DOCCHKCLASS_DOCTYPE_XXX` constants. The default `DOCCHKCLASS_DOCTYPE_NONE` bypasses the document type check.
`DocumentName`	String [R/W]	The name of the document to check (FQN), in OS or URL form.

Methods

`Public`

`AddPropertyCheck()`

```
Public Sub AddPropertyCheck(ByRef pPropName As String, pValue As
Variant)
```

Adds a custom property check.
The actual check is run when calling `CheckDocument()`.

Input

- `pPropName`: the name of the custom property to check.
- `pValue`: the value to check. This may be an array of values, that can be useful to check a value in a set (OR-ed).

Note Properties checks are cumulative (AND) between separate items or alternative (OR) when item values are arrays.

Note A value of `Null` implies that only the property existence is performed.

Usage example

```
MyImporter.AddPropertyCheck("SomeProperty", 2018)
MyImporter.AddPropertyCheck("OtherProperty", Array(2018, 2017))
```

Calling `MyImporter.CheckSource()`
checks that `"SomeProperty"` exists and its value is `2018`, and that `"OtherProperty"` also exists and its value is either `2018` or `2017`.

`CheckDocument()`

```
Public Function CheckDocument() As Integer
```

Checks whether the document is verifiying the constraints.
see DocType property and AddPropertyCheck() method.

Output

`ERR_DOCCHKCLASS_NONE` (`0`) if OK otherwise a result code.
Possible return codes:

- `ERR_DOCCHKCLASS_NONE`: OK.
- `ERR_DOCCHKCLASS_CANTOPENSOURCE`: can't open file.
- `ERR_DOCCHKCLASS_INCORRECTTYPE`: not the type of document wanted.
- `ERR_DOCCHKCLASS_CUSTOMPROPNAME`: custom property not found.
- `ERR_DOCCHKCLASS_CUSTOMPROPVALUE`: custom property value not found.

`ClearError()`

```
Public Sub ClearError()
```

Clears the current error.

`ClearPropertyChecks()`

```
Public Sub ClearPropertyChecks()
```

Clears the custom properties check items.

Private (private methods names are prefixed with underscores "_") They are not intended for use out of the class.	
`_AddCollectionItem()`	`Private Sub _AddCollectionItem(ByRef pColl As Object, ByRef pItem As Variant, pKey As String)` Adds an item to a collection.
`_DocTypeStr()`	`Private Function _DocTypeStr() As String` Returns the internal string identifier for the current `DocType`.
`_OpenDocument()`	`Private Function _OpenDocument(ByRef pFileName As String, Optional pHidden As Boolean, Optional pReadOnly As Boolean) As Object` Opens the document to check in the background and returns that object or `Null` if a problem occurred.

Example

```
1  Dim oDocChecker As Object
2  Dim Result As Integer
3  Set oDocChecker = New DocCheckerClass
4  oDocChecker.DocumentName = "C:\Path\To\Somefile.odt"
5  oDocChecker.DocType = DOCCHKCLASS_DOCTYPE_WRITER
6  oDocChecker.AddPropertyCheck("ApplicationName", "MyApp")
7  Result = oDocChecker.CheckDocument()
```

where `Result` holds the check result (see the `ERR_DOCCHKCLASS_XXX` constants above).

LibOPrim.Document

Dependencies: (none)

This module relates to the open documents in any of the LibreOffice modules.

Global Constants

Constant name	Value	Description
Error constants		
ERR_LOPRIM_DOCTYPE	-1	
LOPRIM_DOCTYPEUNK	0	
LOPRIM_DOCTYPECALC	1	
LOPRIM_DOCTYPEWRITER	2	
LOPRIM_DOCTYPEIMPRESS	3	
LOPRIM_DOCTYPEDRAW	4	
LOPRIM_DOCTYPEMATH	5	
LOPRIM_DOCTYPEBASE	6	

Primitives

CreateDocument()

```
Function CreateDocument(ByRef pType As Long, pHidden As Boolean,
pReadOnly As Boolean) As Object
```

Creates a new LibreOffice document and returns the corresponding object.
This proposes the most common options when opening a LibreOffice document. For more advanced uses, call `OpenDocumentEx()`.

Input

- `pHidden`: the document should be loaded as hidden.
- `pReadOnly`: the document should be loaded as read-only.

Output

The created document object or `Null` if the document couldn't be opened.
See also: `OpenDocument()` and `OpenDocumentEx()`

DocumentProtectionFlag()

```
Function DocumentProtectionFlag(ByRef pDocURL As String) As Integer
```

Checks whether a given (LibreOffice) document is protected and returns a value describing this status.
The test is based upon the fact that a LibO protected document zip structure doesn't contain the otherwise usual '`Thumbnails`' directory.

Input

- `pDocURL`: the FQN document name in URL or OS form.
  This document is supposed to exist.

Output

- `1` if the document is protected in any way.
- `0` if the document is not protected.
- or `-1` if an error occurred (ex: the document is not a LibO one/not in zip format).

GetCurrentDirectory()

```
Function GetCurrentDirectory() As String
```

Returns the document directory name.

Input

- pDoc: (optional) the document object to process.
  Defaults to the current document.

Output

The document directory name in URL form.
The trailing / is part of the returned string.

Note: This information is not available when a document has not yet been written to disk.

GetLibODocumentType()

```
Function GetLibODocumentType(Optional ByRef pDoc As Object) As Long
```

Returns the LibreOffice document type.

Input

- pDoc: the document to look at. If omitted, looks at the current document.

Output

The document type or an error value

- Possible type values: see the LOPRIM_DOCTYPE above.
- Error: the result may also be the error when the pDoc document is not set.

IsBaseDocument()

```
Function IsBaseDocument(Optional ByRef pDoc As Object) As Boolean
```

Returns True if pDoc is a Base document, otherwise False. If omitted, looks at the current document.

Note: if an error code is returned by GetLibODocType, it returns False as well.

IsCalcDocument()

```
Function IsCalcDocument(Optional ByRef pDoc As Object) As Boolean
```

Returns True if pDoc is a Calc document, otherwise False. If omitted, looks at the current document.

Note: if an error code is returned by GetLibODocType, it returns False as well.

IsDrawDocument()

```
Function IsDrawDocument(Optional ByRef pDoc As Object) As Boolean
```

Returns True if pDoc is a Draw document, otherwise False. If omitted, looks at the current document.

Note: if an error code is returned by GetLibODocType, it returns False as well.

IsImpressDocument()

```
Function IsImpressDocument(Optional ByRef pDoc As Object) As Boolean
```

Returns True if pDoc is an Impress document, otherwise False. If omitted, looks at the current document.

Note: if an error code is returned by GetLibODocType, it returns False as well.

IsMathDocument()

```
Function IsMathDocument(Optional ByRef pDoc As Object) As Boolean
```

Returns True if pDoc is a Math document, otherwise False. If omitted, looks at the current document.

Note: if an error code is returned by GetLibODocType, it returns False as well.

`IsWriterDocument()`

```
Function IsWriterDocument(Optional ByRef pDoc As Object) As Boolean
```

Returns `True` if `pDoc` is a Writer document, otherwise `False`. If omitted, looks at the current document.

Note: if an error code is returned by `GetLibODocType`, it returns `False` as well.

`ModuleIdentifierStr()`

```
Function ModuleIdentifierStr(Optional ByRef pDoc As Object) As String
```

Returns the LibreOffice module identifier string for a given document.
This identifier may be used with the `SupportsService()` function.

Input

- `pDoc`: (optional) the document which module identifier to get.
  Defaults to the current document.

Output

The document module identifier.
Ex: for a text document (Writer module), returns
`"com.sun.star.text.TextDocument"`

`OpenDocument()`

```
Function OpenDocument(ByRef pFileName As String, pHidden As Boolean,
pReadOnly As Boolean, pAsTemplate As Boolean) As Object
```

Opens an existing LibreOffice document and returns the corresponding object.
This proposes the most common options when opening a LibreOffice document. For more advanced uses, call `OpenDocumentEx()`. [TBD]

Input

- `pFileName`: The FQN of the document name (in URL or OS form).
  No check is accomplished against the document existence.
- `pHidden`: the document should be loaded as hidden.
- `pReadOnly`: the document should be loaded as read-only.
- `pAsTemplate`: a new document is created, using `pFilename` as a template.

Output

The document object or `Null` if the document couldn't be opened.
See also: `OpenDocumentEx()` and `CreateDocument()`.

`OpenDocumentCopy()`

```
Function OpenDocumentCopy(ByRef pTgtFileName As String, Optional
pHidden As Boolean, Optional ByRef pSrcDoc) As Object
```

Opens a copy of a document and returns the document object.
Process: the source document is stored as a copy as defined by `pTgtFileName` then this copy is opened, hidden or not.

Input

- `pTgtFileName`: the document copy file FQN. (in URL or OS form).
- `pHidden`: (optional) defines whether the copy should be made visible or not.
  Defaults to `False`.
- `pSrcDoc`: (optional) the source document.
  Defaults to the current document.

Output

Returns the document copy object or `Null` if it couldn't be copied/opened.

Usage

`MyDoc = OpenDocumentCopy("C:\Path\To\MyDoc.ods", True, ThisComponent)`
creates a copy of `ThisComponent` under the name of `"C:\Path\To\MyDoc.ods"` then opens it as hidden and returns that new object.

`OpenDocumentEx()`

```
Function OpenDocumentEx(ByRef pFileName As String, pOptions As Variant)
As Object
```

TBD

LibOPrim.SpecialFiles

Dependencies: IOPrim.Files, IOPrim.TextFiles, StringPrim.Strings

This module helps managing the LibreOffice special files. These special files are the lock files and the (Calc-only) share files. These are automatically created by LibreOffice whenever a document is opened (lock file) or a Calc spreadsheet for which the share mode is enabled (share file).

The special file naming is similar in both situations:

- `.~lock.somefile.odt#` (all modules)

  The file is opened in write mode.

- `.~sharing.somefile.ods#` (Calc-only)

  The Calc spreadsheet is opened and is set for share mode (**Tools > Share mode**).

where the `somefile.od?` part is the name of the associated user's file name. These special files are always created in the same directory as the user's file.

The primitives here allow to check that a special file is present, read it and gain access to its contents. This may be of use particularly when using the Calc share mode.

Primitives

`CalcShareFileName()`

```
Function CalcShareFileName(Optional ByRef pDoc As Object) As
String
```

Returns the Calc share file name for a given document.

Input

- `pDoc`: (optional) the document which share file name to retrieve. Defaults to the current document.

Output

The share file name or a zero-length string if the document object has not been saved yet.
A Calc share file name form is: `.~sharing.TheFileName.ods#`

Note: because of the URL form output, the final '`#`' character must be returned as its Ascii hex value, prefixed with a '`%`', that is '`%23`'.

`GetLibreOfficeSpecialFileData()`

```
Function GetLibreOfficeSpecialFileData(ByRef pSpecialFileName As
String) As Variant
```

Returns an array containing the LibreOffice special file's users information.
The LibO special files are "lock files" and (Calc-only) "share files".

Input

- `pSpecialFileName`: the special file name which data to retrieve (FQN, in OS or URL form).

Output

The data in a nested array or `Null` if the specified file name doesn't exist.
Output array format
One line per user, nesting a 5 column array:
0: her/his user name (see LibreOffice identity settings).
1: the OS user name (name & workstation name).
2: the OS user name (only).
3: the date/time the user opened the shared document.
4: the LibreOffice user profile directory.

Note: A lock file has only one user entry: the one for the initial write-mode user.

A share file may have several (as many as users currently accessing the work file).

`IsCalcDocumentShared()`

```
Function IsCalcDocumentShared(Optional ByRef pDoc As Object) As
Boolean
```

Checks whether a Calc document is used in share mode or not.

Input

- `pDoc`: (optional) the document to check.
  There's no check that the document actually is a Calc spreadsheet.
  Defaults to the current document.

Output

`True` if the document is in Calc shared mode, `False` otherwise.

`LockFileName()`

```
Function LockFileName(Optional ByRef pDoc As Object) As String
```

Returns the lock file name for a given document
The lock file name form is: `.~lock.TheFileName.odt#` (here, for a Writer document).

Input

- `pDoc`: (optional) the document which lock file name to retrieve.
  Defaults to the current document.

Output

The lock file name or a zero-length string if the document object has not been saved yet.

Note: because of the URL form output, the final '`#`' character must be returned as its Ascii hex value, prefixed with a '`%`', that is '`%23`'.

LibOPrim.Graphics

Dependencies: (none)

Global Constants

Constant name	Value	Description

Primitives

`GetGraphicFromResource()`

```
Function GetGraphicFromResource(ByRef pGraphicName As String, Optional
ByRef pDoc As Object) As Object
```

Retrieves a graphic object from a document resources.

Input

- `pGraphicName`: the name we had set to the resource.
- `pDoc`: (optional) the document with the resource.
  Defaults to the current document.

Output

The graphic object or `Null` if it wasn't retrieved.

`GetImage()`

```
Function GetImage(ByRef pFileName As String) As Variant
```

Retrieves an image object

Input

- `pFileName`: the image file name in URL or OS form.

Output

The image object or `Null` if not found.

`GetImageManager()`

```
Function GetImageManager(Optional ByRef pDoc As Object) As Object
```

Returns the ImageManager object for the current LibreOffice module (Writer, Base, etc).

Input

- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Adapted from librebel in https://ask.libreoffice.org/en/question/111748/how-to-change-toolbar-icon-back-to-text/

LibOPrim.ToolBars

Global Constants

Constant name	Value	Description
Error Constants		
ERR_TBAR_HIDDEN	-2	(pseudo-error) the toolbar is hidden
ERR_TBAR_VISIBLE	-1	(pseudo-error) the toolbar is visible
ERR_TBAR_NONE	0	No error.
ERR_TBAR_UNKNOWN	1	Unknown toolbar.
ERR_TBAR_DEL	2	The specified toolbar can't be deleted (typically applies to *LibreOffice* native toolbar).
LibreOffice Toolbar Resource Names		
LOPRIM_TB_STANDARD	"private:resource/toolbar/standardbar"	Resource name for the Standard toolbar.
LOPRIM_TB_FIND	"private:resource/toolbar/findbar"	Resource name for the Find toolbar.
LOPRIM_TB_FORMAT	"private:resource/toolbar/formatobjectbar"	Resource name for the Format toolbar.
LOPRIM_TB_FULLSCREEN	"private:resource/toolbar/fullscreenbar"	Resource name for the FullScreen toolbar.
LOPRIM_TB_MENU	"private:resource/menubar/menubar"	Resource name for the Menu toolbar.
LOPRIM_TB_STATUS	"private:resource/statusbar/statusbar"	Resource name for the Status toolbar.

Primitives

CustomToolbarsToArray()

```
Function CustomToolbarsToArray(Optional ByRef pDoc As Object) As Variant
```

Returns an array that holds data about all custom toolbars within a document.

Input

- `pDoc`: (optional) the document to explore.
  Defaults to the current document.

Ouput

The returned array is a 2-dimension structure:
`array(toolbar count, 2)`
Adapted from Loopingss message, in
https://forum.openoffice.org/fr/forum/viewtopic.php?f=15&t=27 370

`DeleteToolbar()`

```
Function DeleteToolbar(ByRef pToolbarName As String, Optional ByRef
pDoc As Object) As Long
```

Deletes a given toolbar.

Input

- `pToolbarName`: the toolbar name.
  `pToolbarName` can be either the UI name (ex. local document toolbars) or the resource name (eg. *LibreOffice* native toolbars; see the `LOPRIM_TB_XXX` constants).
- `pDoc`: (optional) the document (defaults to `ThisComponent`)

Output

Returns an error code:

- `ERR_TBAR_NONE`: no error.
- `ERR_TBAR_UNKNOWN`: the specified toolbar is unknown
- `ERR_TBAR_DEL`: the specified toolbar can't be deleted (LibO toolbars)

`DisplayToolbar()`

```
Function DisplayToolbar(ByRef pToolbarName As String, Optional ByRef
pDoc As Object) As Long
```

Shows a toolbar.

Input

- `pToolbarName`: the toolbar name.
  `pToolbarName` can be either the UI name (ex. local document toolbars) or the resource name (eg. *LibreOffice* native toolbars; see the `LOPRIM_TB_XXX` constants).
- `pDoc`: the document (defaults to `ThisComponent`)

Output

Returns an error code:

- `ERR_TBAR_NONE`: no error.
- `ERR_TBAR_UNKNOWN`: the specified toolbar is unknown
- `ERR_TBAR_DEL`: the specified toolbar can't be displayed (LibO toolbars)

`GetToolbarResName()`

```
Function GetToolbarResName(ByRef pUIToolbarName As String, Optional
pDoc As Object) As String
```

Returns the resource name for a toolbar.

Input

- `pToolbarName`: the toolbar name.
  `pToolbarName` can be either the UI name (ex. local document toolbars) or the resource name (eg. *LibreOffice* native toolbars; see the `LOPRIM_TB_XXX` constants).
- `pDoc`: the document (defaults to `ThisComponent`)

Output

Returns the resource name or a zero-length string if `pToolbarName` is not found.
Adapted from hanya's message in
https://forum.openoffice.org/en/forum/viewtopic.php?f=20&t=56 336

`HideToolbar()`

```
Function HideToolbar(ByRef pToolbarName As String, Optional ByRef pDoc
As Object) As Long
```

Hides a toolbar.

Input

- `pToolbarName`: the toolbar name.
  `pToolbarName` can be either the UI name (ex. local document toolbars) or the resource name (eg. *LibreOffice* native toolbars; see the `LOPRIM_TB_XXX` constants).
- `pDoc`: the document (defaults to `ThisComponent`)

Output

Returns an error code:

- `ERR_TBAR_NONE`: no error.
- `ERR_TBAR_UNKNOWN`: the specified toolbar is unknown
- `ERR_TBAR_HIDDEN`: the specified toolbar is already hidden (LibO toolbars)

`ToolbarVisible()`

```
Function ToolbarVisible(ByRef pToolbarName As String, Optional ByRef
pDoc As Object) As Long
```

Returns the status of a given toolbar: visible or not.

Input

- `pToolbarName`: the toolbar name.
  `pToolbarName` can be either the UI name (ex. local document toolbars) or the resource name (eg. *LibreOffice* native toolbars; see the `LOPRIM_TB_XXX` constants).
- `pDoc`: the document (defaults to `ThisComponent`)

Output

Returns a status or error code:

- `ERR_TBAR_NONE:` no error.
- `ERR_TBAR_UNKNOWN`: the specified toolbar is unknown
- `ERR_TBAR_HIDDEN`: (status) the toolbar is hidden
- `ERR_TBAR_VISIBLE`: (status) the toolbar is visible

LibOPrim.CustomProperties

LibreOffice document custom properties management.

Dependencies: LibOPrim.UNO

Global Constants

Constant name	Value	Description
Custom property types		
CPROP_TYPE_UNK	-1	Unknown type.
CPROP_TYPE_STRING	1	String type.
CPROP_TYPE_NUMBER	2	Numeric type.
CPROP_TYPE_DATE	3	Date type.
CPROP_TYPE_UNODATE	4	UNO date type.
CPROP_TYPE_UNODATETIME	5	UNO datetime type.
CPROP_TYPE_UNODURATION	6	UNO duration.
CPROP_TYPE_YESNO	7	Boolean value (aka Yes/no)
Errors		
ERR_CPROP_NORUN	-1	The process did not run.
ERR_CPROP_OK	0	No error.
ERR_CPROP_CREATE	1	The property could not be created.
ERR_CPROP_DELETE	2	The property could not be deleted
ERR_CPROP_NOTFOUND	3	The property was not found.
ERR_CPROP_NAME	4	Illegal property name.
ERR_CPROP_EXISTS	5	The property name already exists.
ERR_CPROP_TYPE	6	Not supported type for a property.

Primitives

`CreateCustomProperty()`

```
Function CreateCustomProperty(ByRef pArrProps As Variant, Optional
ByRef pDoc As Object) As Long
```

Creates a custom property and returns a status code.

Input

- `pArrProps`: the property defines as an array of two items (`0`: property name ; `1`: property value).
  The actual value type defines the property type.
- `pDoc`: (optional) the document where the property must be created.
  Defaults to the current document.

Output

An error code or `ERR_CPROP_OK (0)` if correctly executed.
Possible error codes:

- `ERR_CPROP_NORUN`: the process did not run.
- `ERR_CPROP_NAME`: the name is not set or illegal.
- `ERR_CPROP_EXISTS`: the name already exists.
- `ERR_CPROP_CREATE`: the property could not be created.
- `ERR_CPROP_TYPE`: the provided type is not supported

Dependencies

`LibOPrim.UNO.ImplementsUNOstruct()`

`CustomPropertyExists()`

```
Function CustomPropertyExists(ByRef pName As String, Optional ByRef
pDoc As Object) As Boolean
```

Checks whether a given custom property exists.

Input

- `pName`: the custom property name.
- `pDoc`: (optional) the document where the property must be created.
  Defaults to the current document.

Output

`True` if the custom property `pName` exists, `False` otherwise.

`CustomPropertiesToArray()`

```
Function CustomPropertiesToArray(Optional ByRef pDoc As Object) As
Variant
```

Returns a document custom properties as an array.

Input

- `pDoc`: (optional) the document to analyze.
  Defaults to the current document.

Output

A 2-D array containing 3 columns with: the properties names (0), values (1) and types (2)
The type references one of the `CPROP_TYPE_Xxxx` constants.

`CustomPropertyType()`

```
Function CustomPropertyType(ByRef pValue As Variant) As Integer
```

Returns a custom property value type.

Input

- `pValue`: the value to check.

Output

The type of `pValue`. The type values is one of the `CPROP_TYPE_Xxx` constants.
If the type is none of the ones supported by custom properties, returns `CPROP_TYPE_UNK`.

DeleteAllCustomProperties()

```
Sub DeleteAllCustomProperties(Optional ByRef pDoc As Object)
```

Removes all custom properties from a document.

Input

- `pDoc`: (optional) the document from which the properties must be deleted. Defaults to the current document.

DeleteCustomProperty()

```
Function DeleteCustomProperty(ByRef pName As String, Optional pDoc
As Object) As Long
```

Deletes a custom property and returns a status code.

Input

- `pName`: the custom property name to suppress,
- `pDoc`: (optional) the document where the property must be deleted. Defaults to the current document.

Output

- `ERR_CPROP_OK` (`0`) if the property was created.
- `ERR_CPROP_NORUN`: the process did not run.
- `ERR_CPROP_NOTFOUND`: the property name was not found.
- other Basic runtime errors.

GetCustomProperty()

```
Function GetCustomProperty(ByRef pName As String, Optional pDoc As
Object) As Object
```

Gets a custom property object.

Input

- `pName`: the custom property name,
- `pDoc`: (optional) the document where the property must be found. Defaults to the current document.

Output

The custom property object or `Null` if not found.

GetCustomPropertyValue()

```
Function GetCustomPropertyValue(ByRef pName As String, Optional
pDocument As Object) As Variant
```

Retrieves a given property value.

Input

- `pName`: the property name to query.
- `pDoc`: (optional) the document to process. Defaults to the current document.

Output

The property value or `Null` if not found.

`SetCustomPropertyValue()`

```
Function SetCustomPropertyValue(ByRef pName As String, pValue As
Variant, Optional pDoc As Object) As Long
```

Sets a custom property value.

Input

- `pName`: the property name.
- `pValue`: the property value to set.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

An error code or `ERR_CPROP_OK` (`0`) if executed correctly.
Possible error codes

- `ERR_CPROP_NORUN`: the process did not run.
- `ERR_CPROP_NOTFOUND`: the property was not found for this document.
- other Basic runtime errors.

LibOPrim.ProgressClass

Dependencies: (none)

This class is very similar to the FormPrim.ProgressWidgetClass(see page 163. While that one is a form progress bar class, this ProgressClass is dedicated to all-purpose displaying and managing progress actions.

A progress bar management class for user feedback during lengthy processes.

This class encapsulates the LibreOffice documents progressbar functionalities. It aims at providing an entry point where to manage progress bars steps, messages and display delays.

This class defines the properties of a progress bar:

- a progress bar object.
- messages and delays for the different process steps.
- messages and delays for startup and termination steps.

TProgressStepInformation

This structure stores a message and a display delay for a given progress step.

Member name	Type	Description
rMessage	`String`	The message to display for a given step.
rDelay	`Long`	The message display delay, in ms..

Primitives

This module offers a factory function that can create an object of `TProgressStepInformation`. Call it from your code to initialize such objects.

`CreateProgressStepInformation()`

```
Function CreateProgressStepInformation(Optional ByRef pMsg As
String, Optional ByRef pDelay As Long) As
TProgressStepInformation
```

A factory function for the `TProgressStepInformation` type.

Input

- `pMsg`: (optional) the message for a step.
  Defaults to a zero-length string.
- `pDelay`: (optional) the delay during which the message is displayed, in ms.
  Defaults to `0`.

Output

A `TProgressStepInformation` object.

Properties

Property	Type	Description
`CurrentStep`	Object [RO]	The progress current step information. Returns an object of `TProgressStepInformation` type.
`CurrentStepNum`	Long [RO]	The progress current step number. The steps are numbered from `0` (first), then `1` (first added step) to `N` (n^{th} added step) then `-1` (last).
`FirstStep`	Object [RO]	The progress first step information Returns an object of `TProgressStepInformation` type. Use `SetFirstStep()` method to set this object.
`LastStep`	Object [RO]	The progress last step information Returns an object of `TProgressStepInformation` type. Use `SetLastStep()` method to set this object.
`ProgressBar`	Object [R/W]	The progress bar to show. Set it from a document `.CurrentController.StatusIndicator` object.

Methods

`Public`

`AddStep()`

```
Public Sub AddStep(ByRef pMsg As String, Optional pDelay As Long)
```

Adds a step to the progress.

Input

- `pMsg`: the message for the step
- `pDelay`: (optional) then message delay for the display.Defaults to `0`.

`AddSteps()`

```
Public Sub AddSteps(ByRef pArrSteps As Variant)
```

Adds a set of steps to the steps collection.
The steps are added at the end of the steps collection.
The first and last steps are set using the dedicated subs `SetFirstStep()` and `SetLastStep()`.

Input

- `pArrSteps`: a nested array of steps information.
  Each item is an array: `("Message", DelayMS)`
  At runtime, the items are displayed in the insertion order.

Usage

```
AddSteps(Array(Array("Step1", 0), Array("Step2", 1000)))
```

Adds two steps: the first one displays “Step1” with no delay, the second one displays “Step2” during 1,000 ms.

`NextStep()`

```
Public Sub NextStep(Optional ByRef pInc As Long, Optional ByRef pMsg As
String)
```

Displays the next step information.

Input

- `pInc`: (optional) the increment value.
  Defaults to `1`.
- `pMsg`: (optional) a message to display.
  Defaults to the message recorded in the steps collection.
  If the message is a zero-length string, the displayed message is not updated.

ProgressEnd()

```
Public Sub ProgressEnd(Optional ByRef pMsg As String)
```

Ends the progress bar display.

Input

- `pMsg`: (optional) a message to display.
  Defaults to the message recorded in the `LastStep`.
  If the message is a zero-length string, the displayed message is not updated.

ProgressStart()

```
Public Sub ProgressStart(Optional ByRef pMsg As String)
```

Starts the progress bar display.

Input

- `pMsg`: (optional) a message to display.
  Defaults to the message recorded in the `FirstStep`.
  If the message is a zero-length string, the displayed message is not updated.

SetFirstStep()

```
Public Sub SetFirstStep(ByRef pMsg As String, Optional pDelay As Long)
```

Sets the first step information.

Input

- `pMsg`: the message to display.
- `pDelay`: (optional) the display delay in ms.
  Defaults to `0`.

SetLastStep()

```
Public Sub SetLastStep(ByRef pMsg As String, Optional pDelay As Long)
```

Sets the last step information.

Input

- `pMsg`: the message to display.
- `pDelay`: (optional) the display delay in ms.
  Defaults to `0`.

Private
(private methods are prefixed with underscores "_")
These are not intended for use out of the class.

_AddCollectionItem()

```
Private Sub _AddCollectionItem(ByRef pColl As Object, ByRef pItem As
Variant, pKey As String)
```

Adds an item to a collection.
If the key is already present, the existing item is replaced with the new one.

Input

- `pColl`: the collection for storing the item.
- `pItem`: the item to add.
- `pKey`: the key for the item.

_ArrayExists()

```
Private Function _ArrayExists(ByRef pArray As Variant) As Boolean
```

Checks whether an array exists (is dimensioned) and can be manipulated.

Input

- `pArray`: the array to check.

Output

`True` if `pArray` is a valid variable otherwise `False`.

LibOPrim.UNO

Dependencies: (none)

Global Constants

Constant name	Value	Description

Primitives

`ImplementsUNOstruct()`

```
Function ImplementsUNOstruct(ByRef pStruct As Object, ByRef pStructName
As String) As Boolean
```

Checks whether a UNO (structure) object implements a given UNO struct.

Input

- `pStruct`: the UNO structure object to check.
- `pStructName`: the UNO structure name to check against.

Output

`True` if `pStruct` implements `pStructName`, `False` otherwise or if `pStruct` is not a UNO structure object.

Note: we can't call the `HasUnoInterfaces()` function on UNO structs as this function only applies to UNO *objects*.

LibOPrim.Extensions

Dependencies: (none)

Global Constants

Constant name	Value	Description

Primitives

`ExtensionDir()`

```
Function ExtensionDir(ByRef pExtID As String) As String
```

Returns the installation directory for a given extension.

Input

- `pExtID`: the extension identifier.

Output

The extension installation directory or a zero-length string if not found.

Note: The directory name gets a final "\".

CalcPrim

Primitives for the LibreOffice Calc module.

CalcPrim.Functions

Dependencies: (none)

Execute Calc functions. The programmatic versions of some Calc functions are only meant as examples of what can be done with all Calc functions.

CalcFunc_CountIf()

```
Function CalcFunc_CountIf(ByRef pRange As Object, pCriterion As
Variant) As Variant
```

A programmatic version of the `COUNTIF()` Calc function.

Input

- `pRange`: the range that contains data to count.
- `pCriterion`: the counting criterion.

Output

The number of occurrences that meet the criterion in the range.

CalcFunc_FilterXML()

```
Function CalcFunc_FilterXML(ByRef pXMLdoc As String, pXPath As String)
As Variant
```

A programmatic version of the `FILTERXML()` Calc function.

Input

- `pXMLdoc`: the XML stream name
- `pXPath`: the XPath expression

Output

The searched data or `Null` if not found

CalcFunc_Match()

```
Function CalcFunc_Match(ByRef pSearch As Variant, pVector As Object,
Optional pMode As Integer) As Long
```

A programmatic version of the `MATCH()` Calc function.

Input

- `pSearch`: the searched value
- `pVector`: the vector (1-column range) that contains data to search
- `pMode`: (optional) `-1`, `0` or `1` (see the `MATCH()` function help). Defaults to `1`.

Output

The row/column on which the searched data was found or `-1` if not found.

CalcFunc_VLookup()

```
Function CalcFunc_VLookup(ByRef pSearch As Variant, pRange As Object,
pLookup As Integer, Optional pExact As Byte) As Variant
```

A programmatic version of the `VLOOKUP()` Calc function.

Input

- `pSearch`: the searched value
- `pRange`: the range that contains data to search. The searched column must be the first one.
- `pLookup`: the lookup column (1-based) within the search range
- `pExact`: (optional) `0` or `1` for exact lookup (`1`) or not (`0`). Defaults to `0`.

Output

The searched data or `Null` if not found in case of exact search

---

CalcFunc_WebService()

```
Function CalcFunc_WebService(Byref pURI As String) As Variant
```

A programmatic version of the WEBSERVICE() Calc function.

Input

- pURI: the URI value

Output

The searched data or Null if not found

GetCalcFunctionObject()

```
Function GetCalcFunctionObject() As Object
```

Returns a Calc spreadsheet function object.

RunSpreadsheetFunction()

```
Function RunSpreadsheetFunction(ByRef pFuncName As String, pArrParams()
As Variant) As Variant
```

Returns the result of a spreadsheet function execution.

Input

- pFuncName: the spreadsheet function name (eg: "AVERAGE")
  pFuncName must be expressed using the internal (English) function name.
- pArrParms(): an array containing all parameters for the function execution. The array contents is function-specific. See the Calc help to know which information to store there.

Output

A variant value with the result of the function execution or Null if something went wrong.

Example: MATCH function

```
RowIndex = RunSpreadsheetFunction("MATCH", Array(Criterion, Vector,
Option))
```

where

- Criterion: the data used as a criterion, type matching the one in the searched vector.
- Vector: the vector in which to match.
- Option: the search option (for more information, see Calc help).

Returns the row index where the data matches the criterion.

Internal subprograms

_CalcFuncRange()

```
Function _CalcFuncRange(ByRef pRange As Object) As Object
```

Returns a range for use in the CalcFunc_Xxxx functions.

CalcPrim.Spreadsheet

Dependencies: (none)

Calc spreadsheet subprograms.

`ShowColumns()`

```
Sub ShowColumns(ByRef pSheetName As String, pRangeName As String, ByRef
pVisible As Boolean, ByRef Optional pDoc)
```

Shows or hides columns of a sheet.

Input

- `pSheetName`: the sheet name to process.
- `pRangeName`: the range which columns to show or hide.
- `pVisible`: set to `True` to show or `False` to hide.
- `pDoc`: (optional) the document to process. When `pDoc` is not specified, `ShowColumns` applies to the current spreadsheet.

`ShowInputLine()`

```
Sub ShowInputLine(ByRef pVisible As Boolean)
```

Displays or hides the Calc input line (formula bar).

Input

- `pVisible`: `True` to set the formula bar visible, otherwise `False`.

Dependency

LibOPrim.App._UNOCommand()

`ShowRows()`

```
Sub ShowRows(ByRef pSheetName As String, pRangeName As String, pVisible
As Boolean, Optional ByRef pDoc As Object)
```

Shows or hides rows of a sheet.

Input

- `pSheetName`: the sheet name to process.
- `pRangeName`: the range which rows to show or hide.
- `pVisible`: set to `True` to show or `False` to hide.
- `pDoc`: (optional) the document to process. When `pDoc` is not specified, `ShowRows` applies to the current spreadsheet.

`ToggleGrid()`

```
Sub ToggleGrid()
```

Toggles the current sheet grid display.

Dependency

LibOPrim.App._UNOCommand()

CalcPrim.Sheet

Dependencies: (none)

Global Constants

Constant name	Value	Description
COLMAX400	1024	The maximum number of columns in a sheet (as of LibreOffice v.4.0 onwards).
LastRowIndex() function return errors		
ERR_ROWINDEX_EMPTY	-1	The column is empty (this is not an error per se).
ERR_ROWINDEX_UNKSHEET	-2	The sheet is unknown.
ERR_ROWINDEX_UNKSHEETINDEX	-3	The sheet index doesn't exist.
ERR_ROWINDEX_UNKSHEETNAME	-4	The sheet name doesn't exist.
ERR_ROWINDEX_UNKCOLINDEX	-5	The column index doesn't exist.
ERR_ROWINDEX_UNKCOLNAME	-6	The column name doesn't exist.

Primitives

GetColNameFromNumber()

```
Function GetColNameFromNumber(ByRef pNum As Long) As String
```

Converts a column number into its name.
Ex: `0` ⟶ `"A"`

Input

- `pNum`: the column number (allowed values in `0..COLMAX400`)

Output

A string representation of the column name, in range `0..COLMAX400` (`'A'..'AMJ'`).
If `pNum` is not in the allowed range, returns `'?'`.
Adapted from a Python function by sundar nataraj
http://stackoverflow.com/questions/23 861 680/convert-spreadsheet-number-to-column-letter

GetSheet()

```
Function GetSheet(Optional ByRef pSheetRef As Variant, Optional pDoc As
Object) As Object
```

Returns a sheet object.

Input

- `pSheetRef`: (optional) the name or the index of the sheet to process, or an initialized sheet object.
  eg: `"Sheet1"` or `0` or `MySheet`
  Defaults to the active sheet.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The sheet object or `Null` if not found.

Usage examples

- `MySheet = GetSheet()  'curr. sheet in curr. document`
- `MySheet = GetSheet("Sheet3")  '"Sheet3" in curr. document`
- `MySheet = GetSheet(1, SomeDoc)`

`LastRowIndex()`

```
Function LastRowIndex(ByRef pColRef As Variant, Optional ByRef
pSheetRef As Variant, Optional ByRef pDoc As Object) As Long
```

Returns the index of the last row with data in a column of a sheet.

Input

- `pColRef`: the name or the index of the column to check.
  Ex: `"A"` or `0`
- `pSheetRef`: (optional) the name or the index of the sheet to process or an initialized sheet object.
  Ex: `"Sheet1"` or `0` or `MySheet`
  Defaults to the active sheet.
- `pDoc`: (optional) the document object in which to check.
  Defaults to the current document.

Output

The row index for the last used cell in the given column (`0` or value above `0`) or a negative number as an error flag.
Possible error values (see the `ERR_ROWINDEX_XXX` constants above):

- `ERR_ROWINDEX_EMPTY`: the whole column is empty.
- `ERR_ROWINDEX_UNKSHEET`: the sheet was not found.
- `ERR_ROWINDEX_UNKCOLINDEX`: the index of the column doesn't exist.
- `ERR_ROWINDEX_UNKCOLNAME`: the name of the column doesn't exist.

Adapted and edited from Martius code in
https://forum.openoffice.org/en/forum/viewtopic.php?f=20&t=10817

Usage

- `x = LastRowIndex("a", "Sheet3")`
- `x = LastRowIndex(5, "Sheet3")`
- `x = LastRowIndex(2000)`

`LastUsedCell()`

```
Function LastUsedCell(ByRef pSheet As Object) As Object
```

Returns the last used cell in a given sheet (the one at the lower-right angle).

Input

- `pSheet`: the sheet object to explore.

Output

The last used cell object in the sheet (the one at the lower-right angle).

`LastUsedColumn()`

```
Function LastUsedColumn(ByRef pSheet As Object) As Long
```

Returns the last column number used in a given sheet.

Input

- `pSheet`: the sheet object to explore.

Output

The last used column number in the sheet (the rightest one).

`LastUsedRow()`

```
Function LastUsedRow(ByRef pSheet As Object) As Long
```

Returns the last row number used in a given sheet.

Input

- `pSheet`: the sheet object to explore.

Output

The last used row number in the sheet (the lowest).

ProtectSheet()

```
Function ProtectSheet(ByRef pSheet As Object, pProtect As Boolean,
Optional pPwd As String) As Integer
```

Protects or unprotects a sheet.

Input

- `pSheet`: the object sheet to protect
- `pProtect`: True to protect, False to unprotect
- `pPwd`: (optional) the password to apply. `pPwd` may be a blank string. Defaults to a blank string.

Output

- `-1` if `pSheet` is not set (`Null`),
- the Basic error code of the action (see LibO help, 'Basic programs debugging' topic),
- `0` if the execution was successful.

ProtectSheetByName()

```
Function ProtectSheetByName(ByRef pSheetName As String, pProtect As
Boolean, pPwd As String, Optional ByRef pDoc As Object) As Integer
```

Protects or unprotects a sheet from its name.

Input

- `pSheet`: the object sheet to protect.
- `pProtect`: `True` to protect, `False` to unprotect.
- `pPwd`: the password to apply. `pPwd` may be a blank string.
- `pDoc`: (optional) the document to be processed.
  If not specified, processes the current spreadsheet.

Output

An execution result code:

- `0` if the execution was successful.
- `-1` if `pSheet` is not set (`Null`).
- the Basic error code of the action (see LibO help, 'Basic programs debugging' topic).

ShowSheetByName()

```
Sub ShowSheetByName(ByRef pSheetName As String, pVisible As Boolean,
Optional ByRef pDoc As Object)
```

Shows or hides a sheet from its name.

Input

- `pSheetName`: the name of the sheet to be hidden/displayed
- `pVisible`: `True` to show, `False` to hide
- `pDoc`: (optional) the document to be processed.
  If `pDoc` is not supplied, applies to the current spreadsheet.

CalcPrim.RangeCell

Dependencies: (none)

Macros for spreadsheet cells and ranges.

Global Constants

Constant name	Value	Description
Errors		
ERR_RANGE_ENUMERATOR	-3	The enumerator couldn't be created.
ERR_RANGE_OUTOFBOUNDS	-2	The (column/row) index is out of the range bounds.
ERR_RANGE_NOEXEC	-1	Operation not executed.
ERR_RANGE_OK	0	The operation was completed without error.
ERR_RANGE_UNK	10001	The specified range is unknown/unset.
ERR_RANGE_NONE	10002	The range is empty.
ERR_RANGE_ARRAY	10003	The specified array is invalid.
ERR_RANGE_SRCSHEET	10004	The source sheet doesn't exist.
ERR_RANGE_TGTSHEET	10005	The target sheet doesn't exist.
Range Types		
RANGETYPE_NULL	-1	The range object is Null (undefined).
RANGETYPE_UNK	0	The object type is unknown (not a range).
RANGETYPE_CELL	1	The object is a single cell.
RANGETYPE_RANGE	2	The object is a single range.
RANGETYPE_RANGES	3	The object is a multiple range.
RANGETYPE_NAMED	4	The object is a named range.
Misc.		
RANGE_NONE	-1	The range is not contained in any of the outer ranges.

Primitives

`ArrayFromVectorRangeName()`

```
Function ArrayFromVectorRangeName(ByRef pRangeName As String,
Optional pIgnoreEmpty As Boolean, Optional ByRef pDoc As
Object) As Variant
```

Creates an array from a one-dimension range name (vector).
The array is typically retrieved from a Calc range `.DataArray` property.

Input

- `pRangeName`: the range name
- `pIgnoreEmpty`: (optional) empty items won't be part of the list. Defaults to `True`
- `pDoc`: (optional) the document to query. Defaults to the current document.

Output

The array read or `Null` if the range doesn't exist.

`CalcValue()`

```
Function CalcValue(ByRef pValue As Variant) As Variant
```

Converts any value to a data type a Calc cell can store, that is, numeric or string.

Input

- `pValue`: the value to convert.

Output

The converted value into a numeric or string type, which may be stored into a Calc cell object (using `Cell.Value` property).
When a type cannot be converted (ex: object), the function result is `0`.

`CellAddressFromReference()`

```
Function CellAddressFromReference(ByRef pCellRef As String,
Optional ByRef pDoc As Object) As Object
```

Returns a cell address object from its lettered reference.

Input

- `pCellRef`: the lettered reference (`"A1"`).
- `pDoc`: (optional) a Calc document.

Output

The cell address or `Null` if an error occurred.

Adapted from Marcelly & Godard in "Programmation OpenOffice.org et LibreOffice", Paris, Eyrolles, 2011, ISBN : 978-2-212-13 247-2

`ClearRange()`

```
Function ClearRange(ByRef pRange As Object, pMode As Integer,
Optional pPwd As String) As Long
```

Clears a range contents according to the specified mode.
This function encapsulates the range `.ClearContents` method adding error checking.

Input

- `pRange`: the range object to be cleared.
- `pMode`: the erase mode to use (see the `com.sun.star.sheet.CellFlags.Xxx` LibreOffice API constants)
- `pPwd`: (optional) the owning sheet password, if any.
  Defaults to a zero-length string.

Output

The execution status. Possible error codes (see `ERR_RANGE_XXX` constants above):

- `ERR_RANGE_OK (0)`: process went `OK`.
- `ERR_RANGE_NOEXEC`: not executed (unexpected function exit).
- `ERR_RANGE_UNK`: the range was not found/is invalid. Check the range references.

`ClearRangeContents()`

```
Function ClearRangeContents(ByRef pSheetRef As Variant,
pRangeRef As Variant, pMode As Integer, Optional pPwd As
String, Optional pDoc As Object) As Long
```

Clears a range contents according to the specified mode.

Input

- `pSheetRef`: the reference of the sheet to process.
  Might be it name or its index or an initialized sheet object.
  eg: `"Sheet1"` or `0` or `MySheet`
  If the sheet reference is missing then defaults to the active sheet.
- `pRangeRef`: the reference of the range to clear.
  Might be its name (`"MyRange"` or `"A1:B12"`) or its position (`Array(0, 0, 1, 11)`) or an initialized range object.
- `pMode`: the erase mode to use (see the `com.sun.star.sheet.CellFlags.Xxx` LibreOffice API constants)
- `pPwd`: (optional) the owning sheet password, if any.
  Defaults to a zero-length string.
- `pDoc`: (optional) the document in which the operation will take place.
  Defaults to the current document.

Output

The execution status. Possible error codes (see `ERR_RANGE_XXX` constants above):

- `ERR_RANGE_NOEXEC`: not executed (unexpected function exit).
- `ERR_RANGE_OK (0)`: the operation was correctly executed.
- `ERR_RANGE_UNK`: the range was not found/is invalid. Check the range references.
- `1`: an exception has occurred (mainly: incorrect sheet password)

ClearRanges()

```
Function ClearRanges(ByRef pRangeInfo As Variant, pMode As
Integer, Optional ByRef pDoc As Object, Optional pProgress As
Object) As Long
```

Clears a set of ranges.

Input

- `pRangeInfo`: a *nested* array containing sheet reference and password (if any) and range references (see example below).
  The ranges are erased in `pRangeInfo` order.
- `pMode`: the erase mode to use (see the `com.sun.star.sheet.CellFlags.Xxx` LibreOffice API constants)
- `pDoc`: (optional) the document in which the operation will take place. Defaults to the current document.
- `pProgress`: (optional) a progress bar object.
  Defaults to `Null`.
  If a progress bar is provided, then the process can be displayed to the user.

Output

The execution status (`0` if OK, otherwise an error code)
Possible error codes (see `ERR_RANGE_XXX` constants above):

- `ERR_RANGE_NOEXEC`: not executed (unexpected function exit).
- `ERR_RANGE_OK`: the operation was correctly executed.
- `ERR_RANGE_UNK`: the range was not found/is invalid. Check the range references.
- `ERR_RANGE_ARRAY`: `pRangeInfo` is not a valid array. Check the range contents.
- `1`: an exception has occurred (mainly: incorrect sheet password)

Usage example:

```
MyMode = com.sun.star.sheet.CellFlags.DATETIME _
       + com.sun.star.sheet.CellFlags.STRING _
       + com.sun.star.sheet.CellFlags.VALUE

                        sheet     pwd    range
MyArray = Array(Array("Sheet1", "",    "Range1"), _
                Array("Sheet1", "",    "B12:C34"), _
                Array("Sheet3", "qwe", "AnotherRange"))

Result = ClearRanges(MyArray, MyMode)
```

clears the ranges in `MyArray()` for the current document, using the specified mode.

ColumnIndexFromReference()

```
Function ColumnIndexFromReference(ByRef pCellRef As String,
Optional ByRef pDoc As Object) As Long
```

Returns a column index from its lettered reference.

Input

- `pCellRef`: the lettered reference (`"A1"`).
- `pDoc`: (optional) a Calc document.

Output

The column index or `-1` if an error occurred.

Adapted from Marcelly & Godard in “Programmation OpenOffice.org et LibreOffice”, Paris, Eyrolles, 2011, ISBN : 978-2-212-13 247-2

`CopyUsedRange()`

```
Function CopyUsedRange(ByRef pSourceDoc As Object,
pSourceSheetName As String, pTargetDoc As Object,
pTargetSheetName As String, Optional ByRef pOrigin As String)
As Long
```

Copies the (formatted) range in use within a spreadsheet to another one.

Input

- `pSourceDoc`: the source document object.
- `pSourceSheetName`: the source sheet name (to copy).
- `pTargetDoc`: the target document object (may be the same document).
- `pTargetSheetName`: the target sheet name.
- `pOrigin`: (optional) the origin cell in the target sheet where to start copying the range (top-left cell coordinates).
  Defaults to `A1`.

Output

The result status for the copy process, `ERR_RANGE_OK (0)` if successful, otherwise one of the possible return codes:

- `ERR_RANGE_NOEXEC`: the code didn't execute.
- `ERR_RANGE_NONE`: the source range is empty.
- `ERR_RANGE_TGTSHEET`: target sheet not found.
- `ERR_RANGE_SRCSHEET`: source sheet not found.

`CreateCalcRangeEnumerator()`

```
Function CreateCalcRangeEnumerator(ByRef pRange As Object) As
Object
```

Creates a range enumerator.

Input

- `pRange`: the range against which to create the enumerator.

Output

The enumeration object or `Null` if not created.

`FetchInRangeColumn()`

```
Function FetchInRangeColumn(ByRef pSearch As Variant, pRange
As Object, pSearchCol As Integer, pFetchCol As Integer) As
Variant
```

Returns a value in a column of a range that satisfies a search condition in another column of that range.

Input

- `pSearch`: the searched value.
- `pRange`: the target range.
- `pSearchCol`: the column index in which to search for `pSearch` value (0-based).
- `pFetchCol`: the column index which data to return if the search succeeds (0-based).

Output

The found value or the searched value otherwise.

FormatRange()

```
Function FormatRange(ByRef pRange As Object, pFormatStr As
String, Optional pIsTemp As Boolean, Optional pDoc As Object)
As Long
```

Sets a format to a cell range and returns the status result.
The formatting uses the default locale, as defined in Tools > Options > Linguistic parameters.

Input

- `pRange`: the range to be set.
- `pFormatStr`: the format mask.
- `pIsTemp`: (optional) `True` if an unknown mask should be temporarily created, `False` otherwise.
  Defaults to `True`.
- `pDoc`: (optional) the document in which to check the formats collection.
  Defaults to the current document.

Output

The execution status (`0` if executed without error).

Usage

```
FormatRange(MySheet.Columns(5), "JJ/MM/AAAA", True)
```

Formats the whole 5th column of `MySheet` object with a date format mask of `"JJ/MM/AAAA"`. The format is searched within `ThisComponent` format collection (default) and, if not found, is temporarily created, then deleted (`True`).

GetAdjustedRange()

```
Function GetAdjustedRange(ByRef pRefRange As Object,
pTopLeftCell As Object) As Object
```

Creates a range which size is the same as a reference range, starting at a cell position.
The range has the given top-left cell as its own upper-left cell.

Input

- `pRefRange`: the reference range, to which size the target range must be set.
- `pTopLeftCell`: the top-left cell of the range to get.

Output

The adjusted range or `Null` if an error occurred.

GetDataArea()

```
Function GetDataArea(ByRef pDoc As Object, pSheetRef As
Variant, Optional pTopLeftCellAddr As Variant) As Object
```

Returns the sheet data area as a range object.
This is equivalent to hitting Ctrl + * when in the UI.

Input

- `pDoc`: the document to explore.
- `pSheetRef`: the name or the index of the sheet to process or an initialized sheet object.
  Ex: `"Sheet1"` or `0` or `MySheet`
- `pTopLeftCellAddr`: (optional) the top-left target cell address (`CellAddress` type).
  Defaults to `A1`.

Output

The data area we're looking for or `Null` if not found.

Note: this function uses the dispatcher.

`GetNamedCell()`

```
Function GetNamedCell(ByRef pName As String, Optional ByRef
pDoc As Object) As Object
```

Returns a named *single* cell object.

Input

- `pName`: the name of the searched cell
- `pDoc`: (optional) the spreadsheet object in which to search.
  If omitted, lookup within the current document.

Output

The cell object. If the cell was not found, returns `Null`.

`GetNamedCellString()`

```
Function GetNamedCellString(ByRef pName As String, Optional
ByRef pDoc As Object) As String
```

Returns a named *single* cell string contents.

Input

- `pName`: the name of the searched cell
- `pDoc`: (optional) the spreadsheet object in which to search.
  If omitted, lookup within the current document.

Output

The cell string contents. If the cell was not found, returns an empty string.

`GetNamedCellValue()`

```
Function GetNamedCellValue(ByRef pName As String, Optional
ByRef pDoc As Object) As Variant
```

Returns a named *single* cell contents value.

Input

- `pName`: the name of the searched cell
  The name is either a reference (`A1`) or a user-defined name (`myCell`)
- `pDoc`: (optional) the spreadsheet object in which to search.
  If omitted, lookup within the current document.

Output

The cell contents value. If the cell was not found, or if the cell named provided refers to a larger range, returns `Null`.

`GetNamedRange()`

```
Function GetNamedRange(ByRef pSheetName As String, pRangeName
As String, Optional ByRef pDoc As Object) As Object
```

Returns a range object from a sheet name and the range name.

Input

- `pSheetName`: the sheet name
- `pRangeName`: the searched range name.
  The range name is a user-defined name (ex. `MyRange`).
  The range name may apply to any range type: single cell, single range, multiple range or named range.
- `pDoc`: (optional) the document object.
  Defaults to the current spreadsheet.

Output

The range object or `Null` if not found.

GetRange()

```
Function GetRange(Optional pRangeRef As Variant, Optional
ByRef pSheetRef As Variant, Optional pDoc As Object) As
Object
```

Returns a range object.

Input

- `pRangeRef`: (optional) the reference of a range.
  Might be its name (`"MyRange"` or `"A1:B12"`) or its position (`Array(0, 0, 1, 11)`), or a `RangeAddress` UNO struct or an initialized range object.
  Defaults to the current range.
- `pSheetRef`: (optional) the reference of the sheet to explore.
  Might be its name or its index, or an initialized sheet object.
  eg: `"Sheet1"` or `0` or `MySheet`
  Defaults to the active sheet.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The range object or `Null` if not found

Usage

```
MyRange = GetRange()
MyRange = GetRange("Sheet3", "SomeRange")    'named range
MyRange = GetRange(Array(0, 0, 1, 11), 1, SomeDoc)  'A1:B12 in
first sheet of SomeDoc
```

GetRangeColumn()

```
Function GetRangeColumn(ByRef pRange As Object, pColNum As
Integer) As Object
```

Returns a column range object from a range.

Note: a range `Columns` property returns all 1,024,000 rows of the sheet. This function returns only the actual used height of the range column.

Input

- `pRange`: the range from which we want a column
- `pColNum`: the column number (1-based)

Output

The column range object.

GetRangeFromColumns()

```
Function GetRangeFromColumns(ByRef pSheet As Object,
pStartColumn As Integer, Optional pEndColumn As Integer) As
Object
```

Returns a cell range that includes whole columns.

Input

- `pSheet`: the sheet to examine.
- `pStartColumn`: the starting column (0-based).
- `pEndColumn`: (optional) the ending column (0-based).
  Defaults to `pStartColumn`.

Output

A cell range encompassing all columns from `pStartColumn` to `pEndColumn` or `Null` if an error occurred.

Adapted from Silas in http://ooo-forums.apache.org/en/forum/viewtopic.php?f=20&t=58763

GetRangeFromRows()

```
Function GetRangeFromRows(ByRef pSheet As Object, pStartRow
As Integer, Optional pEndRow As Integer) As Object
```

Returns a cell range that includes whole rows.

Input

- `pSheet`: the sheet to examine.
- `pStartRow`: the starting row (0-based).
- `pEndRow`: (optional) the ending row (0-based).
  Defaults to `pStartRow`.

Output

A cell range encompassing all rows from `pStartRow` to `pEndRow` or `Null` if an error occurred.

Adapted from Silas in http://ooo-forums.apache.org/en/forum/viewtopic.php?f=20&t=58763

GetRangeRow()

```
Function GetRangeRow(ByRef pRange As Object, pRowNum As
Integer) As Object
```

Returns a row range object from a range.

Note: a range `Rows` property returns all 1,024 columns of the sheet. This function returns only the actual used width of the row.

Input

- `pRange`: the range from which we want a column
- `pRowNum`: the row number (1-based)

Output

The row range object.

GetRangeType()

```
Function GetRangeType(ByRef pRange As Object) As Integer
```

Returns a range type.

Input

- `pRange`: the range to check

Output

The range type (see `RANGETYPE_XXX` constants)

GotoLastCell()

```
Sub GotoLastCell(ByRef pSheet As Object, pSearchColNum As
Integer, pColNum As Integer, Optional ByRef pDoc As Object)
```

Searches the last data cell in a given column of a sheet, then moves the active cell on the same row to another column of the same sheet.

Input

- `pSheet`: the sheet object to process.
- `pSearchColNum`: the column number to search for absence of data (0-based: 0 = A; 1 = B; etc.).
- `pColNum`: the column number to reach.
  `pColNum` may be the same as `pSearchColNum`.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

IsRangeInRange()

```
Function IsRangeInRange(ByRef pInnerRange As Object,
pOuterRange As Object) As Boolean
```

Tests whether a range is *fully* contained within another.

Input

- `pInnerRange`: the inner cell range.
- `pOuterRange`: the outer cell range.

Output

`True` if `pInnerRange` in fully included within `pOuterRange`, otherwise `False`.

Note: Applies only to single ranges.

IsRangeInRanges()

```
Function IsRangeInRanges(ByRef pInnerRange As Object,
pOuterRanges As Variant) As Integer
```

Tests whether a range is fully contained within another from a range set (multi-range array).

Input

- `pInnerRange`: the inner cell range.
- `pOuterRanges`: the outer cell ranges array.
  This array may be obtained using:
  ```
  Const THERANGES = "Sheet1.A3:B4;Sheet4.C10;Sheet1.H2:K157"
  ARangesArray =
  MyDocument.Sheets.getCellRangesByName(THERANGES)
  ```

Output

The outer range array index in which the inner one is contained or `-1` (`RANGE_NONE`) if it is not contained in any.

PasteSpecial()

```
Sub PasteSpecial(ByRef pSrcDoc As Object, pSrcRange As
Object, pTgtDoc As Object, pTgtRange As Object)
```

Applies the **Edition > Paste special** command to a given source range towards a given target range.
This only copies text, dates and numerical values (without formatting).

Input

- `pSrcDoc`: the source document object.
- `pSrcRange`: the range to be copied.
- `pTgtDoc`: the target document object.
- `pTgtRange`: the range in which the copy must be processed.

Caution

This sub uses the dispatcher. This implies that it can only process visible documents.

PasteTransferable()

```
Sub PasteTransferable(ByRef pSrcDoc As Object, pSrcRange As
Object, pTgtDoc As Object, pTgtRange As Object)
```

Pastes a range into another one. The formatting is retained.

Input

- `pSrcDoc`: the source document object.
- `pSrcRange`: the range to be copied.
- `pTgtDoc`: the target document object.
- `pTgtRange`: the range in which the copy must be processed.

Caution

This sub can only process visible documents.

`RangeAddressFromReference()`

```
Function RangeAddressFromReference(ByRef pRangeRef As String,
Optional ByRef pDoc As Object) As Object
```

Returns a range address from its lettered reference.

Input

- `pRangeRef`: the lettered range reference (`"A1"` or `"A1:B2"`).
- `pDoc`: (optional) a Calc document.

Output

The range address object or `Null` if an error occurred.

Adapted from Marcelly & Godard in "Programmation OpenOffice.org et LibreOffice", Eyrolles, Paris 2011, ISBN 978-2-212-13 247-2

`RangeAddrString()`

```
Function RangeAddrString(ByRef pDocument As Object,
pRangeAddr As Object) As String
```

Returns the string for a range address

Input

- `pDocument`: the document
- `pRangeAddr`: the range being queried

Output

A string representing the range address. This string is empty if any error occurred.

`RangeAsSheetCellRange()`

```
Function RangeAsSheetCellRange(ByRef pRange As Object) As
Object
```

Returns a `SheetCellRange` object from any compatible range type.

Input

- `pRange`: the range object to be converted

Output

A range that supports the `SheetCellRange` service (`"com.sun.star.sheet.SheetCellRange"`) or `Null` if no conversion was possible.
This allows to process ranges uniformly.

`RangeColumnToVector()`

```
Function RangeColumnToVector(ByRef pRange As Object, pColNum
As Integer) As Variant
```

Returns a range column contents as a 1D array (vector).

Input

- `pRange`: the range to process.
- `pColNum`: the column number to read (0-based).

Output

A 1D array of the values contained within the wanted column or `Null` if an error occurred.

SetActiveCellByName()

```
Function SetActiveCellByName(ByRef pCellName As String, ByRef
pSheetName As String, Optional ByRef pDoc As Object) As
Object
```

Sets a cell as active.

Input

- `pCellName`: the cell name to activate.
- `pSheetName`: the sheet name in which to activate a cell.
- `pDoc`: (optional) the document to process.
  If `pDoc` is not specified, processes the current spreadsheet.

Output

The cell object that was activated or `Null` if it doesn't exist.

SetActiveSheetByName()

```
Function SetActiveSheetByName(ByRef pSheetName As String,
Optional pDoc As Object) As Object
```

Sets a sheet as active.

Input

- `pSheetName`: the sheet name to activate.
- `pDoc`: (optional) the document to process.
  If `pDoc` is not specified, processes the current spreadsheet.

Output

Returns the corresponding Sheet object or `Null` if it doesn't exist/wasn't activated.

SetNamedCellValue()

```
Sub SetNamedCellValue(ByRef pName As String, pValue As
Variant, Optional ByRef pDoc As Variant)
```

Sets a single named cell value.
If the named cell is not found, does nothing.

Input

- `pName`: the name of the searched cell.
- `pValue`: the value to set.
- `pDoc`: (optional) the spreadsheet object in which to search.
  If omitted, lookup within the current document.

ShiftRange()

```
Function ShiftRange(ByRef pOldRange As Object, pNewTLCell As
Object) As Object
```

Returns a new range from a given range, shifted to a specified top-left cell.
Both ranges are of the same size. The new range is on the same sheet as the top-left cell.

Input

- `pOldRange`: the template range.
- `pNewTLCell`: the new top-left cell for the wanted range.

Output

The new range on the same sheet as the specified top-left cell, or `Null` if an error occurred.

`UpdateRangeColumnValues()`

```
Function UpdateRangeColumnValues(ByRef pSearch As Variant,
pRange As Object, pLookupIndex As Integer, pUpdateIndex As
Integer, pValue As Variant) As Long
```

Updates all cell values in a range column that meet a search criterion.

Input

- `pSearch`: the searched value.
- `pRange`: the range in which to search.
- `pLookupIndex`: the lookup column (1-based) within the search range. This index must be within `pRange` bounds (1 to columns count).
- `pUpdateIndex`: the update column (1-based) within the range. This index must be within `pRange` bounds (1 to columns count).
- `pValue`: the value to insert into the matching cells.

Output

The result code.
Possible return values:

- `ERR_RANGE_OK (0)`: process completed without error.
- `ERR_RANGE_OUTOFBOUNDS`: the lookup index is out of range.
- `ERR_RANGE_ENUMERATOR`: the range enumerator couldn't be created.
- (other): the Basic error code.

Note: Depending on the range height, the update process may be lengthy. It is recommended that Calc controllers be locked in the interval.

Usage

`Result = UpdateRangeColumnValues(25, MyRange, 1, 27, 5)` replaces column `27` items with value `5` into `MyRange` range where column `1` items equal `25`.
See also: `UpdateRangeMultiColumnValuesArray()`.

UpdateRangeMultiColumnValues()

```
Function UpdateRangeMultiColumnValues(ByRef pSearch As
Variant, pRange As Object, pLookupIndex As Integer,
pUpdateValues As Variant) As Long
```

Updates all cells in a range columns that match a criterion in another column of that range.

Note: Do *not* use this function if you want to update cell *data*. Use it to update other cell items (formulas or settings), otherwise use the `UpdateRangeMultiColumnValuesArray()` function.
Note that, because it uses an enumerator object, `UpdateRangeMultiColumnValues()` is *much* slower than `UpdateRangeMultiColumnValuesArray()` (uses a data array).

Input

- `pSearch`: the searched value in the range.
- `pRange`: the selected range that contains data to search.
- `pLookupIndex`: the lookup column (1-based) within the range.This index must be within `pRange` bounds (1 to columns count).
- `pUpdateValues`: the update columns array (indices are 1-based) within the range. This is a nested array with (col_index, value, mode) array items (see Usage below).
  The column indices must all be within `pRange` bounds (1 to columns count).

Mode: The Calc update mode. This value is one of the `com.sun.star.sheet.CellFlags.XXX` constants.
The currently supported constants are: `FORMULA`, `VALUE`, `DATETIME`, `STRING`.

Output

The process result code. Possible return values:

- `ERR_RANGE_OK` (0): process completed without error.
- `ERR_RANGE_OUTOFBOUNDS`: the lookup or update index is out of range.
- `ERR_RANGE_ENUMERATOR`: the enumerator couldn't be created.
- `ERR_RANGE_BADSRCARRAY`: the update array is not usable.
- (other): the Basic error code

Note: depending on the range height, the update process may be lengthy. It is recommended that Calc controllers be locked in the interval.

Usage

```
Mode1 = com.sun.star.sheet.CellFlags.VALUE
Mode2 = com.sun.star.sheet.CellFlags.FORMULA
Formula = "=SUM(A1:A5)"
Result = UpdateRangeMultiColumnValues(10, MyRange, 1,
Array(Array(27, 1, Mode1), Array(28, Formula, Mode2)))
```

replaces column `27` items with value `1` and column `28` items with a formula into `MyRange` range where column `1` items equal `10`.

`UpdateRangeMultiColumnValuesArray()`

```
Function UpdateRangeMultiColumnValuesArray(ByRef pSearch As
Variant, pRange As Object, pLookupIndex As Integer,
pUpdateValues As Variant) As Long
```

Updates all cells in a set of range columns that match a criterion in another column of that range.

Note: Use this function only if you want to update *cell data*.
For other cell items (formulas or settings), use the `UpdateRangeMultiColumnValues()` function above.
Note that because `UpdateRangeMultiColumnValuesArray()` uses the range `DataArray` property, it is *much* faster than `UpdateRangeMultiColumnValues()`.

Input

- `pSearch`: the searched value in the range.
- `pRange`: the selected range that contains data to search.
- `pLookupIndex`: the lookup column (1-based) within the range.
  This index must be within `pRange` bounds (1 to columns count).
- `pUpdateValues`: the update columns array (indices are 1-based) within the range.
  This is a nested array with (`ColumnIndex`, `Value`) array items.
  The column indices must all be within `pRange` bounds (1 to columns count).

Note: This function does its best to convert the provided values into admissible Calc values types (numbers and strings only).

Output

The result code. Possible return values:

- `ERR_RANGE_OK` (0): process completed without error.
- `ERR_RANGE_OUTOFBOUNDS`: the lookup or update index is out of range.
- `ERR_RANGE_BADSRCARRAY`: the update array is not usable.
- (other): the Basic error code

Note: depending on the range height, while this function is fast, the update process may be lengthy. It is recommended that Calc controllers be locked in the interval.

Usage

```
Result = UpdateRangeMultiColumnValuesArray(SomeValue, MyRange,
1, Array(Array(27, True), Array(28, SomeDate)))
```

This replaces column `27` items with value `True` and column `28` items with value `SomeDate` into the `MyRange` range where column `1` items equal `SomeValue`.

`UsedRange()`

```
Function UsedRange(ByRef pSheet As Object, Optional ByRef
pOrigin As String) As Object
```

Returns the used cells range in a given sheet.

Input

- `pSheet`: the sheet object to explore.
- `pOrigin`: (optional) the top-left cell from which to get the range.
  Defaults to `A1`.

Output

The used range object in the sheet.

`VLookupCell()`

```
Function VLookupCell(ByRef pSearch As Variant, pRange As
Object, pLookupIndex As Integer, pMatchType As Integer) As
Object
```

This function is much alike the `VLOOKUP()` Calc function, but it returns a cell object instead of a value.

Input

- `pSearch`: the searched value
- `pRange`: the range that contains data to search. The searched column must be the first one.
- `pLookupIndex`: the lookup column (1-based) within the search range
- `pMatchType`: `-1`, `0` or `1` (see the `MATCH()` function help).

Output

The matching cell object or `Null` if none found.

CalcPrim.Document

Global Constants

Constant name	Value	Description
`Errors`		

Primitives

`SecureCalcUI()`

```
Sub SecureCalcUI(ByRef pSecure As Boolean, Optional ByRef pDoc As
Object)
```

(un)secures the Calc UI against user's actions.

Input

- `pSecure`: `True`: secures the interface, `False`: releases the UI security.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Warning: this sub (un)locks controllers and action locks.
Thus, any call to `SecureCalcUI(True)` *must* be followed with another call with `False` as an argument.

Note: A second call with `True` as an argument is ignored.

CalcPrim.CalcExportClass

Dependencies: `CalcPrim.CalcExportPropertyClass`

This class module defines a spreadsheet export management class. It uses `CalcPrim.CalcExport-PropertyClass` as a sheet property source.

This class eases data backup or export from a given Calc document to another.

Note that this class is currently under heavy refactoring.

Constants

All constants have a `CALCEXPORT` prefix. Moreover, error constants are `ERR_` prefixed.

Constant name	Value	Description
Error Constants		
ERR_CALCEXPORTCLASS_NORUN	-1	The process did not run.
ERR_CALCEXPORTCLASS_NONE	0	No error (OK).
ERR_CALCEXPORTCLASS_CANTCREATEFILE	1	The output file can't be created.
ERR_CALCEXPORTCLASS_CANTOPENSOURCE	2	The source spreadsheet can't be opened.
ERR_CALCEXPORTCLASS_SOURCENOTFOUND	3	The source file wasn't found.
ERR_CALCEXPORTCLASS_CANTWRITEFILE	4	Can't write to output file.
ERR_CALCEXPORTCLASS_NOTASPREADSHEET	5	The source document is not a spreadsheet.
ERR_CALCEXPORTCLASS_SOURCEUNDEFINED	6	The source is undefined.
ERR_CALCEXPORTCLASS_TARGETUNDEFINED	7	The target is undefined.
ERR_CALCEXPORTCLASS_NOTHINGTOEXPORT	8	There's nothing to export.
ERR_CALCEXPORTCLASS_CUSTOMPROP	9	Can't create a custom property.
Other constants		
(none)		

Properties

Public Properties		
Self	Object [WO]	Points to itself.
Application	String [R/W]	The source application name. Can be used for backup checking at restore time.
LastError	Long [RO]	Stores the last error code while running the object. See the error constants above.
Progress	Object [R/W]	Connection with a progress bar for UI user feedback. Use a ProgressWidgetClass object for that. If it is present, this widget is updated when processing each sheet. See the FormPrim.ProgressWidgetClass module.
SheetInformation	Object [RO]	The collection of sheet export properties. See the `CalcPrim.CalcExportPropertyClass` below for details about the sheets export properties.

`SourceDocument`	Object [R/W]	The document from which to export (must be a spreadsheet).
`TargetDocumentName`	String [R/W]	The target document FQN. Caution: The extension is not automatically added, thus you must specify one, if needed.
`Version`	String [R/W]	Specify a version property for the target export `.ods` file.

Methods

Public Methods

`AddAllSheets()`

```
Public Sub AddAllSheets()
```

Adds all sheets of the source document to the selection.

`AddSheet()`

```
Public Sub AddSheet(ByRef pSheetRef As Variant)
```

Add a given sheet to the selection.

Input

- `pSheetRef`: the sheet reference; it is the name or the index of the sheet or an initialized sheet object
  eg: `"Sheet1"` or `0` or `MySheet`.

`AddSheetInfo()`

```
Public Sub AddSheetInfo(ByRef pSheetInfo As Object)
```

Add a `SheetInfo` object to the selection.
This is the preferred way of feeding the exporter.

Input

- `pSheetInfo`: the sheet selection as an object of class `CalcExportPropertyClass`
  If the sheet name is already present in the collection, the new one silently replaces the previous one.

`ClearError()`

```
Public Sub ClearError()
```

Clears the error flag.

`ClearSelection()`

```
Public Sub ClearSelection()
```

Clears the sheet information collection.

`Execute()`

```
Public Sub Execute()
```

Exports to the target document, from the sheet property items.
The execution process is:

1. Create a hidden Calc document,
2. Scan the selection and export each sheet, using this sheet settings,
3. Add information to the document properties (source file name, date/time and version, application name).
4. Then store the document to disk and close it.

GetSheetInformation()

```
Public Function GetSheetInformation(ByRef pSheetName As String) As
Object
```

Returns the sheet selection information.

Input
- `pSheetName`: the name of the sheet for which we want the information

Output
The sheet information object for that sheet, or `Null` if not found.

Reset()

```
Public Sub Reset()
```

Resets all export definitions.

Private Methods

`_AddCollectionItem()`

```
Private Sub _AddCollectionItem(ByRef pColl As Object, ByRef pItem As
Variant, pKey As String)
```

Adds an item to a collection.
If the key is already present, the existing item is replaced with the new one.

Input
- `pColl`: the collection for storing the item.
- `pItem`: the item to add.
- `pKey`: the key for the item.

`_AddSheet()`

```
Private Sub _AddSheet(ByRef pDoc As Object, pSheetName As String)
```

Adds a sheet to a document.

Input
- `pDoc`: the Calc document to process.
- `pSheetName`: the name for the sheet to insert.

`_ArrayExists()`

```
Private Function _ArrayExists(ByRef pArray As Variant) As Boolean
```

Checks whether an array exists (is dimensioned) and can be manipulated.

Input
- `pArray`: the array to check.

Output
`True` if `pArray` is a valid variable otherwise `False`.
See the `ArrayExists()` function in the `ArrayPrim.Arrays` module.

`_CheckExport()`

```
Private Function _CheckExport() As Long
```

Checks whether the export process can start.

Ouput
Returns a status code:
- `ERR_CALCEXPORTCLASS_NORUN`: (–1) no check was made.
- `ERR_CALCEXPORTCLASS_NONE`: (0) ready to go.
- `ERR_CALCEXPORTCLASS_SOURCEUNDEFINED`: source document not set.
- `ERR_CALCEXPORTCLASS_TARGETUNDEFINED`: target name not set.
- `ERR_CALCEXPORTCLASS_NOTHINGTOEXPORT`: no sheet selected for export.

`_CheckSheetName()`

```
Private Function _CheckSheetName(ByRef pSheetName As String,
pSheetNames As Variant) As String
```

(recursive)
Checks whether a sheet name already exists in a names list and returns the name set so that there's no duplicate.

Input

- `pSheetName`: the tested sheet name.
- `pNames`: a vector (1-D array) of existing names against which to check. The vector must exist.

Output

The sheet name, defined so that there's no name collision. If the sheet name exists, it is appended a number (`1..n`) until there's no more collision.

`_CleanUpTargetDocument()`

```
Private Sub _CleanUpTargetDocument(ByRef pDoc As Object)
```

Final ancillary processes on the target document.

Input

- `pDoc`: the target document to process.

`_CopyData()`

```
Private Sub _CopyData(ByRef pSheetInfo As Object, ByRef pTgtDoc As
Object)
```

Actually copy data from the source document sheets to the target document.

Input

- `pSheetInfo`: the sheet selection information
- `pTgtDoc`: the target document (same sheet name)

`_CreateCalcDocument()`

```
Private Function _CreateCalcDocument() As Object
```

Creates a hidden spreadsheet document and returns that object.

Output

The in-memory Calc document object, or `Null` if an error occurred.

`_CreateCustomProperties()`

```
Private Function _CreateCustomProperties(ByRef pTgtDoc As Object,
pVersion As String, pDateTime As Date, pSource As String) As Long
```

Creates the custom properties in the target `.ods` file.

Input

- `pTgtDoc`: the export target document.
- `pVersion`: the version string to add.
- `pDataTime`: the date and time to add.
- `pSource`: the export source file name.

Output

A status code.

`_EmptyColumns()`

```
Private Sub _EmptyColumns(ByRef pSheet As Object, pArray As Variant)
```

Empties columns in the sheet

Input

- `pSheet`: the sheet object to process.
- `pArray`: the empty column information array.

`_EmptyRows()`

```
Private Sub _EmptyRows(ByRef pSheet As Object, pArray As Variant)
```

Empties sheet rows.

Input

- `pSheet`: the sheet object to process.
- `pArray`: the empty row information array

`_GetRange()`

```
Function _GetRange(Optional ByRef pSheetRef As Variant, Optional
pRangeRef As Variant, Optional pDoc As Object) As Object
```

Returns a range object.

Input

- `pSheetRef`: (optional) the reference of the sheet to explore.
  May be its name or its index, or an initialized sheet object.
  ex: `"Sheet1"` or `0` or `MySheet`
  Defaults to the active sheet.
- `pRangeRef`: (optional) the reference of a range.
  May be its name (`"MyRange"` or `"A1:B12"`) or its position (`Array(0, 0, 1, 11)`), or an initialized range object.
  Defaults to the current range.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The range object or `Null` if not found.

Usage examples

```
MyRange = GetRange()
```
→ the current range in the current sheet of the current document
```
MyRange = GetRange("SomeRange", "Sheet3")
```
→ named range
```
MyRange = GetRange(1, Array(0, 0, 1, 11), SomeDoc)
```
→ `A1:B12` in first sheet of `SomeDoc`

`_GetSheet()`

```
Function _GetSheet(Optional ByRef pSheetRef As Variant, Optional
pDoc As Object) As Object
```

Returns a sheet object.

Input

- `pSheetRef`: (optional) the name or the index of the sheet to process or an initialized sheet object
  ex: `"Sheet1"` or `0` or `MySheet`
  Defaults to the active sheet.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The sheet object or `Null` if not found

Usage examples

```
MySheet = GetSheet()
```
→ current sheet for current document
```
MySheet = GetSheet("Sheet3")
```
→ sheet named `"Sheet3"` in current document
```
MySheet = GetSheet(1, SomeDoc)
```
→ 2nd sheet in `SomeDoc` document object

`_GetUsedRange()`

```
Private Function _GetUsedRange(ByRef pDoc As Object, pSheetRef As
Variant) As Object
```

Returns the sheet used area as a range object.

Input

- `pDoc`: the document to explore.
- `pSheetRef`: (optional) the name or the index of the sheet to process or an initialized sheet object
  Ex: `"Sheet1"` or `0` or `MySheet`

If the sheet reference is missing then defaults to the active sheet.

Output

The used range object or `Null` if an error occurred.

`_InsertMetadata()`

```
Private Function _InsertMetadata(ByRef pDoc As Object) As long
```

Inserts the export metadata into the target document and returns a result status. The metadata are the version, date/time, source file name and application name of the export process.

Input

- `pDoc`: the document in which to insert the metadata.

Output

A result status.

`_ProcessSheets()`

```
Private Sub _ProcessSheets(ByRef pTgtDoc As Object)
```

Exports the sheets selection to another Calc document.

Input

- `pTgtDoc`: the target Calc document.

`_RemoveColumns()`

```
Private Sub _RemoveColumns(ByRef pSheet As Object, pArray As
Variant)
```

Removes columns from a sheet.

Input

- `pSheet`: the sheet object to process.
- `pArray`: the deletion information array.
  /!\ This array is sorted in reverse order (descending)!

`_RemoveRows()`

```
Private Sub _RemoveRows(ByRef pSheet As Object, pArray As Variant)
```

Removes rows from a sheet.

Input

- `pSheet`: the sheet object to process.
- `pArray`: the deletion information array.
  /!\ This array is sorted in reverse order (descending)!

`_ReplaceStr()`

```
Private Function _ReplaceStr(ByRef pSourceStr As String, pSearchStr
As String, pReplStr As String) As String
```

Replaces a searched string by a replacement string within a source string and returns the result.
See: `ReplaceStr()` in `StringsPrim.Strings` module.

`_SaveTargetDocument()`

```
Private Function _SaveTargetDocument(ByRef pDoc As Object) As Long
```

Saves the in-memory document to the target file name and returns the result status.

CalcPrim.CalcExportPropertyClass

Dependencies: (none)

This module defines an export property to use with the previous `CalcPrim.CalcExportClass`.

Properties

Public Properties		
`EmptyColumns`	`Variant [R/W]`	This property defines the list of columns that will be present but empty in the target document. The list is 0-based. [W] Set the empty columns list from a 1-D array (vector). For practical reasons, the vector may hold discrete values or ranges: `EmptyColumns(Array(1, 2-5, 15))` [R] Get the `EmptyColumns` array.
`EmptyRows`	`Variant [R/W]`	This property defines the list of rows that will be present but empty in the target document. The list is 0-based. [W] Set the empty rows list from a 1-D array (vector). For practical reasons, the vector may hold discrete values or ranges: `EmptyRows(Array(1-3))` [R] Get the `EmptyRows` array.
`RemoveColumns`	`Variant [R/W]`	This property defines the list of columns that will be removed from the target document. The list is 0-based. [W] Set the removed columns list from a 1-D array (vector). For practical reasons, the vector may hold discrete values or ranges: `RemoveColumns(Array(0))` [R] Get the `RemoveColumns` array.
`RemoveRows`	`Variant [R/W]`	This property defines the list of rows that will be removed from the target document. The list is 0-based. [W] Set the removed rows list from a 1-D array (vector). For practical reasons, the vector may hold discrete values or ranges: `RemoveRows(Array(1, 2))` [R] Get the `RemoveRows` array.
`Self`	`Object [WO]`	Self reference.
`SourceRangeAddress`	`Object [R/W]`	This property defines a range to export within the source sheet. [W] Set the range address using a `RangeAddress` object. Note that the `RangeAddress.Sheet` property is not used, so its value in of no importance. [R] Get the source range address as a `RangeAddress` object.
`SourceSheetName`	`String [R/W]`	This property defines the name of the source sheet. This information is unique and identifies the properties, which means that the same sheet cannot be exported twice (the newest added properties for a given sheet would replace the previous ones).. [W] Set the source sheet name. If the target name is not set, then the same name is assumed. [R] Get the `SourceSheetName` string.
`TargetSheetName`	`String [R/W]`	This property defines the name of the source sheet which can differ from the source sheet name. [W] Set the target sheet name. [R] Get the `TargetSheetName` string.
Private Properties (none)		

Methods

Public Methods	
AddEmptyColumn()	`Public Sub AddEmptyColumn(ByRef pItem As Variant)`
AddEmptyRow()	`Public Sub AddEmptyRow(ByRef pItem As Variant)`
AddRemoveColumn()	`Public Sub AddRemoveColumn(ByRef pItem As Variant)`
AddRemoveRow()	`Public Sub AddRemoveRow(ByRef pItem As Variant)`
ClearEmptyColumns()	`Public Sub ClearEmptyColumns()`
ClearEmptyRows()	`Public Sub ClearEmptyRows()`
ClearIgnoredColumns()	`Public Sub ClearIgnoredColumns()`
ClearIgnoredRows()	`Public Sub ClearIgnoredRows()`
ClearRangeAddress()	`Public Sub ClearRangeAddress()`
Private Methods	
`_AddToVector()`	`Private Sub _AddToVector(ByRef pArray As Variant, pItem as Variant)` Adds an item to a single-dimension array.

`_ArrayExists()`	`Private Function _ArrayExists(ByRef pArray As Variant) As Boolean`
	Checks whether an array exists (is dimensioned) and can be manipulated.
	Input
	• `pArray`: the array to check
	Output
	`True` if `pArray` is a valid variable otherwise `False`

`_SortVector()`

```
Private Function _SortVector(ByRef pArray As Variant, Optional ByRef
pAsc As Boolean)
```

Sort vector data using bubble sort (case sensitive).

Input

- `pArray`: the vector to sort.
  The input array must exist.
- `pAsc`: (optional) `True` if the sort must be in ascending order, otherwise `False`.
  Defaults to `True`.

Output

The sorted array.
Adapted from https://helloacm.com/bubble-sort-in-vbscript/

`_SwapValues()`

```
Private Sub _SwapValues(ByRef pVal1 As Variant, pVal2 As Variant)
```

Swaps the two values `pVal1` and `pVal2`.

CalcPrim.CalcImportClass

Dependencies: `CalcPrim.CalcImportPropertyClass`

This class module defines a spreadsheet import management class. It uses `CalcPrim.CalcImportPropertyClass` as a sheet property source.

This class eases data restoration or import from a given Calc document to another.

This class is currently under heavy refactoring!

Constants

All constants have a `CALCIMPORT` prefix. Moreover, error constants are `ERR_` prefixed.

Constant name	Value	Description
Error Constants		
`ERR_CALCIMPORTCLASS_NORUN`	`-1`	The process did not run.
`ERR_CALCIMPORTCLASS_NONE`	`0`	No error (OK).
`ERR_CALCIMPORTCLASS_CANTCREATEFILE`	`1`	The output file can't be created.
`ERR_CALCIMPORTCLASS_CANTOPENSOURCE`	`2`	The source spreadsheet can't be opened.
`ERR_CALCIMPORTCLASS_SOURCENOTFOUND`	`3`	The source file wasn't found.
`ERR_CALCIMPORTCLASS_CANTWRITEFILE`	`4`	Can't write to output file.
`ERR_CALCIMPORTCLASS_NOTASPREADSHEET`	`5`	The source document is not a spreadsheet.
`ERR_CALCIMPORTCLASS_SOURCEUNDEFINED`	`6`	The source is undefined.
`ERR_CALCIMPORTCLASS_TARGETUNDEFINED`	`7`	The target is undefined.
`ERR_CALCIMPORTCLASS_NOTHINGTOIMPORT`	`8`	There's nothing to import.
Other constants		
`(none)`		

Properties

Public Properties		
`Self`	Object [WO]	Points to itself.
`DocChecker`	Object [RO]	The (optional) document check object. See LibOPrim.DocCheckerClass.
`LastError`	Long [RO]	The last error encountered.
`Progress`	Object [R/W]	A progress bar information object for display purposes.
`SheetInformation`	Object [RO]	A collection of all sheet information for import.
`SourceDocType`	Byte [R/W]	The source document type to check. Possible values: See LibOPrim.DocCheckerClass `DOCCHKCLASS_DOCTYPE_XXX` constants. Note The default `DOCCHKCLASS_DOCTYPE_NONE` bypasses the document type checking.
`SourceDocument`	Object [RO]	The source document object.

`SourceDocumentName`	String [R/W]	the source document name (FQN), in URL or OS form.
`TargetDocument`	Object [R/W]	The target calc document object to which to import.

Methods

Public Methods

Private Methods

CalcPrim.CalcImportPropertyClass

Dependencies: (none)

This module defines an import property to use with the previous `CalcPrim.CalcImportClass`.

Properties

Public Properties
Private Properties (none)

Methods

Public Methods	
Private Methods	

WriterPrim

LibreOffice Writer module primitives.

WriterPrim.Styles

Dependencies: (none)

Global Constants

Constant name	Value	Description
`Style families`		
`STY_WFAMPAGES`	`"PageStyles"`	Style family for pages.
`STY_WFAMPARAS`	`"ParagraphStyles"`	Style family for paragraphs.
`STY_WFAMCHARS`	`"CharacterStyles"`	Style family for characters.
`STY_WFAMFRAMES`	`"FrameStyles"`	Style family for frames.
`STY_WFAMNUMBER`	`"NumberingStyles"`	Style family for numbering and bullets.
`STY_WFAMTABLES`	`"TableStyles"`	Style family for tables. Was added since LibreOffice v.5.3.

Primitives

`GetStyleAtCursor()`

```
Function GetStyleAtCursor(ByRef pTextCursor As Object, pStyleFamily As
String) As Object
```

Returns the style object for the text cursor.

Input

- `pTextCursor`: a text cursor object.
- `pStyleFamily`: the style family to check.

Output

The style object for the family the text cursor is in, or `Null` if the style is not found or if the text cursor is not set.

WriterPrim.Fields

Masterfields (aka user fields) management.

Dependencies: (none)

Masterfields are stored in a container object (`Document.TextfieldMasters`). Their names can be read from its `ElementNames()` array, where each name is stored with the following syntax:

- User-defined masterfields: `"com.sun.star.text.fieldmaster.User.SomeMasterFieldName"`
- Sequences: `"com.sun.star.text.fieldmaster.SetExpression.SomeSequenceName"`

Global Constants

Constant name	Value	Description
Services		
MFLD_USERSERVICE	`"com.sun.star.text.fieldmaster.User."`	Root name for a masterfield, incl. the final dot.
MFLD_USERINSTANCE	`"com.sun.star.text.fieldmaster.User"`	Root name for a masterfield, not incl. the final dot.
Master field types		
MFLD_TYPE_UNK	`0`	Unknown type
MFLD_TYPE_USER	`1`	User-created field
MFLD_TYPE_EXPR	`2`	LibO sequence field (Illustration, Table, Text or Drawing)
Master field identifiers		
MFLD_TYPEUSERID	`".User."`	User field.
MFLD_TYPEEXPRID	`".SetExpression."`	Sequence field.

Primitives

CreateMasterField()

```
Function CreateMasterField(ByRef pFieldName As String, Optional pValue
As Variant, Optional pDoc As Object) As Boolean
```

Creates a master field.

Input

- `pFieldName`: the user field name
- `pValue` (optional): if provided, sets the initial value for the created master field.
- `pDoc` (optional): the document in which the master field is created.
  If not specified, the current document is assumed.

Output

`True` if the operation was successful otherwise `False`.

DeleteMasterField()

```
Function DeleteMasterField(ByRef pFieldName As String, Optional pDoc As
Object) As Boolean
```

Deletes a master field.

Input

- `pFieldName`: the user field name
- `pDoc`: the document in which the master field is created.
  If not specified, the current document is assumed.

Output

`True` if the operation was successful otherwise `False`.

ExportMasterFields()

```
Function ExportMasterFields(ByRef pDocTgt As Object, Optional pReset As
Boolean, Optional pDocSrc As Object) As Boolean
```

Exports user fields to some other Writer document.

Input

- `pTargetDoc`: the target Writer document
- `pSourceDoc`: (optional) the source document.
  Defaults to the current document.

Output

`True` if the process went well, otherwise `False`

GetMasterFieldNameOnly()

```
Function GetMasterFieldNameOnly(ByRef pFieldName As String) As String
```

Returns the field name alone, as seen in the UI.

Input

- `pFieldName`: the full field name (incl. the `MFLD_USERSERVICE` part)

Output

The name alone.

GetMasterFieldType()

```
Function GetMasterFieldType(ByRef pFieldName As String) As Long
```

Returns the master field type.

Input

- `pFieldName`: the full field name (incl. the `MFLD_USERSERVICE` part)

Output

The master field type (see the `MFLD_TYPE_XXX` constants)

GetMasterFieldValue()

```
Function GetMasterFieldValue(ByRef pFieldName As String, Optional pDoc
As Object) As Variant
```

Returns the value of a master field.

Input

- `pFieldName`: the user field name,
- `pDoc`: the document in which the master field is searched. If not specified, the current document is assumed.

Output

The value of the field or `Nothing`

`IsMasterFieldUser()`

```
Function IsMasterFieldUser(ByRef pFieldName As String) As Boolean
```

Checks whether a masterfield is a user one or not.

Input

- `pMasterFieldName`: the masterfield name to check

Output

`True` is its name contents a `".User."` part, otherwise `False`

`SetMasterFieldValue()`

```
Sub SetMasterFieldValue(ByRef pFieldName As String, pValue As Variant,
Optional pDoc As Object)
```

Inserts a value into a user field.

Input

- `FieldName`: the user field name to be set
- `pValue`: the value to set to the user field
- `pDoc`: the document in which the master field is searched. If not specified, the current document is assumed.

WriterPrim.Tables

Dependencies: (none)

`GetColumnWidths()`

```
Function GetColumnWidths(ByRef pTableName As String, pRowNum As Long,
Optional pDoc As Object) As Variant
```

Gets a table column widths and returns them into an array.

Input

- `pTableName`: the table name.
- `pRowNum`: the row to browse (0-based) or –1 for a global process.
- `pDoc`: (optional) the document container. If `pDoc` is missing the current document is assumed.

Output

An array of all column widths for a given row, one item per column.
If the row number passed is out of bounds, returns a non-initialised array.

`GetTableActualWidth()`

```
Function GetTableActualWidth(ByRef pTable As Object) As Long
```

Returns the actual width of a table, in 100ths of a millimeter

Input

- `pTable`: a valid table object to measure

Output

The table overall width, in 100ths of a millimeter. If `pTable` is `Null`, returns `0` (zero).
Note: Uses `WriterPrim.Styles.GetStyleAtCursor()`

`GetTableColCountByName()`

```
Function GetTableColCountByName(ByRef pTableName As String, pRowNum As
Long, Optional pDoc As Object) As Long
```

Returns the column count for a given row in a table name.

Input

- `pTableName`: the table name
- `pRowNum`: the row number to count columns
- `pDoc`: (optional) the document container. If `pDoc` is missing the current document is assumed.

Output

The number of columns or `-1` if an error occurred (unknown table or row number out of bounds).

`GetTableRowCountByName()`

```
Function GetTableRowCountByName(ByRef pTableName As String, Optional
pDoc As Object) As Long
```

Returns the row count for a given table name.

Input

- `pTableName`: the table name
- `pDoc`: (optional) the document container. If `pDoc` is missing the current document is assumed.

Output

The number of rows or `-1` if the table name is unknown.

WriterPrim.Autotexts

Dependencies: (none)

Deals with autotexts: retrieving, creating, deleting.

Vocabulary

An Autotext (or glossary entry) is made of three items:

- its shortcut

  This is the string that is entered by the user in the document before hitting F3.

- its name/description/title

  This is the contents of the **Name** textbox within the UI

- its contents

  This is the actual text that is inserted in the document after hitting F3
  The contents may be either raw text or formatted text.

  Note that, because of a long-standing bug with LibreOffice, inserting a raw text adds an (empty) paragraph just after.

Warning
A few of the following functions attempt to create new autotexts or groups. When the target group is the LibO Basis container (global level), some systems might refuse write access under non-administrative accounts, thus cause a runtime error. To deal with this, the error is intercepted by an `On Local Error` statement, hence in this situation the returned object would be Null.
Applies to `AddAutoTextGroup()`, `NewAutoTextGroup()`, `AddAutoText()`, `NewAutoText()`, `UpdateAutoText()`, `UpdateAutoTextTitle()`.

Global Constants

Constant name	Value	Description
Services		
SVC_AUTOCONTAINER	`"com.sun.star.text.AutoTextContainer"`	
SVC_TEXTRANGE	`"com.sun.star.text.TextRange"`	
Errors		
ERR_AUTOTEXT_NONE	`0`	Everything went ok.
ERR_AUTOTEXT_UNKNOWN	`1`	The shortcut wasn't found.
ERR_AUTOTEXT_CANTCREATE	`2`	Autotext entry creation problem.
ERR_AUTOTEXT_CANTDELETE	`3`	Autotext entry deletion problem.
ERR_AUTOTEXT_CANTUPDATE	`4`	Autotext entry update problem.
ERR_AUTOTEXT_EXISTS	`5`	Autotext entry shortcut already exists.
ERR_AUTOTEXT_NAME	`6`	Non valid autotext entry name.
ERR_AUTOTEXT_SHORTCUT	`7`	Non valid autotext entry shortcut.
ERR_AUTOTEXTGROUP_UNKNOWN	`11`	Group not found.
ERR_AUTOTEXTGROUP_CANTCREATE	`12`	Group creation problem.
ERR_AUTOTEXTGROUP_CANTDELETE	`13`	Group deletion problem.

Constant name	Value	Description
`ERR_AUTOTEXTGROUP_EXISTS`	`14`	Group already exists.

Primitives

Autotext Groups

`AddAutoTextGroup()`

```
Function AddAutoTextGroup(ByRef pGroupName As String, Optional ByRef pLocal As Boolean, Optional ByRef pReplace As Boolean) As Long
```

Creates an autotext group and returns the creation status.

Input

- `pGroupName`: the group name to create
- `pLocal`: (optional) if `True`, create it within the local profile tree. `Defaults` to True.
- `pReplace`: (Optional) if `True`, will replace any existing group with the same name. Defaults to `False`.

Output

The creation status. Returns `ERR_AUTOTEXT_NONE` (`0`) if OK, otherwise see `ERR_AUTOTEXT_XXX` constants.

- `ERR_AUTOTEXTGROUP_EXISTS`: the group already exists.
- `ERR_AUTOTEXTGROUP_CANTCREATE`: creation problem. The group wasn't created.

`AutoTextGroupExists()`

```
Function AutoTextGroupExists(ByRef pGroupName As String) As Boolean
```

Returns `True` if a group name exists

Input

- `pGroupName`: the name to check.

Output

`True` if the group named `pGroupName` exists, otherwise `False`.

`AutoTextGroupID()`

```
Function AutoTextGroupID(ByRef pGroupName As String, ByRef pLocal As Boolean) As String
```

Returns the string ID for a group name.
This is used as a group naming convention.

Input

- `pGroupName`: the name for the group.
- `pLocal`: `True` if the group is local to the user, otherwise `False`.

Output

The string ID.

Information

Structure of the string ID: `GroupName*{0|1}`
where `0` = global, `1` = local
Example: if the local group name is `MyGroup`, returns `MyGroup*1`

`AutoTextGroupNameIndex()`

```
Function AutoTextGroupNameIndex(ByRef pGroupName As String) As Long
```

Returns the index of an autotext group within the container from its name.

Input

- `pGroupName`: the name of the group to check.

Output

The index of the autotext group, or `-1` if not found.

`DeleteAutoTextGroupByName()`

```
Function DeleteAutoTextGroupByName(ByRef pGroupName As String) As
Long
```

Deletes an existing autotext group.

Input

- `pGroupName`: the name of the autotext group to be deleted.

Output

A status code. No error is `ERR_AUTOTEXT_NONE` (`0`). See constants `ERR_AUTOTEXT_XXX` above.

- `ERR_AUTOTEXTGROUP_UNKNOWN`: the group is unknown.
- `ERR_AUTOTEXTGROUP_CANTDELETE`: the group couldn't be deleted.

Warning: the group is deleted, whether it is empty or not.

`GetAutoTextGroupByIndex()`

```
Function GetAutoTextGroupByIndex(ByRef pIndex As Long) As Object
```

Retrieves an autotext group object from its index in group.

Input

- `pIndex`: the index of the group to retrieve.

Output

The group object or `Null` if it doesn't exist.

`GetAutoTextGroupByName()`

```
Function GetAutoTextGroupByName(ByRef pGroupName As String) As
Object
```

Retrieves an autotext group object from its name.

Input

- `pGroupName`: the name of the group to retrieve.

Output

The group object or `Null` if it doesn't exist.

`GetAutoTextGroupNames()`

```
Function GetAutoTextGroupNames() As Variant
```

Returns the autotext groups list.
The list is a 1-dimension array of strings.

Output

An array containing the names of all known autotext groups.

`NewAutoTextGroup()`

```
Function NewAutoTextGroup(ByRef pGroupName As String, ByRef pLocal
As Boolean, Optional ByRef pReplace As Boolean) As Object
```

Creates an autotext group and returns the group object.

Input

- `pGroupName`: the autotext group name to create
- `pLocal`: if `True`, create it within the local profile tree
- `pReplace`: (Optional) if True, will replace any existing group with the same name. Defaults to `False`.

Output

The autotext group object created or the existing group object if it was not replaced or `Null` if an error occurred.
Warning: when `pLocal` is set to `False`, the function attempts to create the group within the LibO Basis container on which some systems might refuse write access.

Autotexts

AddAutoText()

```
Function AddAutoText(ByRef pGroupName As String, pTitle As String,
pShortcut As String, pText As Object, Optional pUpdate As Boolean)
As Long
```

Creates a new autotext and returns the creation status.

Input

- `pGroupName`: the owning autotext group name for the new autotext
- `pTitle`: the new autotext title (the full name)
- `pShortcut`: the name for the new autotext (the shortcut)
- `pText`: the autotext actual contents (text cursor)
- `pUpdate`: (Optional) set to `True` will cause the existing autotext with the same name to be updated.

See `UpdateAutoText()` for more details.

Output

The error status. Returns `ERR_AUTOTEXT_NONE` (`0`) if OK, otherwise see `ERR_AUTOTEXT_XXX` constants.

- `ERR_AUTOTEXT_SHORTCUT`: the shortcut is not valid.
- `ERR_AUTOTEXTGROUP_UNKNOWN`: the group is unknown.
- `ERR_AUTOTEXT_EXISTS`: the shortcut already exists in this group.
- `ERR_AUTOTEXT_CANTCREATE`: creation problem. The AutoText wasn't created.

See also: `NewAutoText()`

AddAutoTexts()

```
Function AddAutoTexts(ByRef pGroupName As String, ByRef
pAutoTextArray() As Variant) As Long
```

Adds formatted autotexts to a group.

Input

- `pGroupName`: the target autotext group name
- `pAutoTextArray()`: an array containing the autotexts to be added.
  This is a 2-dimensions string array: dim1 = rows (as many as necessary); dim2 = 2 (3 columns)
  – Col0: the shortcut for the autotext
  – Col1: the name/title of the autotext
  – Col2: the raw text contents (no formatting)

Output

The number of autotext entries added. Returns `-1` if the group name doesn't exist.

AddRawAutoTexts()

```
Function AddRawAutoTexts(ByRef pGroupName As String, ByRef
pAutoTextArray() As Variant) As Long
```

Adds raw (unformatted) autotexts to a group.

Input

- `pGroupName`: the target autotext group name
- `pAutoTextArray()`: an array containing the autotexts to be added.
  This is a 2-dimensions string array: dim1 = rows (as many as necessary); dim2 = 2 (3 columns)
  – Col0: the shortcut for the autotext
  – Col1: the name/title of the autotext
  – Col2: the raw text contents (no formatting)

Output

The number of autotext entries added. Returns `-1` if the group name doesn't exist.

`AutoTextExists()`

```
Function AutoTextExists(ByRef pGroupName As String, pShortcut As
String) As Boolean
```

Returns `True` if an autotext shortcut exists within a given group name.

Input

- `pGroupName`: the name of the owning group.
- `pShortcut`: the autotext entry shortcut to check.

Output

`True` if the shortcut exists within the group named `pGroupName`, otherwise `False`.

`AutoTextShortcutIndex()`

```
Function AutoTextShortcutIndex(ByRef pGroup As Object, pShortcut As
String) As Long
```

Returns the index of an autotext within the group from its name.

Input

- `pGroup`: the group owner object
- `pShortcut`: the name of the shortcut to check.

Output

The index of the autotext, or `-1` if not found.

`CreateAutoTextContainer()`

```
Function CreateAutoTextContainer() As Object
```

Returns an autotext container object.
Input (none)

Output

The container object.

`DeleteAutoTextByShortcut()`

```
Function DeleteAutoTextByShortcut(ByRef pGroupName As String,
pShortcut As String) As Long
```

Deletes an existing autotext from its shortcut, in a given group.

Input

- `pGroupName`: the name of the owning autotext group.
- `pShortcut`: the shortcut for the autotext to be deleted

Output

A status code. No error is `ERR_AUTOTEXT_NONE` (`0`). See constants `ERR_AUTOTEXT_XXX` above.

- `ERR_AUTOTEXT_SHORTCUT`: the shortcut is not valid.
- `ERR_AUTOTEXT_CANTDELETE`: the AutoText couldn't be deleted.
- `ERR_AUTOTEXTGROUP_UNKNOWN`: the group is unknown.
- `ERR_AUTOTEXT_CANTDELETE`: the AutoText couldn't be deleted.

`GetAutoTextByShortcut()`

```
Function GetAutoTextByShortcut(ByRef pGroupName As String, pShortcut
As String) As Object
```

Retrieves an autotext object from its shortcut.

Input

- `pGroupName`: the group owning the autotext.
- `pShortcut`: the shortcut to retrieve.

Output

The autotext object or `Null` if the shortcut wasn't found.

`GetAutoTextShortcuts()`

```
Function GetAutoTextShortcuts(ByRef pGroupName As String) As Variant
```

Returns the autotext shortcut list for a group
The list is a 1-dimension array of strings (vector).

Input

- `pGroupName`: the name of the group to explore

Output

The resulting array of strings with the autotext shortcuts ('element names' in LibO Basic).
The array is `Null` if the group doesn't exist or is empty.

`GetAutoTextTitles()`

```
Function GetAutoTextTitles(ByRef pGroupName As String) As Variant
```

Returns the autotext titles/names list for a group
The list is a 1-dimension array of strings (vector).

Input

- `pGroupName`: the name of the group to explore

Output

The resulting array of strings with the autotext titles.
The array is `Null` if the group doesn't exist or is empty.

`NewAutoText()`

```
Function NewAutoText(ByRef pGroup As Object, pTitle As String,
pShortcut As String, pText As Object, Optional pUpdate As Boolean)
As Object
```

Creates a new autotext and returns the corresponding object.

Input

- `pGroup`: the owning autotext group object for the new autotext.
- `pTitle`: the new autotext title (the full name).
- `pName`: the name for the new autotext (the shortcut).
- `pText`: the autotext actual contents (text cursor).
- `pUpdate`: (Optional) set to `True` will cause the existing autotext with the same name to be updated.

Output

The autotext created or the existing object if it wasn't replaced or `Null` if the creation couldn't be achieved.
See also: `AddAutoText()`

`RenameAutoText()`

```
Function RenameAutoText(ByRef pGroupName As String, ByRef
pOldShortcut As String, ByRef pNewShortcut As String) As Long
```

Renames an entry shortcut. The other items of the entry are left untouched.

Input

- `pGroupName`: the owning autotext group name for the autotext.
- `pOldShortcut`: the current autotext shortcut.
- `pNewShortcut`: the new autotext shortcut.

Output

The error status. Returns `ERR_AUTOTEXT_NONE` (`0`) if OK, otherwise see `ERR_AUTOTEXT_XXX` constants.

- `ERR_AUTOTEXT_SHORTCUT`: a shortcut is not valid.
- `ERR_AUTOTEXT_UNKNOWN`: the shortcut doesn't exist for this group.
- `ERR_AUTOTEXTGROUP_UNKNOWN`: the group is unknown.
- `ERR_AUTOTEXT_EXISTS`: the shortcut already exists in this group.
- `ERR_AUTOTEXT_CANTCREATE`: creation problem. The AutoText wasn't created.

`UpdateAutoText()`

```
Function UpdateAutoText(ByRef pGroupName As String, pTitle As
String, pShortcut As String, pNewText As Object) As Long
```

Updates an existing autotext contents and returns the result.

Input

- `pGroupName`: the owning autotext group name for the autotext
- `pTitle`: the new autotext title (the full name). If not provided (0-length string) then the current value is retained.
- `pShortcut`: the name of the autotext (the shortcut)
- `pNewText`: the autotext actual contents that will replace the existing one (text cursor)

Output

The error status. Returns `ERR_AUTOTEXT_NONE` (`0`) if OK, otherwise see `ERR_AUTOTEXT_XXX` constants.

- `ERR_AUTOTEXT_SHORTCUT`: the shortcut is not valid.
- `ERR_AUTOTEXT_UNKNOWN`: the shortcut doesn't exist for this group.
- `ERR_AUTOTEXTGROUP_UNKNOWN`: the group is unknown.
- `ERR_AUTOTEXT_CANTCREATE`: creation problem. The AutoText wasn't created.

`UpdateAutoTextTitle()`

```
Function UpdateAutoTextTitle(ByRef pGroupName As String, pNewTitle
As String, pShortcut As String) As Long
```

Updates an existing autotext title and returns the result.

Input

- `pGroupName`: the owning autotext group name for the autotext
- `pNewTitle`: the new autotext title (the full name)
- `pShortcut`: the autotext shortcut

Output

The error status. Returns `ERR_AUTOTEXT_NONE` (`0`) if OK, otherwise see `ERR_AUTOTEXT_XXX` constants.

- `ERR_AUTOTEXT_SHORTCUT`: the shortcut is not valid.
- `ERR_AUTOTEXT_TITLE`: the title is not valid.
- `ERR_AUTOTEXT_UNKNOWN`: the shortcut doesn't exist for this group.
- `ERR_AUTOTEXTGROUP_UNKNOWN`: the group is unknown.

Internal

`_CreateHiddenDocument()`

```
Function _CreateHiddenDocument() As Object
```

Creates a hidden writer document in order to get a text cursor.

WriterPrim.Text

Dependencies: (none)

Text-related primitives.

Global Constants

Constant name	Value	Description

Primitives

`GetSelection()`

```
Function GetSelection(Optional pNearestWord As Boolean, Optional ByRef
pDoc As Object) As Object
```

Returns the current selection.

Input

- `pNearestWord`: (optional) if there's no current selection, causes the function to return the caret nearest word.
  Defaults to `False`.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The current selection or the nearest word or `Null` if none or if no document found.

`HasSelection()`

```
Function HasSelection(Optional ByRef pDoc As Object) As Boolean
```

Returns `True` if some text is selected.

Input

- `pDoc`: (optional) the document to check.
  Defaults to the current document.

Output

`True` if there's at least one selection, otherwise `False`.

WriterPrim.Bookmarks

Dependencies: (none)

Bookmark-related primitives.

Note: a bookmark can be a single point or encompass a number of characters.

Global Constants

Constant name	Value	Description

Primitives

`CreateBookmark()`

```
Function CreateBookmark(ByRef pBookmarkName As String, ByRef pCursor As Object, Optional ByRef pDoc As Object) As Object
```

Creates a bookmark in a document at a cursor place.

Input

- `pBookmarkName`: the name of the bookmark to be created.
- `pCursor`: the cursor where the bookmark should be created.
- `pDoc`: (optional) the document that will get the new bookmark.
  Defaults to the current document.

Output

The created bookmark object or `Null` if the process failed.
Possible process failures:

- the cursor is not set or not compatible.
- no bookmark name was provided.
- the bookmark name already exists.

`GotoBookmark()`

```
Function GotoBookmark(ByRef pBookmarkName As String, Optional pSelect As Boolean, Optional ByRef pVisible As Boolean, Optional ByRef pDoc As Object) As Object
```

Sets a cursor to a bookmark and returns the bookmark object.

Input

- `pBookmarkName`: the name of the bookmark to be reached.
- `pSelect`: (optional) select the text between the current cursor position and the bookmark.
  Defaults to `False`.
- `pVisible`: (optional) `True` to use the visible cursor otherwise use a (invisible) text cursor.
  Defaults to `True`.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The bookmark object or `Null` in case of failure.
The possible process failures (result `Null`) are:

- no bookmark name was provided
- the bookmark name wasn't found.

GotoBookmarkFromCursor()

```
Function GotoBookmarkFromCursor(ByRef pBookmarkName As String, ByRef
pCursor As Object, Optional pSelect As Boolean, Optional ByRef pDoc As
Object) As Object
```

Sets a given cursor to a bookmark and returns the bookmark object.

Input

- `pBookmarkName`: the name of the bookmark to be reached.
- `pCursor`: the cursor to use to go to the bookmark.
- `pSelect`: (optional) select the text between the current cursor position and the bookmark.
  Defaults to `False`.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

The bookmark object or `Null` in case of failure.
The possible process failures (result `Null`) are:

- no bookmark name was provided
- the bookmark name wasn't found.

RemoveBookmark()

```
Function RemoveBookmark(ByRef pBookmarkName As String, Optional ByRef
pDoc As Object) As Boolean
```

Deletes a bookmark and returns the result.

Input

- `pBookmarkName`: the name of the bookmark to be removed.
- `pDoc`: (optional) the document to process.
  Defaults to the current document.

Output

`True` if the bookmark was removed otherwise `False`.
The possible process failures (result `False`) are:

- no bookmark name was provided.
- the bookmarkname wasn't found.

DataStructPrim

DataStructPrim.Collections

(to be finished)

Global Constants

Constant name	Value	Description
Errors		
ERR_COLL_OK	0	
ERR_COLL_NOTSET	1	
ERR_COLL_KEYEXISTS	2	
ERR_COLL_OBJECT	3	

Primitives

AddCollectionItem()

```
Function AddCollectionItem(ByRef pColl As Object, ByRef pItem As
Variant, pKey As String, Optional ByRef pForce As Boolean) As Long
```

Adds an item to a collection.
The action doesn't end with an internal error when the key already exists.
If the key is already present, the existing item may be replaced with the new one.

Input

- `pColl`: the collection for storing the item.
- `pItem`: the item to add.
- `pKey`: the key for the item.
- `pForce`: (optional) the existing item is replaced if the key already exists. Defaults to `False`.

Output

The process result.
Possible values (to be replaced with constants):

- `ERR_COLL_OK`: processed OK.
- `ERR_COLL_NOTSET`: the collection object is not initialized.
- `ERR_COLL_KEYEXISTS`: the key already exists and the `pForce` option is `False`.
- `ERR_COLL_OBJECT`: the specified collection object is not a collection. (TBD)

DialogPrim

DialogPrim.Dialogs

Dependencies: (none)

Primitives

`BrowseForDir()`

```
Function BrowseForDir(ByRef pDefDir As String, pTitle As String,
pDescription As String) As String
```

Browse for a directory and returns the user's choice.
This is a simple `FolderPicker` encapsulation.

Input

- `pDefDir`: the default (starting) directory (in URL or OS form).
- `pTitle`: the dialog title.
- `pDescription`: the description string for display in the dialog.
  Note that the description might not be displayed under all OSes.

Output

The selected folder name or a zero-length string if the user canceled the operation.
The output folder name is in URL form with a trailing "\".

`CreateDialog()`

```
Function CreateDialog(ByRef pLibName As String, pModuleName As String,
Optional ByRef pLibCtnr As Object) As Object
```

Creates a Uno dialog.

Input

- `pLibName`: the dialog library name.
- `pModuleName`: the dialog module name.
- `pLibCtnr`: (optional) the library container.
  If not specified, looks into `DialogLibraries`.

Output

The dialog object created or `Null` if the dialog couldn't be created.

Note: this is an enhanced version of the LibreOffice `Tools.ModuleControls.LoadDialog()` function, adding error checking.

`YesNoDialog()`

```
Function YesNoDialog(ByRef pTitle As String, pMsg As String) As Boolean
```

Displays a "Yes/No" dialog and returns the user's choice.

Input

- `pTitle`: the dialog box title.
- `pMsg`: the message to display in the dialog box.

Output

`True` if the user selected Yes, `False` otherwise.

FormPrim

LibreOffice forms primitives.

FormPrim.Widgets

Dependencies: (none)

Global Constants

Constant name	Value	Description
Errors		
ERR_FORMPRIM_NODOC	-1	The specified document doesn't exist.
ERR_FORMPRIM_NOFORM	-2	The specified form doesn't exist.
ERR_FORMPRIM_NOCONTROL	-3	No controls on the form.
ERR_FORMPRIM_NOWIDGET	-4	No such widget on the form.
Misc.		
WIDG_ID_CBX	5	Checkbox identifier

Primitives

CheckBoxCount()

```
Function CheckBoxCount(ByRef pFormName As String, Optional ByRef pDoc
As Object, Optional pChecked As Boolean, Optional pNonPrintable As
Boolean) As Long
```

Counts the (un)checked checkboxes.

Input

- `pFromName`: the form where checkboxes are to be checked
- `pDoc`: (optional) The document where checkboxes are to be counted. Defaults to the current document.
- `pChecked`: a flag to detect (un)checked chekboxes. Defaults to `True` (counts checked boxes).
- `pNonPrintable`: a flag for non-printable widgets count. If set to `False` (default), non-printable widgets are not counted.

Output

The count of checked checkboxes or `-1` if an error was encountered.
Possible errors:

- no form on the document
- no controls on the form
- no checkbox control

GetFormControl()

```
Function GetFormControl(ByRef pFormName As String, ByRef pCtrlName As
String, Optional ByRef pDoc As Object) As Object
```

Returns a control object on a form from its name.

Input

- `pFormName`: the form name on which the control is placed.
- `pCtrlName`: the control name.
- `pDoc`: (optional) the document owning the form. Defaults to the current document.

Output

The control object of `Null` if not found or the form is not found.

FormPrim.ProgressWidgetClass

Dependencies: (none)

A progress bar management class for user feedback during lengthy processes in forms.

This class is very similar to the LibOPrim.ProgressClass(see page 98). While that one is an all purpose progress bar class, the ProgressWidgetClass is dedicated to displaying and managing progress actions within forms.

This class encapsulates the LibreOffice form progressbar widget functionalities. It aims at providing an entry point where to manage progress bars steps, messages and display delays.

This class defines the properties of a progress bar:

- a progress bar object.
- messages and delays for the different process steps.
- messages and delays for startup and termination steps.

TProgressStepInformation

This structure stores a message and a display delay for a given progress step.

Member name	Type	Description
rMessage	`String`	The message to display for a given step.
rDelay	`Long`	The message display delay, in ms..

Primitives

This module offers a factory function that can create an object of `TProgressStepInformation`. Call it from your code to initialize such objects.

`CreateProgressStepInformation()`

```
Function CreateProgressStepInformation(Optional ByRef pMsg As
String, Optional ByRef pDelay As Long) As
TProgressStepInformation
```

A factory function for the `TProgressStepInformation` type.

Input

- `pMsg`: (optional) the message for a step.
  Defaults to a zero-length string.
- `pDelay`: (optional) the delay during which the message is displayed, in ms.
  Defaults to `0`.

Output

A `TProgressStepInformation` object.

Properties

Property	Type	Description
`CurrentStep`	Object [RO]	The progress current step information. Returns an object of `TProgressStepInformation` type.
`CurrentStepNum`	Long [RO]	The progress current step number. The steps are numbered from `0` (first), then `1` (first added step) to `N` (n^{th} added step) then `-1` (last).
`FirstStep`	Object [RO]	The progress first step information Returns an object of `TProgressStepInformation` type. Use `SetFirstStep()` method to set this object.
`LastStep`	Object [RO]	The progress last step information Returns an object of `TProgressStepInformation` type. Use `SetLastStep()` method to set this object.
`Mute`	Boolean [R/W]	(un)sets the display of messages. Defaults to `False`.
`ProgressBar`	Object [R/W]	The progress bar to update. Set it from a form ProgressBar widget.
`ProgressLabel`	Object [R/W]	A label object for user feedback. If not set, messages are not displayed.

Methods

`Public`

`AddStep()`

```
Public Sub AddStep(ByRef pMsg As String, Optional pDelay As Long)
```

Adds a step to the progress.

Input

- `pMsg`: the message for the step
- `pDelay`: (optional) then message delay for the display.
  Defaults to `0`.

`AddSteps()`

```
Public Sub AddSteps(ByRef pArrSteps As Variant)
```

Adds a set of steps to the steps collection.
The steps are added at the end of the steps collection.
The first and last steps are set using the dedicated subs `SetFirstStep()` and `SetLastStep()`.

Input

- `pArrSteps`: a nested array of steps information.
  Each item is an array: `("Message", DelayMS)`
  At runtime, the items are displayed in the insertion order.

Usage

```
AddSteps(Array(Array("Step1", 0), Array("Step2", 1000)))
```

Adds two steps: the first one displays "Step1" with no delay, the second one displays "Step2" during 1,000 ms.

NextStep()

```
Public Sub NextStep(Optional ByRef pInc As Long, Optional ByRef pMsg As
String)
```

Displays the next step information.

Input

- `pInc`: (optional) the increment value.
  Defaults to `1`.
- `pMsg`: (optional) a message to display.
  If used, this message overrides the message recorded in the steps collection.
  If the message is a zero-length string, the displayed message is not updated.

ProgressEnd()

```
Public Sub ProgressEnd(Optional ByRef pMsg As String)
```

Ends the progress bar display.

Input

- `pMsg`: (optional) a message to display.
  Defaults to the message recorded in the `LastStep`.
  If the message is a zero-length string, the displayed message is not updated.

ProgressStart()

```
Public Sub ProgressStart(Optional ByRef pMsg As String)
```

Starts the progress bar display.

Input

- `pMsg`: (optional) a message to display.
  Defaults to the message recorded in the `FirstStep`.
  If the message is a zero-length string, the displayed message is not updated.

SetFirstStep()

```
Public Sub SetFirstStep(ByRef pMsg As String, Optional pDelay As Long)
```

Sets the first step information.

Input

- `pMsg`: the message to display.
- `pDelay`: (optional) the display delay in ms.
  Defaults to `0`.

SetLastStep()

```
Public Sub SetLastStep(ByRef pMsg As String, Optional pDelay As Long)
```

Sets the last step information.

Input

- `pMsg`: the message to display.
- `pDelay`: (optional) the display delay in ms.
  Defaults to `0`.

Private (private methods are prefixed with underscores "_")
They are not intended for use out of the class.

`_AddCollectionItem()`

```
Private Sub _AddCollectionItem(ByRef pColl As Object, ByRef pItem As
Variant, pKey As String)
```

Adds an item to a collection.
If the key is already present, the existing item is replaced with the new one.

Input

- `pColl`: the collection for storing the item.
- `pItem`: the item to add.
- `pKey`: the key for the item.

`_ArrayExists()`	`Private Function _ArrayExists(ByRef pArray As Variant) As Boolean` Checks whether an array exists (is dimensioned) and can be manipulated. **Input** • `pArray`: the array to check. **Output** `True` if `pArray` is a valid variable otherwise `False`.

EmailPrim

EmailPrim.SendMailClass

Dependencies: (none)

This module defines a mail sending class directly from LibreOffice.

Warning: only tested using the `SMTP_SERVINSECURE` mode.

Constants

All constants have a `EMAILCLASS_` prefix. Moreover, error constants are `ERR_` prefixed.

Constant name	Value	Description
Error Constants		
ERR_EMAILCLASS_NORUN	-1	
ERR_EMAILCLASS_NONE	0	
ERR_EMAILCLASS_NORECIPIENT	1	
ERR_EMAILCLASS_AUTHENTICATION	2	
ERR_EMAILCLASS_RECIPIENTEXISTS	3	
ERR_EMAILCLASS_NOSUCHRECIPIENT	4	
ERR_EMAILCLASS_NOSUCHCC	5	
ERR_EMAILCLASS_NOSUCHBCC	6	
ERR_EMAILCLASS_ATTACHMENTEXISTS	7	
ERR_EMAILCLASS_CANTATTACH	8	
Server Context		
SMTP_SERVINSECURE	"Insecure"	Insecure server flag.
SMTP_SERVSSL	"SSL"	SSL security server flag.
SMTP_SERVTLS	"STARTTLS"	STARTTLS security server flag.

Properties

EmailBodyText	String [R/W]	The email body. The body is text-only (no formatting).
EmailReplyTo	String [R/W]	The reply-to address. Mail addresses are not checked before sending the messages.
EmailSubject	String [R/W]	The email subject. A check is made before sending the message: the subject must be present.
ErrorCode	Long [RO]	The error code. See the `ERR_EMAILCLASS_Xxxx` constants above.
ErrorMessage	String [RO]	The error message.
Log	Object [R/W]	(optional) A log object for logging purposes. If this object is set, the message sending process is logged accordingly. The log object should be of type `LogClass` (see that class, at page 58).

ServerName	`String [R/W]`	The server name.
ServerPort	`Long [R/W]`	Server port number. No default value is assumed, hence this property must be set.
ServerSecurity	`String [R/W]`	Server security string. See the `SMTP_SERVxxx` constants.
SMTPPassword	`String [R/W]`	The user password for sending messages. If the security type is `Insecure`, then the password **must** be unset.
UserName	`String [R/W]`	The user name for sending messages. If the security type is `Insecure`, then the user name **must** be unset.
UserAddress	`String [R/W]`	The sender mail address. Mail addresses are not checked before sending the messages.

Methods

AddAttachement()

```
Function AddAttachment(ByRef pAttachmentURL As String) As Long
```

Adds an attachment to the message.

AddBCC()

```
Sub AddBCC(ByRef pBCC As String)
```

Adds a recipient to the 'BCC' recipients list.

Input

- `pCC`: the recipient email address.
  No check is made against the address compliance with RFC 822.

Output

An execution result code, in:

- `ERR_SENDMAILCLASS_NONE`: the 'BCC' recipient was added.

`ERR_SENDMAILCLASS_RECIPIENTEXISTS`: the 'BCC' recipient already exists.

AddBCCbyCSV()

```
Sub AddBCCbyCSV(ByRef pBCCcsv As String)
```

Adds multiple recipients to the 'BCC' recipients list at once.

Input

- `pBCCCsv`: the recipient email addresses, in a comma separated string.
  Ex: `"me@isp.com,her@otherisp.net,someone.else@someisp.com"`
  No check is made against the address compliance with RFC 822.

Output

A execution result code, in:

- `ERR_SENDMAILCLASS_NONE`: the 'BCC' recipient was added.
- `ERR_SENDMAILCLASS_RECIPIENTEXISTS`: the 'BCC' recipient already exists.

AddCC()

```
Sub AddCC(ByRef pCC As String)
```

Adds a recipient to the 'CC' recipients list.

Input

- `pCC`: the recipient email address.
  No check is made against the address compliance with RFC 822.

Output

An execution result code, in:

- `ERR_SENDMAILCLASS_NONE`: the 'CC' recipient was added.
- `ERR_SENDMAILCLASS_RECIPIENTEXISTS`: the 'CC' recipient already exists.

AddCCbyCSV()

```
Sub AddCCbyCSV(ByRef pCCcsv As String)
```

Adds multiple recipients to the 'CC' recipients list at once.

Input

- pCCCsv: the recipient email addresses, in a comma separated string.
  Ex: "me@isp.com,her@otherisp.net,someone.else@someisp.com"
  No check is made against the address compliance with RFC 822.

Output

A execution result code, among:

- ERR_SENDMAILCLASS_NONE: the 'CC' recipient was added.
- ERR_SENDMAILCLASS_RECIPIENTEXISTS: the 'CC' recipient already exists.

AddTo()

```
Sub AddTo(ByRef pTo As String)
```

Adds a recipient to the 'To' recipients list.

Input

- pRecipient: the recipient email address.
  No check is made against the address compliance with RFC 822.

Output

An execution result code, in:

- ERR_SENDMAILCLASS_NONE: the 'To' recipient was added.
- ERR_SENDMAILCLASS_RECIPIENTEXISTS: the 'To' recipient already exists.

Note: At least one primary 'To' recipient must be provided as the SendMail() method won't transmit any message that has none.

AddToByCSV()

```
Sub AddToByCSV(ByRef pToCsv As String)
```

Adds multiple recipients to the 'To' recipients list at once.

Input

- pToCsv: the recipient email addresses, in a comma separated string.
  Ex: "me@isp.com,her@otherisp.net,someone.else@someisp.com"
  No check is made against the address compliance with RFC 822.

Output

A execution result code, among:

- ERR_SENDMAILCLASS_NONE: the 'To' recipient was added.
- ERR_SENDMAILCLASS_RECIPIENTEXISTS: the 'To' recipient already exists.

ClearAttachments()

```
Sub ClearAttachments()
```

Deletes all attachments from the message.

ClearBCCs()

```
Sub ClearBCCs()
```

Clears the BCCs list

ClearCCs()

```
Sub ClearCCs()
```

Clears the CCs list.

ClearError()

```
Sub ClearError()
```

clears any error code and message.

ClearTos()

```
Sub ClearTos()
```

Clears the recipients list.

`GetAttachment()`

```
Function GetAttachment(ByRef pNum As Long) As String
```

Returns an attachment

Input

- `pNum`: the wanted attachment number (1-based)

Output

The attachment name or an empty string if not found.

`RemoveBCC()`

```
Function RemoveBCC(ByRef pBCC As String) As Long
```

Removes a BCC from the BCCs list.

Output

An error code (0 if on error occurred).
The actual error information can be retrieved using the `ErrorCode` and `ErrorMessage` properties.

`RemoveCC()`

```
Function RemoveCC(ByRef pCC As String) As Long
```

Removes a CC from the CCs list.

Output

An error code (0 if on error occurred).
The actual error information can be retrieved using the `ErrorCode` and `ErrorMessage` properties.

`RemoveTo`

```
Function RemoveTo(ByRef pRecipient As String) As Long
```

Removes a recipient from the recipients list.

Output

An error code (0 if on error occurred).
The actual error information can be retrieved using the `ErrorCode` and `ErrorMessage` properties.

`SendMail()`

```
Function SendMail() As Long
```

Actually sends the email to the recipients.

Output

An error code (0 if on error occurred).
The actual error information can be retrieved using the `ErrorCode` and `ErrorMessage` properties.

`TestConnection()`

```
Function TestConnection() As Long
```

A connection test tool.

Output

An error code (0 if on error occurred).
The actual error information can be retrieved using the `ErrorCode` and `ErrorMessage` properties.

Usage

TBD

EmailPrim.SendMailClassListeners

Dependencies: (none)

This is a complementary module to the `SendMailClass` one. Both go together and must not (cannot) be used separately. This module implements the listeners the `SendMailClass` uses and needs.

Primitives

`CreateBodyListener()`	`Function CreateBodyListener(ByRef pBodyText As String) As Object`
`ResetAttachmentCounters()`	`Sub ResetAttachmentCounters()`
`CreateAttachmentListener(`	`Function CreateAttachmentListener(ByRef pAttachments As Object) As Object`
`GetMimeTypeFromUrl()`	`Function GetMimeTypeFromUrl(ByRef pFileName As String) As Variant`
`GetFileContents()`	`Function GetFileContents(ByRef pFileName As String) As Variant`
`CreateServerContextListener()`	`Function CreateServerContextListener(ByRef pServerName As String, pPort As Long, pIsSecure As Boolean) As Object`
`CreateAuthenticationListener()`	`Function CreateAuthenticationListener(ByRef pUserName As String, pPwd As String) As Object`

Appendices

LibreOffice Runtime Error Codes

Obtained from LibreOffice v.5.1.1.

Code	Description
1	An exception occurred.
2	Syntax error.
3	Return without Gosub.
4	Incorrect entry; please retry.
5	Invalid procedure call.
6	Overflow.
7	Not enough memory.
8	Array already dimensioned.
9	Index out of defined range.
10	Duplicate definition.
11	Division by zero.
12	Variable not defined.
13	Data type mismatch.
14	Invalid parameter.
18	Process interrupted by user.
20	Resume without error.
28	Not enough stack memory.
35	Sub-procedure or function procedure not defined.
48	Error loading DLL file.
49	Wrong DLL call convention.
51	Internal error .
52	Invalid file name or file number.
53	File not found.
54	Incorrect file mode.
55	File already open.
57	Device I/O error.
58	File already exists.
59	Incorrect record length.
61	Disk or hard drive full.
62	Reading exceeds EOF.
63	Incorrect record number.
67	Too many files.
68	Device not available.
70	Access denied.

Code	Description
71	Disk not ready.
73	Not implemented.
74	Renaming on different drives impossible.
75	Path/File access error.
76	Path not found.
91	Object variable not set.
93	Invalid string pattern.
94	Use of zero not permitted.
250	DDE Error.
280	Awaiting response to DDE connection.
281	No DDE channels available.
282	No application responded to DDE connect initiation.
283	Too many applications responded to DDE connect initiation.
284	DDE channel locked.
285	External application cannot execute DDE operation.
286	Timeout while waiting for DDE response.
287	User pressed ESCAPE during DDE operation.
288	External application busy.
289	DDE operation without data.
290	Data are in wrong format.
291	External application has been terminated.
292	DDE connection interrupted or modified.
293	DDE method invoked with no channel open.
294	Invalid DDE link format.
295	DDE message has been lost.
296	Paste link already performed.
297	Link mode cannot be set due to invalid link topic.
298	DDE requires the DDEML.DLL file.
323	Module cannot be loaded; invalid format.
341	Invalid object index.
366	Object is not available.
380	Incorrect property value.

Code	Description
382	This property is read-only.
394	This property is write only.
420	Invalid object reference.
423	Property or method not found: .
424	Object required.
425	Invalid use of an object.
430	OLE Automation is not supported by this object.
438	This property or method is not supported by the object.
440	OLE Automation Error.
445	This action is not supported by given object.
446	Named arguments are not supported by given object.
447	The current locale setting is not supported by the given object.
448	Named argument not found.
449	Argument is not optional.
450	Invalid number of arguments.
451	Object is not a list.
452	Invalid ordinal number.
453	Specified DLL function not found.
460	Invalid clipboard format.
951	Unexpected symbol: .
952	Expected: .
953	Symbol expected.
954	Variable expected.
955	Label expected.
956	Value cannot be applied.
957	Variable already defined.
958	Sub procedure or function procedure already defined.

Code	Description
959	Label already defined.
960	Variable not found.
961	Array or procedure not found.
962	Procedure not found.
963	Label undefined.
964	Unknown data type .
965	Exit expected.
966	Statement block still open: missing.
967	Parentheses do not match.
968	Symbol already defined differently.
969	Parameters do not correspond to procedure.
970	Invalid character in number.
971	Array must be dimensioned.
972	Else/Endif without If.
973	not allowed within a procedure.
974	not allowed outside a procedure.
975	Dimension specifications do not match.
976	Unknown option: .
977	Constant redefined.
978	Program too large.
979	Strings or arrays not permitted.
1000	Object does not have this property.
1001	Object does not have this method.
1002	Required argument lacking.
1003	Invalid number of arguments.
1004	Error executing a method.
1005	Unable to set property.
1006	Unable to determine property.
65535	General error.

Alphabetical Index

Global Constants

Primitives