

KitchenDraw SDK

The KitchenDraw SDK (Software Development Kit) is a set of computer files (functions library, sample programmes) as well as the associated documentation. It allows to extend and customize KitchenDraw and to interface it with third party management softwares, ERP and production softwares.

The KitchenDraw SDK can be used to develop the following pieces of software :

- Programmes to generate or update all or part of KitchenDraw catalogues
- Programmes to create paper catalogues or on-line catalogues from KitchenDraw catalogues
- Order files generation modules (order files to be sent to suppliers or manufacturers)
- Highly parametrical objects configurators (wizards for special worktops, stairs, conservatories, etc.)
- Dynamic interface of KitchenDraw with third party management software
- « plug-in » functions that are launched automatically when certain events occur (opening a scene, closing KitchenDraw, ...)
- KitchenDraw extensions (specific commands added to the KitchenDraw menus)
- Modification or replacement of KitchenDraw standard dialog boxes

This document is going to describe the various aspects of the KitchenDraw SDK.

« ASPInSitu » ActiveX Component

The « ASPInSitu » ActiveX component allows developing programmes that can read and write information from and into KitchenDraw catalogues and scenes.

It includes 4 objects (classes) :

- Appli
- Catalog
- Scene
- Dico

The « Appli » class

The two basic functions of the **Appli** class allow to allocate and to free a workspace (session) in order to attach a specific context to each user. This is particularly interesting for multi users applications like Web applications.

The session will stock among other information, the current language, the current catalogue and the current scene that can be accessed by the programmers through the **Catalog** and **Scene** classes.

The principle is to call first the **StartSession** function. the *SessionId* value which is return by this function will be passed systematically as a parameter to the functions of **Catalog**, **Scene** ou **Dico** classes that will be called later on. At the end of the process, the **EndSession** function must be called to free the memory resource occupied by the workspace (session).

Among the functions of the **Appli** class, we can find also miscellaneous functions that can't be affected to the Catalog or Scene classes like the function that defines the current language or the function that returns the translation of a characters string into the current language (provided that this characters string is present in the current catalog).

At last we can find functions to access data into catalogues that are not the current catalogue (catalogues that are not loaded in memory). These functions require that the catalogue file name be passed as a parameter.

The list of the **Appli** class functions follows:

StartSession(*IniFileNameWithoutpath* As String) As Long

This function allocates a memory workspace (session) that will be used to store the current catalogue, the current scene, the current language among other information.

The *IniFileNameWithoutpath* parameter specifies the application configuration file name (with the .INI extension but without any path because the file must be located in the same directory than the ASPINSITU.DLL file). The configuration file defines the application working directories (catalogues, textures, ...), the name of the dictionary file as well as other parameters like the rules used to place the linear articles automatically.

Return value: the function returns a numerical value which identifies the workspace (session) attached to each user. This value will have to be passed as a parameter to most of the functions called subsequently.

StartSessionFromCallParams(*CallParamsBlock* As Long) As Long

This function allocates a memory workspace (session) like the **StartSession** function does but with a significant difference; it doesn't create a new scene but it uses the scene passed in the *CallParamsBlock* parameter as the current scene.

Apart from the current scene, the *CallParamsBlock* parameter contains information about a possible supplier order like for example the supplier identifier.

When the *CallParamsBlock* parameter is passed to a MobiScript extension function (a function that has been added to a MobiScript menu), it refers to the catalogue that is currently loaded in MobiScript. So, if no catalogue has been previously created or loaded by programme, the functions of the Catalog class apply to the catalogue that is currently loaded in MobiScript. However, before calling any function of the Catalog class, it is highly recommended to call the **IsLoaded** function of the **Catalog** class to be sure a catalogue is currently loaded in MobiScript.

The **StartSessionFromCallParams** function must be called when implementing standard functions that are called directly by KitchenDraw: supplier order generation function (GenerateOrder, ProcessOrder), supplier order validity checking function (TestOrder), or when developing a « plug-in ».

Return value: the function returns a numerical value which identifies the workspace (session) attached to each user. This value will have to be passed as a parameter to most of the functions called subsequently.

ATTENTION : the function returns 0 if there are no more hours of use AND if there is no dongle plugged. Therefore, it's necessary to check the returned value before passing it as a parameter.

EndSession(*SessionId* As Long) As Boolean

This function frees (de allocates) the memory space allocated by the **StartSession** or **StartSessionFromCallParams** function at the beginning of the process.

The *SessionId* parameter represents the value returned by the **StartSession** or **StartSessionFromCallParams** function.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SetLanguage(*SessionId* As Long, *LanguageCode* As String) As Boolean

This function sets up the current language among the available languages in KitchenDraw. The *LanguageCode* parameter represents the language code as it's displayed in the « Language » combo box of the « Setup | System » KitchenDraw dialog box. For example, « FRA » corresponds to French language, « ENG » to English language, « DEU » to German language, etc. The default current language is the English language.

Return value: the function returns 1 if no problem occurred; else it returns 0.

GetLanguage(*SessionId* As Long) As String

Return value: the function returns the code of the current language as it's displayed in the « Language » combo box of the « Setup | System » KitchenDraw dialog box.

GetTranslatedText (*SessionId* As Long, *Text* As String) As String

This function looks up the translation into the current language of the characters string passed in the *Text* parameter. The characters string to be translated must be written in the working language of the current catalogue and its translation is looked up into the dictionary table of the current catalogue.

Return value: the function returns the translated characters string or an empty characters string if the characters string to be translated has not been found.

GetCallParamsWindow (*SessionId* As Long) As Long

This function must be called after the **StartSessionFromCallParams** function.

Return value: the function returns the window « handle » of the KitchenDraw application having provided the *CallParamsBlock* parameter.

SetCallParamsInfo (*SessionId* As Long, *Value* As String, *InfoType* As Long) As String

This function writes information into the *CallParamsBlock* parameters block passed by the KitchenDraw application while calling standard functions or “plug-in” functions. Only the *InfoType* having a value of 7 (APPCALLPARAM_BUFFER) can be written (the other *InfoType* are read only). This function is useful while developing a “plug-in” function that must return a value or modify a value passed by KitchenDraw as a parameter through the APPCALLPARAM_BUFFER field (for example, a “plug-in” function that would modify the “object” of the email message related to an order to a supplier).

Return value: the function returns 1 if no problem occurred; else it returns 0.

GetCallParamsInfo (*SessionId* As Long, *InfoType* As Long) As String

This function reads the information contained into the *CallParamsBlock* parameter block passed by the KitchenDraw application while calling standard functions or “plug-in” functions.

The required information is specified in the *InfoType* parameter. Here is the list of the possible *InfoType* values:

<i>InfoType</i> value	Description
APPCALLPARAM_ORDERDIRECTORY	Directory where the supplier order file should be generated.
APPCALLPARAM_ORDERFILENAME	Supplier order file name (including the access path to the directory)
APPCALLPARAM_ORDERWEEK	Delivery week number of the supplier order
APPCALLPARAM_ORDERYEAR	Delivery year of the supplier order
APPCALLPARAM_ACTIVEOBJECT	Active object rank in the scene

APPCALLPARAM_SUPPLIERID	Supplier identifier of the supplier order
APPCALLPARAM_BUFFER = 7	Buffer allowing a bidirectional exchange of a characters string between KitchenDraw and a “plug-in” function
APPCALLPARAM_BUFFERSIZE = 8	Size of the buffer (in bytes) allocated by KitchenDraw

Return value: the function returns the required information as a characters string.

InsertMenuItem(*SessionId* As Long, *MenuItemText* As String, *AccelControlKey* As Long, *AccelKeyCode* As Long, *IconFileName* As String, *MenuRank* As Long, *ItemRank* As Long, *DLLFileName* As String, *ClassName* As String, *FunctionName* As String) As Boolean

This function inserts a menu item (a command) into the KitchenDraw application.

Normally this function is called (once or several times) in a *OnAppStartAfter* “plug-in” function to add custom commands. Please see the “sdk_plugin” VB6 sample application for more information.

The *MenuItemText* parameter represents the command name, *AccelControlKey* and *AccelKeyCode* the key combinaison to be pressed to run the command (see values table bellow), *IconFileName* the image file name representing the icon (not managed in this version), *MenuRank* the rank of the menu where the command is inserted (starting from 1) and *ItemRank* the rank of the inserted command in the menu (starting from 1).

The following parameters specify the function that will be called when the command will be chosen by the user.

The *DLLFileName* parameter represents the DLL file name (with the.DLL extension) of the DLL which contains the function to be called. If the DLL is not located in the KitchenDraw directory (which is normally the case) it's possible to prefix this file name with the access path in order to indicate the directory where the DLL is located.

If the function belongs to an ordinary DLL, the *ClassName* parameter must be left empty.

If the function belongs to an ActiveX DLL, the *ClassName* parameter must be indicated.

The *FunctionName* parameter represents the name of the function to be called.

This function must handle a unique parameter of type Long (*CallParamsBlock*) that contains the address of a CALLPARAMSBLOCK data structure that will be passed by KitchenDraw when calling the function.

This function must return a value of type Boolean (True if the scene has been modified by the function and False if it hasn't).

This function must not be called before the **StartSessionFromCallParams** function.

Here is the list of the control keys:

Code value	Control key description
CK_SHIFT = 1	Shift key
CK_CONTROL = 2	Control key
CK_MENU = 4	Alt key

It's possible to combine control keys with OR operators to indicate that several key are pressed simultaneously. For example, in VB, you could write « CK_SHIFT Or CK_CONTROL ».

Here is the list of the keys:

Key code value	Key description
VK_0 = 0x30 (&H30)	Key « 0 »
VK_1 = 0x31 (&H31)	Key « 1 »
VK_2 = 0x32 (&H32)	Key « 2 »
VK_3 = 0x33 (&H33)	Key « 3 »
VK_4 = 0x34 (&H34)	Key « 4 »
VK_5 = 0x35 (&H35)	Key « 5 »
VK_6 = 0x36 (&H36)	Key « 6 »
VK_7 = 0x37 (&H37)	Key « 7 »
VK_8 = 0x38 (&H38)	Key « 8 »
VK_9 = 0x39 (&H39)	Key « 9 »
VK_A = 0x41 (&H41)	Key « A »
VK_B = 0x42 (&H42)	Key « B »
VK_C = 0x43 (&H43)	Key « C »

VK_D = 0x44 (&H44)	Key « D »
VK_E = 0x45 (&H45)	Key « E »
VK_F = 0x46 (&H46)	Key « F »
VK_G = 0x47 (&H47)	Key « G »
VK_H = 0x48 (&H48)	Key « H »
VK_I = 0x49 (&H49)	Key « I »
VK_J = 0x4A (&H4A)	Key « J »
VK_K = 0x4B (&H4B)	Key « K »
VK_L = 0x4C (&H4C)	Key « L »
VK_M = 0x4D (&H4D)	Key « M »
VK_N = 0x4E (&H4E)	Key « N »
VK_O = 0x4F (&H4F)	Key « O »
VK_P = 0x50 (&H50)	Key « P »
VK_Q = 0x51 (&H51)	Key « Q »
VK_R = 0x52 (&H52)	Key « R »
VK_S = 0x53 (&H53)	Key « S »
VK_T = 0x54 (&H54)	Key « T »
VK_U = 0x55 (&H55)	Key « U »
VK_V = 0x56 (&H56)	Key « V »
VK_W = 0x57 (&H57)	Key « W »
VK_X = 0x58 (&H58)	Key « X »
VK_Y = 0x59 (&H59)	Key « Y »
VK_Z = 0x5A (&H5A)	Key « Z »
VK_NUMPAD0 = 0x60 (&H60)	Key « 0 » on the num pad
VK_NUMPAD1 = 0x61 (&H61)	Key « 1 » on the num pad
VK_NUMPAD2 = 0x62 (&H62)	Key « 2 » on the num pad
VK_NUMPAD3 = 0x63 (&H63)	Key « 3 » on the num pad
VK_NUMPAD4 = 0x64 (&H64)	Key « 4 » on the num pad
VK_NUMPAD5 = 0x65 (&H65)	Key « 5 » on the num pad
VK_NUMPAD6 = 0x66 (&H66)	Key « 6 » on the num pad
VK_NUMPAD7 = 0x67 (&H67)	Key « 7 » on the num pad
VK_NUMPAD8 = 0x68 (&H68)	Key « 8 » on the num pad
VK_NUMPAD9 = 0x69 (&H69)	Key « 9 » on the num pad
VK_MULTIPLY = 0x6A (&H6A)	Key « * » on the num pad
VK_ADD = 0x6B (&H6B)	Key « + » on the num pad
VK_SEPARATOR = 0x6C (&H6C)	Key « / » on the num pad
VK_SUBTRACT = 0x6D (&H6D)	Key « - » on the num pad
VK_DECIMAL = 0x6E (&H6E)	Key « . » on the num pad
VK_DIVIDE = 0x6F (&H6F)	
VK_F1 = 0x70 (&H70)	Key « F1 »
VK_F2 = 0x72 (&H72)	Key « F2 »
VK_F3 = 0x73 (&H73)	Key « F3 »
VK_F4 = 0x74 (&H74)	Key « F4 »
VK_F5 = 0x75 (&H75)	Key « F5 »
VK_F6 = 0x76 (&H76)	Key « F6 »
VK_F7 = 0x77 (&H77)	Key « F7 »
VK_F8 = 0x78 (&H78)	Key « F8 »
VK_F9 = 0x79 (&H79)	Key « F9 »
VK_F10 = 0x7A (&H7A)	Key « F10 »
VK_F11 = 0x7B (&H7B)	Key « F11 »
VK_F12 = 0x7C (&H7C)	Key « F12 »
VK_F13 = 0x7D (&H7D)	Key « F13 »
VK_F14 = 0x7E (&H7E)	Key « F14 »
VK_F15 = 0x7F (&H7F)	Key « F15 »
VK_F16 = 0x80 (&H80)	Key « F16 »
VK_F17 = 0x81 (&H81)	Key « F17 »
VK_F18 = 0x82 (&H82)	Key « F18 »
VK_F19 = 0x83 (&H83)	Key « F19 »

VK_ F20 = 0x84 (&H84)	Key « F20 »
VK_ F21 = 0x85 (&H85)	Key « F21 »
VK_ F22 = 0x86 (&H86)	Key « F22 »
VK_ F23 = 0x87 (&H87)	Key « F23 »
VK_ F24 = 0x88 (&H88)	Key « F24 »

Return value: the function returns 1 if no problem occurred; else it returns 0.

InsertMobiScriptMenuItem(*SessionId* As Long, *MenuItemText* As String, *AccelControlKey* As String, *AccelKeyCode* As String, *IconFileName* As String, *MenuRank* As Long, *ItemRank* As Long, *DLLFileName* As String, *ClassName* As String, *FunctionName* As String) As Boolean

This function inserts a menu item (a command) into the MobiScript application.
To know the meaning of the parameters, please refer to the **InsertMenuItem** function described above.

Return value: the function returns 1 if no problem occurred; else it returns 0.

CenterWindow(*WindowHandle* As Long) As Boolean

This function moves to the centre of the screen the window which handle is passed through the *WindowHandle* parameter.

This function is useful when developing wizards to control the position of its window on the screen.

Return value: the function always returns 1.

GetCatalogsList(*SessionId* As Long, *AllItems* As, *Format* As String) As String

Return value: this function returns the list of the catalogues that are installed on the system (.CAT files that are contained in the directory specified in the CatalogsDir entry of the [Local] or [Remote] section of the active SPACE.INI file).

If the *AllItems* parameter is equal to True, all the catalogues are returned ; else, only the valid catalogues appear in the return string (catalogues that would appear in the KitchenDraw catalogues option box).

The list presents itself as a characters strings sequence, each characters string corresponding to a particular catalogue. The content of each characters string is specified by the *Format* parameter which is a characters string mixing characters and parameters. If you wish to separate the various characters strings with a given character, you have to put it into the *Format* string.

CatalogGetName (*CatalogFileName* As String) As String

Return value: this function returns the catalogue name (30 characters maximum) whose file name (including the full access path) has been passed in the *CatalogFileName* parameter.

CatalogGetCode (*CatalogFileName* As String) As String

Return value: this function returns the catalogue code whose file name (including the full access path) has been passed in the *CatalogFileName* parameter.

CatalogGetPackCode (*CatalogFileName* As String) As String

Return value: this function returns the pack code (2 characters) to which belongs the catalogue whose file name (including the full access path) has been passed in the *CatalogFileName* parameter.

CatalogConvertToCCA (*CatalogFileName* As String) As String

This function converts a normal KitchenDraw catalogue (.CAT) to a compressed catalogue of .CCA format. The catalogue file name (including the full access path) is passed in the *CatalogFileName* parameter.

The .CCA compressed catalogues are produced by MobiScript using the « File | Compress for Upload » command. Their destiny is to be uploaded to the catalogues update web site.

Return value: the function returns 1 if no problem occurred; else it returns 0.

CatalogGetSectionsList(*SessionId* As Long, *CatalogFileName* As String, *AllItems* As, *Format* As String) As String

Return value: this function returns the list of the sections belonging to the catalogue which name (with or without the full path and with or without the .CAT extension) is specified in the *CatalogFileName* parameter. If the *AllItems* parameter is equal to true, all the sections are returned ; else, only the sections that are visible in the KitchenDraw catalogues window are returned (sections not having a “@” character as the first character of their name). Furthermore, the section names are neither translated into the current language nor filtered (the name extensions that appear after the « @ » character are not removed).

The list presents itself as a characters strings sequence, each characters string corresponding to a particular section. The content of each characters string is specified by the *Format* parameter which is a characters string mixing characters and parameters. If you wish to separate the various characters strings with a given character, you have to put it into the *Format* string.

CatalogGetBlocksList(*SessionId* As Long, *CatalogFileName* As String, *SectionRank* As Long, *AllItems* As, *Format* As String) As String

Return value: this function returns the list of the blocks belonging to the section of rank *SectionRank* in the catalogue which name (with or without the full path and with or without the .CAT extension) is specified in the *CatalogFileName* parameter.

If the *AllItems* parameter is equal to true, all the blocks are returned ; else, only the blocks that are visible in the KitchenDraw catalogues window are returned (blocks not having a “@” character as the first character of their name). Furthermore, the block names are neither translated into the current language nor filtered (the name extensions that appear after the « @ » character are not removed).

The list presents itself as a characters strings sequence, each characters string corresponding to a particular model. The content of each characters string is specified by the *Format* parameter which is a characters string mixing characters and parameters. If you wish to separate the various characters strings with a given character, you have to put it into the *Format* string.

CatalogGetArticlesList(*SessionId* As Long, *CatalogFileName* As String, *BlockRank* As Long, *AllItems* As, *Format* As String) As String

Return value: la this function returns the list of the articles belonging to the block of rank *BlockRank* in the catalogue which name (with or without the full path and with or without the .CAT extension) is specified in the *CatalogFileName* parameter.

At the moment the *AllItems* parameter has no influence on the return value. However, it may be used in a future version.

The list presents itself as a characters strings sequence, each characters string corresponding to a particular model. The content of each characters string is specified by the *Format* parameter which is a characters string mixing characters and parameters. If you wish to separate the various characters strings with a given character, you have to put it into the *Format* string.

CatalogGetModelsList(*SessionId* As Long, *CatalogFileName* As String, *AllItems* As, *Format* As String) As String

Return value: this function returns the list of the models belonging to the catalogue which name (with or without the full path and with or without the .CAT extension) is specified in the *CatalogFileName* parameter. If the *AllItems* parameter is equal to true, all the models are returned ; else, only the models that are visible in the KitchenDraw catalogues window are returned (models not having a “@” character as the first character of their name). Furthermore, the model names are neither translated into the current language nor filtered (the name extensions that appear after the « @ » character are not removed).

The list presents itself as a characters strings sequence, each characters string corresponding to a particular model. The content of each characters string is specified by the *Format* parameter which is a characters string mixing characters and parameters. If you wish to separate the various characters strings with a given character, you have to put it into the *Format* string.

CatalogGetFinishTypeName (*SessionId* As Long, *CatalogFileName* As String, *FinishTypeRank* As Long) As String

Return value: this function returns the name of the finish type of rank *FinishTypeRank* in the catalogue whose file name (including the full access path) has been passed in the *CatalogFileName* parameter.

CatalogGetFinishCodeAndName (*SessionId* As Long, *CatalogFileName* As String, *FinishTypeRank* As Long, *FinishRank* As Long) As String

Return value: this function returns, separated with « ; ; », the code and the name of the finish of rank *FinishRank* corresponding to the finish type of rank *FinishTypeRank* in the catalogue whose file name (including the full access path) has been passed in the *CatalogFileName* parameter.

CatalogModifyFinishesConfig (*SessionId* As Long, *CatalogFileName* As String, *FinishTypesList* As String, *FinishesList* As String, *ModifiedLine* As Long, *NewFinish* As Long) As String

Return value: this function returns a characters string representing the finishes configuration to be presented following the modification of the configuration passed in the *FinishTypesList* and *FinishesList* parameters by a finish change (*NewFinish*) carried out at the level of the finish type of rank *ModifiedLine*.

The characters string has the following format :

«-1,10005,10006,10007,20022,20023;;1,2,1,4,9,7».

At the left hand side of « ; ; », we can find the finishes types separated with a coma. It's possible to get the name of the finishes types corresponding to these values thanks to the **CatalogGetFinishTypeName** function of the **Appli** class.

The -1 code represents the front model; the 1XXXX codes represent the model finishes types and the 2XXXX codes represent the family finishes types.

At the right hand side of « ; ; », we can find the ranks (numbered from 0 and separated with a coma) of the finishes corresponding respectively to the finishes types listed previously. Each value represents the rank of the selected finish in the list of the available finishes for the corresponding finish type.

The « Catalog » class

The functions of the **Catalog** class allow reading and writing data from and into KitchenDraw catalogues by programme.

Thanks to them, it will be possible to generate or update all or part of a KitchenDraw catalogue with data coming from a corporate database.

In the other direction, they will allow to read data from KitchenDraw catalogues to “render” them another way (generation of paper catalogues or on-line catalogues) or to use them in other applications and particularly in “plug-in” functions or wizards.

The way to use these functions is quite simple:

First, it's necessary to create a new catalogue (**FileNew**) or to load an existing catalogue into memory (**FileLoad**) specifying the password if the catalogue is password protected;

then, it becomes possible to read or write data in tables just like one could do it manually with MobiScript.

At last, if the modifications must be saved, the **FileSave** command can be run.

The functions of the **Catalog** class refer sometimes to the notion of *cluster*.

A *cluster* in a MobiScript table represents a set of lines belonging to the same higher level entity.

For example, in the “Blocks” table, the lines corresponding to the blocks belonging to the same section compose one cluster. They are displayed with a yellow background colour to better identify them. The *cluster* of the blocks referring to the first section of the catalogue is the cluster of rank 1 and so and.

The functions that have a *ClusterRank* parameter consider the *LineRank* parameter as being relative to the specified cluster.

If the *ClusterRank* parameter is equal to 0, then the functions consider the *LineRank* parameter as being absolute (i.e. relative to the beginning of the table).

The cluster ranks (*ClusterRank* parameter) and the line ranks (*LineRank* parameter) are always numbered starting from 1.

Here is the list of the tables that can be accessed and the corresponding *Table* values that must be passed as a parameter to the functions:

Table Value	Table description
CATTABLE_CONSTANTS = 0	Constantes
CATTABLE_SECTIONS = 1	Sections
CATTABLE_BLOCKS = 2	Blocks
CATTABLE_ARTICLES = 3	Articles
CATTABLE_PRICES = 4	Prices (contains purchase prices when the catalogue's « price » type is « Purchase » or selling prices when the catalogue's « price » type is « Sell » or « Purchase and sell »)
CATTABLE_PURCHASEPRICES = 5	Purchase prices (used only when the catalogue's « price » type is « Purchase and sell »)
CATTABLE_REFERENCES = 6	References
CATTABLE_TEXTURES = 7	Textures
CATTABLE_MODELS = 8	Models (front models)
CATTABLE_MODELFINISHTYPES = 9	Model finish types
CATTABLE_MODELFINISHES = 10	Model finishes
CATTABLE_MODELHANDLES = 11	Model handles
CATTABLE_FAMILIES = 12	Families
CATTABLE_FAMILYFINISHTYPES = 13	Family finish types
CATTABLE_FAMILYFINISHES = 14	Family finishes
CATTABLE_2DENTITIES = 15	2D entities
CATTABLE_2DPRIMITIVES = 16	2D Primitives
CATTABLE_3DENTITIES = 17	3D Entities
CATTABLE_3DPRIMITIVES = 18	3D Primitives
CATTABLE_RESOURCES = 19	Embedded files (textures, ...)

The **Catalog** class functions list is the following:

FileNew (*SessionId* As Long, *BaseCatalogFileName* As String, *MeasurementUnit* As Long, *CatalogType* As Long) As Boolean

This function creates a new catalogue in memory. If a base catalogue has been chosen, the new catalogue incorporates the data from the base catalogue whose name (without path and without extension) is specified in the *BaseCatalogFileName* parameter.

The *MeasurementUnit* parameter allows specifying the unit of measurement used in the catalogue for dimensions.

The *CatalogType* parameter indicates if the created catalog is a standard catalog (*CatalogType*=0) or a base or intermediate catalog (*CatalogType*=1).

The available units of measurement list is the following :

Unit of measurement value	Unit of measurement description
CATUNIT_MILLIMETRE = 0	Millimetre
CATUNIT_MILLIMETRE_ONEDEC = 8	Millimetre with one decimal
CATUNIT_MILLIMETRE_TWODEC = 9	Millimetre with two decimals
CATUNIT_CENTIMETRE_ONEDEC = 1	Centimetre with one decimal
CATUNIT_INCHDECIMAL = 2	Inch (decimal)
CATUNIT_INCHFRACT32 = 3	Inch (1/32)
CATUNIT_INCHFRACT16 = 4	Inch (1/16)
CATUNIT_INCHFRACT8 = 5	Inch (1/8)

For a given session, only one catalogue can be loaded in memory at the same time. The created catalogue is called the **current catalogue**.

Return value: if there is enough memory to create the catalogue, the function returns 1; else it returns 0.

FileLoad (*SessionId* As Long, *CatalogFileName* As String, *Password* As String) As Boolean

This function loads in memory the catalogue whose file name (including the full access path) is specified in the *CatalogFileName* parameter.

For a given session, only one catalogue can be loaded in memory at the same time. The loaded catalogue is called the **current catalogue**.

Return value: if the catalogue file is found and if there is enough memory to load it in memory, the function returns 1; else it returns 0.

FileSave (*SessionId* As Long, *CatalogFileName* As String) As Boolean

This function saves the current catalogue to the disk.

The *CatalogFileName* parameter contains the file name (including the full access path) in which the current catalogue will be saved.

Attention: if the catalog is a base or intermediate catalog (*CatalogType*=1 when creating the catalogue with the **FileNew** function) then, the catalog's file name must begin with a « # » character.

Return value: the function returns 1 if the operation has been completed successfully; else it returns 0.

FileExportImage (*SessionId* As Long, *ArticleRank* As Long, *Opened* As Long, *ViewMode* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *BackGroundColor* As String, *Transparent* As Boolean, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image corresponding to the article of rank *ArticleRank* in the articles table. The generated file can be a .JPG, .BMP, .GIF, .PNG or .TIF file whose name (including the full access path) is specified in the *ImageFileName* parameter.

If the *Opened* parameter is equal to 0, the article is shown closed; else it's shown opened.

The *ViewMode* parameter indicates what kind of view type must be generated.

<i>ViewMode</i> value	Description
VIEWMODE_2D = 0	Top view
VIEWMODE_ELEVATION = 1	Wireframe elevation
VIEWMODE_REALELEVATION = 2	Realistic elevation
VIEWMODE_3D = 3	Wireframe perspective
VIEWMODE_REAL3D = 4	Realistic perspective
VIEWMODE_PHOTO3D = 5	Photorealistic perspective

The *XRes* and *YRes* parameters specify the resolution in pixels of the exported image.

The *BackGroundColor* parameter defines the colour of the image background. It's a characters string representing the decimal values (between 0 and 255) of the red component, then the green component and then the bleu component, separated with a coma like in the following example: « 255,128,0 ».

If the *BackGroundColor* parameter is empty, the background colour will be white.

If the *Transparent* parameter is set as True, the background colour will be considered as transparent (for the .GIF, .PNG and .TIF formats only).

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

<i>AntiAliasing</i> value	Description
---------------------------	-------------

1	No anti-aliasing
2	The image is first generated at a resolution of 2xXRes and 2xYRes and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.
3	The image is first generated at a resolution of 3xXRes and 3xYRes and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportModellImage (*SessionId* As Long, *FinishTypesList* As String, *FinishesList* As String, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image corresponding to the front model specified in the *FinishTypesList* and *FinishesList* parameters. The generated file can be a .JPG, .BMP, .GIF, .PNG or .TIF file whose name (including the full access path) is specified in the *ImageFileName* parameter. For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *XRes* and *YRes* parameters specify the resolution in pixels of the exported image. The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportFamilyFinishImage (*SessionId* As Long, *FinishTypeRank* As Long, *FinishRank* As Long, *TextureRank* as Long, *Width* As Long, *Depth* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image corresponding to the family finish texture specified in the *FinishTypesRank*, *FinisheRank* and *TextureRank* parameters. The generated file can be a .JPG, .BMP, .GIF, .PNG or .TIF file whose name (including the full access path) is specified in the *ImageFileName* parameter. Normally, *TextureRank* should be equal to 1 but it can take another value if the finish type is defined by several texture columns (for example the "WT colour" finish type which has "Top colour" and "Edge colour" columns) and if a "secondary" colour must be exported.

The *Width* and *Depth* parameters specify the size (in catalogue units of measurement) of the material sample represented in the exported image.

The *XRes* and *YRes* parameters specify the resolution in pixels of the exported image.

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportModelFinishImage (*SessionId* As Long, *FinishTypeRank* As Long, *FinishRank* As Long, *TextureRank* as Long, *Width* As Long, *Depth* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image corresponding to the model finish texture specified in the *FinishTypesRank*, *FinisheRank* and *TextureRank* parameters. The generated file can be a .JPG, .BMP, .GIF, .PNG or .TIF file whose name (including the full access path) is specified in the *ImageFileName* parameter. Normally, *TextureRank* should be equal to 1 but it can take another value if the finish type is defined by several texture columns (for example the "Front colour" finish type which has "Centre 1" and "Centre 2" columns) and if a "secondary" colour must be exported.

The *Width* and *Depth* parameters specify the size (in catalogue units of measurement) of the material sample represented in the exported image.

The *XRes* and *YRes* parameters specify the resolution in pixels of the exported image.

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportResource (*SessionId* As Long, *LineRank* As Long, *ResourceFileName* As String) As Boolean

This function generates a file from the resource (normally pictures but it could be any file) embedded in the current catalogue.

The name of the generated file (including the full access path) is specified in the *ResourceFileName* parameter.

Embedded resources are for example the textures files indicated in the "JPG or BMP files" column in the Textures table when the corresponding "Embedded" checkbox is checked.

However, in theory it's possible to embed any kind of files in a catalogue (images, sounds, videos, etc.) and to extract them with this function when needed.

For example, handle photos could be embedded in a catalogue. Then, an application using this SDK could use the **FileExportResource** function to export these photos and show them in a "graphical combo box".

The *LineRank* parameter which indicates the rank in the CATTABLE_RESOURCES table of the resource to be exported can be obtained thanks to the **TableGetLineRankFromName** function if the name of the resource is known.

Notice : it's possible to embed a file in a catalogue by programme using the **TableSetLineInfo** function which is documented later. The name of the file to be embedded (including the full access path) must be placed in the *InfoValue* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

IsLoaded(*SessionId* As Long) As Boolean

This function tells if a catalogue is currently loaded into memory and consequently if it is possible to call functions that apply to the current catalogue (most of the functions belonging to the Catalog class).

Return value: the function returns 1 if a catalogue is loaded into memory; else it returns 0.

GetActiveTable(*SessionId* As Long) As Long

Return value: the function returns the integer value identifying the table that is currently selected in the « Entity » MobiScript options box (that is to say the table that is shown in MobiScript).

GetInfo (*SessionId* As Long, *InfoType* As Long) As String

This function returns various information about the current catalogue (catalogue name, catalogue code, base catalogue used, etc.).

In fact, the data that can be retrieved with this function are all the data presented in the « Catalog | Information » MobiScript dialog box plus few internal data.

Be carefull, if the information depends on the language like the catalogue name, it will be returned in the current language.

The required information is specified in the *InfoType* parameter and here is the possible values:

<i>InfoType</i> value	Description
CATINFO_NAME = 0	Name of the catalogue (30 characters maxi)
CATINFO_CODE = 1	Code of the catalogue (8 characters maxi)
CATINFO_PASSWORD = 10	Password controlling the access to the catalogue
CATINFO_TYPE = 6	Type of the catalogue (rank in the « Type » drop down list in the MobiScript « Catalog Information » dialog box) (example : 1 stands for « Kitchens and bathrooms », 2 for « Closets », 3 for « Appliances », etc.)
CATINFO_SUBTYPE = 8	Sub-type of the catalogue (rank in the « Sub-type » drop down list) (example : 1 stands for « None », 2 for « Fronts », 3 for « Carcasses », 4 for « Accessories » if Type equals 1)
CATINFO_BASECATALOG = 2008	Base catalogue used (30 characters maxi)
CATINFO_LANGUAGE = 17	Working language of the catalogue (example : « FRA », « ENG », etc.)
CATINFO_UNIT = 15	Measuring unit used in the catalogue (rank in the « Measuring unit » drop down list) (example : 1 stands for « mm », 2 for « cm », etc.)
CATINFO_PRICETYPE = 18	Price type in the catalogue (rank in the « Price » drop down list) (example : 1 stands for « Purchase », 2 for « Sell », 3 for « Purchase and sell »)
CATINFO_CURRENCY = 2	Symbol of the currency used to represent the prices (example : « EUR », « USD », « GBP »)
CATINFO_VATINCLUDED = 1008	Equals « 1 » if the selling prices are entered including taxes, « 0 » otherwise
CATINFO_VATRATE = 9	Taxes rate if the selling prices are entered including taxes (example : « 19.6 »)
CATINFO_CREATIONDATE = 19	Catalogue creation date (YYYYMMDD format)
CATINFO_MODIFICATIONDATE = 20	Catalogue last modification date (YYYYMMDD format)
CATINFO_STARTVALIDITYDATE = 21	Catalogue validity start date (YYYYMMDD format)
CATINFO_ENDVALIDITYDATE = 22	Catalogue validity end date (YYYYMMDD format)
CATINFO_SELLINGPRICESWOPW = 3003	Equals « 1 » if the selling prices table of the catalogue can be accessed (read and write) even without any password

Return value: the function returns the characters string representing the required information.

SetInfo (*SessionId* As Long, *InfoValue* As String, *InfoType* As Long) As Boolean

This function writes the *InfoValue* content into various information of the current catalogue.

Please, refer to the **GetInfo** function for a list of the information.

Attention: the **Measuring unit** information is read only information.

Return value: the function returns 1 if no problem occurred; else it returns 0.

TableAddLines (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *NbLines* As Long) As Long

This function adds a sequence of *NbLines* empty lines to the table defined in the *Table* parameter at the end of the cluster defined in the *ClusterRank* parameter.

Executing this function has the same effect than clicking on the “Add” button located at the bottom of the MobiScript Window.

Return value: the function returns the rank of the first added line relatively to the specified cluster.

It returns -1 if the *ClusterRank* parameter is invalid.

TableInsertLines (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *NbLines* As Long) As Long

This function inserts a sequence of *NbLines* empty lines into the table defined in the *Table* parameter. The lines are inserted into the cluster defined in the *ClusterRank* parameter and before the line defined in the *LineRank* parameter.

Executing this function has the same effect than selecting a line and clicking on the "Insert" button located at the bottom of the MobiScript Window.

Return value: the function returns the rank of the first inserted line relatively to the specified cluster. It returns -1 if the *ClusterRank* or the *LineRank* parameter is invalid.

TableDeleteLines (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *NbLines* As Long) As Long

This function removes a sequence of *NbLines* lines from the table defined in the *Table* parameter. The lines are removed from the cluster defined in the *ClusterRank* parameter and starting from the line defined in the *LineRank* parameter.

Executing this function has the same effect than selecting a sequence of lines in the table and clicking on the "Delete" button located at the bottom of the MobiScript Window.

Return value: the function returns the rank of the first deleted line relatively to the specified cluster. It returns -1 if the *ClusterRank* or the *LineRank* parameter is invalid.

TableGetLineInfo (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *InfoColumnRank* As Long) As String

This function reads the information contained in the cell corresponding to the column whose rank is *InfoColumnRank* (starting from 1) and the line whose rank is *LineRank* (starting from 1) in the *ClusterRank* cluster of the *Table* table in the current catalogue.

Attention, each information which is language dependant like a name or a description is returned in the current language.

If the *ClusterRank* parameter equals CLUSTER_FROM_ITEM (that is to say 0), then the *LineRank* parameter is considered as being a rank in the whole table and not a rank in the cluster.

Return value: the function returns the characters string representing the required information. It returns an empty characters string if the *ClusterRank* or the *LineRank* parameter is invalid.

TableSetLineInfo (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *InfoColumnRank* As Long, *InfoValue* As String) As Boolean

This function writes the *InfoValue* characters string in the cell corresponding to the column whose rank is *InfoColumnRank* (starting from 1) and the line whose rank is *LineRank* (starting from 1) in the *ClusterRank* cluster of the *Table* table in the current catalogue.

Attention, each information which is language dependant like a name or a description must be provided in the current language.

If the *ClusterRank* parameter equals CLUSTER_FROM_ITEM (that is to say 0), then the *LineRank* parameter is considered as being a rank in the whole table and not a rank in the cluster.

Return value: the function returns 1 if no problem occurred; else it returns 0.

TableGetLine (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long) As String

This function reads in a unique characters string all the information contained in the cells composing the line whose rank is *LineRank* (starting from 1) in the *ClusterRank* cluster of the *Table* table in the current catalogue.

Attention, each information which is language dependant like a name or a description is returned in the current language.

If the *ClusterRank* parameter equals CLUSTER_FROM_ITEM (that is to say 0), then the *LineRank* parameter is considered as being a rank in the whole table and not a rank in the cluster.

Return value: the function returns the characters string representing the information composing the specified line. The various individual information are separated with a « tab » character.
It returns an empty characters string if the *ClusterRank* or the *LineRank* parameter is invalid.

TableGetLineRankFromCode(*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *Code* As String, *BackScan* As Boolean) As Long

This function finds, in the *Table* table, the rank of the line whose code is the same than *Code*.
The operation starts from the *LineRank* line of the *ClusterRank* cluster and goes in the ascending order of the line ranks if *BackScan* equals 0 (from the end to the start if *BackScan* equals 1).

Return value: the function returns the rank of the found line relatively to the specified cluster and -1 if no line has been found.

TableGetLineRankFromName (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *Name* As String, *BackScan* As Boolean) As Long

This function finds, in the *Table* table, the rank of the line whose name is the same than *Name*.
The operation starts from the *LineRank* line of the *ClusterRank* cluster and goes in the ascending order of the line ranks if *BackScan* equals 0 (from the end to the start if *BackScan* equals 1).
Attention, the *Name* parameter must be written in the current language.

Return value: the function returns the rank of the found line relatively to the specified cluster and -1 if no line has been found.

TableGetLinesNb (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long) As Long

Return value: the function returns the number of lines composing the *ClusterRank* cluster of the *Table* table.
It returns -1 if the *ClusterRank* value is invalid.

TableGetClusterRankFromLineRank (*SessionId* As Long, *Table* As Long, *LineRank* As Long) As Long

Return value: the function returns the rank of the cluster to which belongs the line of rank *LineRank* in the *Table* table. It returns -1 if the *LineRank* value is invalid.

TableGetFirstLineRankFromClusterRank (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long) As Long

Return value: the function returns the rank in the *Table* table of the first line belonging to the *ClusterRank* cluster. The function returns 0 if the cluster is empty and -1 if the *ClusterRank* value is invalid.

TableGetColumnsNb (*SessionId* As Long, *Table* As Long) As Long

Return value: the function returns the number of columns in the *Table* table.

TableGetColumnTitle (*SessionId* As Long, *Table* As Long, *ClusterRank* As Long, *LineRank* As Long, *ColumnNb* As Long, *ColumnRank* As Long) As String

Return value: the function returns the characters string representing the title of the column whose rank in the *Table* table is *ColumnRank* and this at the level of the line defined by the *LineRank* and *ClusterRank* parameters.

Indicating the number of columns in the table can reduce dramatically the processing time. This can be very valuable if the function has to be called many times.

If the *ColumnNb* parameter equals 0, the determination of the table columns number is carried out inside the function.

TableGetActiveLineRank(*SessionId* As Long, *Table* As Long) As Long

Return value: the function returns the rank (starting from 1) of the active line in the *Table* table. The active line is the one corresponding to the cell that receives the characters that are entered from the keyboard. Attention, the returned rank is the rank in the whole table. It may not correspond to the line number displayed in the first column of the table in MobiScript if you have checked the “Left” or the “Right” checkbox in the CATTABLE_ARTICLES, CATTABLE_PRICES, CATTABLE_PURCHASEPRICES or CATTABLE_REFERENCES table.

Notice : This function can be useful when developing MobiScript extension functions (for example a script wizard) to know which line the function should apply to.

TableSelectionGetFirstLineRank(*SessionId* As Long) As Long

Return value: the function returns the rank (starting from 1) of the first selected line in the MobiScript active table.

The function returns -1 if the “Left” or the “Right” checkbox is checked in the CATTABLE_ARTICLES, CATTABLE_PRICES, CATTABLE_PURCHASEPRICES or CATTABLE_REFERENCES table.

Notice : Along with the *TableSelectionGetLinesNb* function, this function can be useful when developing MobiScript extension functions in order to scan the selected lines in the active table so as to apply a particular treatment to them.

TableSelectionGetLinesNb(*SessionId* As Long) As Long

Return value: the function returns the number of selected lines in the MobiScript active table. The function returns 0 if the “Left” or the “Right” checkbox is checked in the CATTABLE_ARTICLES, CATTABLE_PRICES, CATTABLE_PURCHASEPRICES or CATTABLE_REFERENCES table.

Notice : Along with the *TableSelectionGetFirstLineRank* function, this function can be useful when developing MobiScript extension functions in order to scan the selected lines in the active table so as to apply a particular treatment to them.

GetClusterRankFromModelFinishType (*SessionId* As Long, *ModelRank* As Long, *FinishType* As String) As Long

This function retrieves in the « Front model finishes » table the rank of the cluster which is defined by the *ModelRank* parameter and the *FinishType* parameter.

The *ModelRank* parameter represents the rank of the concerned front model in the « Font models » table. The *FinishType* parameter must be one of the characters strings that can be found in the “Finish type” combo box of the “Front model finishes” table.

Attention, these characters strings depend on the current language.

The cluster rank which is returned by this function is intended to be used as a parameter for the functions of the **Catalog** class that require this type of parameter.

Return value: the function returns the cluster rank and -1 if the *ModelRank* parameter or the *FinishType* characters string is invalid.

GetClusterRankFromModelHandleType(*SessionId* As Long, *ModelRank* As Long, *HandleType* As String) As Long

This function retrieves in the « Front model handles » table the rank of the cluster which is defined by the *ModelRank* parameter and the *HandleType* parameter.

The *ModelRank* parameter represents the rank of the concerned front model in the « Font models » table.

The *HandleType* parameter must be one of the characters strings that can be found in the “Handle type” combo box of the “Front model handles” table.

Attention, these characters strings depend on the current language.

The cluster rank which is returned by this function is intended to be used as a parameter for the functions of the **Catalog** class that require this type of parameter.

Return value: the function returns the cluster rank and -1 if the *ModelRank* parameter or the *HandleType* characters string is invalid.

GetClusterRankFromFamilyFinishType(*SessionId* As Long, *FamilyRank* As Long, *FinishType* As String) As Long

This function retrieves in the « Family finishes » table the rank of the cluster which is defined by the *FamilyRank* parameter and the *FinishType* parameter.

The *FamilyRank* parameter represents the rank of the concerned family in the « Families » table.

The *FinishType* parameter must be one of the characters strings that can be found in the « Finish type » combo box of the “Family finishes” table.

Attention, these characters strings depend on the current language.

The cluster rank which is returned by this function is intended to be used as a parameter for the functions of the **Catalog** class that require this type of parameter.

Return value: the function returns the cluster rank and -1 if the *FamilyRank* parameter or the *FinishType* characters string is invalid.

DicoUpdate(*SessionId* As Long) As Boolean

This function adds to the catalogue translation dictionary table all the texts (section names, block names and descriptions, front model names and descriptions, finishes, etc.) that have been added to this catalogue since the last update. It also removes all the texts corresponding to the entities that have been removed from the catalogue.

An update of the catalogue translation dictionary table is also carried out automatically when the translation window is loaded in MobiScript.

Return value: the function returns 1 if no problem occurred; else it returns 0.

DicoGetLinesNb(*SessionId* As Long) As Long

Retour : the function returns the number of lines present in the translation dictionary table corresponding to the current catalogue.

DicoSetStringFromLineRank(*SessionId* As Long, *LineRank* As Long, *LanguageCode* As String, *Text* As String) As Boolean

This function writes into the translation dictionary table corresponding to the current catalogue the *Text* characters string representing the translation into the language defined in *LanguageCode* of the characters string located at the *LineRank* line.

Return value: the function returns 1 if no problem occurred; else it returns 0.

DicoGetStringFromLineRank(*SessionId* As Long, *LineRank* As Long, *LanguageCode* As String) As String

This function reads from the translation dictionary table corresponding to the current catalogue the characters string representing the translation into the language defined in *LanguageCode* of the characters string located at the *LineRank* line.

Return value: the function returns the translated characters string or an empty characters string if *LineRank* is invalid.

DicoSetStringFromKey(*SessionId* As Long, *KeyString* As String, *LanguageCode* As String, *TranslatedString* As String) As Boolean

This function writes into the translation dictionary table corresponding to the current catalogue the *TranslatedString* characters string representing the translation into the language defined in the *LanguageCode* parameter of the *KeyString* characters string.

Return value: the function returns 1 if no problem occurred; else it returns 0.

DicoGetStringFromKey(*SessionId* As Long, *KeyString* As String, *LanguageCode* As String) As String

This function reads from the translation dictionary table corresponding to the current catalogue the characters string representing the translation into the language defined in *LanguageCode* of the characters string located at the *LineRank* line.

Return value: the function returns the translated characters string or the *KeyString* characters string itself if *KeyString* can't be found in the translation dictionary.

The « Scene » class

The functions of the **Scene** class allow reading and writing data from and into KitchenDraw scenes by programme.

Thanks to them, it will be possible to develop suppliers order files generation modules, or KitchenDraw extensions (“plug-in” functions that are executed automatically when certain events are fired or that are associated to specific menu commands).

These functions are also used to develop wizards which are specific programmes to configure highly “parametrable” objects like special worktops or cabinets, staircases, conservatories, etc.

The functions belonging to this class and which address a catalogue thanks to its file name without the extension assume that this catalogue is loaded into memory or at least that it's located in the directory which is specified in the « Dir » entry of the « Catalogs » section of the SPACE.INI file.

The SPACE.INI file should be located in the same directory than DV.DLL, that is to say in the application's directory or in the Windows/System directory.

The **Scene** class functions list is the following:

FileLoad (*SessionId* As Long, *SceneFileName* As String) As Boolean

This function loads into the memory a scene whose file name (including the full access path) is specified in the *SceneFileName* parameter.

For a given session, only one scene can be loaded at a time. The loaded scene is called the **current scene**.

Return value: if the scene file has been found and if there is enough memory to load it, the function returns 1; else it returns 0.

FileSave (*SessionId* As Long, *SceneFileName* As String) As Boolean

This function saves the current scene on a storage unit.

The *SceneFileName* parameter contains the file name (including the full access path) that is written.

Return value: if the save operation has been carried out without any problem, the function returns 1; else it returns 0.

FileLoadCatalogScene (*SessionId* As Long, *CatalogFileName* As String) As Boolean

This function loads into the memory the *presentation scene* corresponding to the catalogue whose name (including the full access path) is specified in the *CatalogFileName* parameter.

The *presentation scene* of a catalogue contains the drawing settings for the catalogue blocks images.

This scene is created in the catalogues directory when one of the MobiScript dialog boxes that define the way the blocks images must be drawn is validated (commands of the “Catalogue | 3D drawings” sub-menu). It has the same name than the corresponding catalogue.

If there is no presentation scene for the catalogue, an empty scene is created in memory.

Once loaded, the presentation scene becomes the current scene. Then, you can apply the functions of the **Scene** class described later to read or write information and eventually save it with the **FileSaveCatalogScene** function.

You must load the presentation scene and setup your preferences prior to exporting the catalogue blocks images if you wish to get the images with a particular front model, finish or handle.

Return value: if the catalogue file is found there is enough memory to load the corresponding presentation scene, the function returns 1; else it returns 0.

FileSaveCatalogScene (*SessionId* As Long, *SceneFileName* As String) As Boolean

This function saves on a storage unit the *presentation scene* corresponding to the current catalogue. The *SceneFileName* parameter contains the file name (including the full access path) in which the *presentation scene* will be saved in.

Return value: if the save operation has been carried out without any problem, the function returns 1; else it returns 0.

FileExportBOMList(*SessionId* As Long, *ListFileName* As String, *ListType* As Long, *FormatName* As String, *Factorized* As Boolean, *Concatenate* As Boolean) As Boolean

This function exports a Bill Of Material (BOM) corresponding to the current scene as a text file whose name (including the full access path) is specified in the *ListFileName* parameter.

The *ListType* parameter specifies the Bill Of Material type that should be generated.

<i>ListType</i> Value	Description
SCNBOMEXPTYPE_ALL = 0	Complete Bill Of Material containing all the scene objects constituent component that is to say the components of type PANC and PANF (panels), PROF (profiles), PIECE (fittings), PROD (product), MO (manpower)
SCNBOMEXPTYPE_PANCUTLIST = 1	Panels cutting list only containing the components of type PANC and PANF (panels)

The *FormatName* parameter specifies the file format to be generated. It represents the name of the software product that is supposed to open or import the exported file.

Here is the list of the file formats that are currently supported:

- SheetLayout
- FastCut
- The Itemizer
- Cut Planner
- Cutting Optimizer
- OptiCoupe IV
- Optimik
- Leonardo
- CutList Plus

A *Factorized* parameter at **True** indicates the BOM list must be factorized that is to say identical lines must be represented by a single line but with a “quantity” information attached.

A *Concatenate* parameter at **True** indicates the BOM list must be appended to the *ListFileName* file (if the file already exists). A *Concatenate* parameter at **False** indicates the file should be cleared first.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportImage (*SessionId* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *BackGroundColor* As String, *Transparent* As Boolean, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports the current view of the current scene (top view, elevation or perspective) as a .JPG, .BMP, .GIF, .PNG or .TIF file whose name (including the full access path) is specified in the *ImageFileName* parameter.

The *XRes* and *YRes* parameters specify the resolution of the exported image in pixels.

The *BackGroundColor* parameter defines the colour of the image background. It's a characters string representing the decimal values (between 0 and 255) of the red component, then the green component and then the bleu component, separated with a coma like in the following example: « 255,128,0 ».

If the *BackGroundColor* parameter is empty, the background colour will be white.

If the *Transparent* parameter is set as True, the background colour will be considered as transparent (for the .GIF, .PNG and .TIF formats only).

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

<i>AntiAliasing</i> value	Description
1	No anti-aliasing
2	The image is first generated at a resolution of 2xXRes and 2xYRes and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.
3	The image is first generated at a resolution of 3xXRes and 3xYRes and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportSelectionMap (*SessionId* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long) As Boolean

This function exports an image representing the **selection map** of the current view of the current scene (top view, elevation or perspective) as a .BMP file whose name (including the full access path) is specified in the *ImageFileName* parameter.

This image doesn't provide a realistic representation of the current view but it allows to determine, thanks to the Cette fonction permet d'exporter une image représentant la **carte de sélection** de la vue en cours (vue de dessus, élévation ou perspective) sous la forme d'un fichier .BMP dont le nom (avec le chemin complet) est spécifié dans le paramètre *Image*.

Cette image ne donne pas une représentation réaliste de la vue en cours mais permet de déterminer, grâce à la fonction **FileGetPointedObject**, à quel objet de la scène correspond chaque point (x,y) de l'image.

Les paramètres *XRes* et *YRes* permettent de spécifier la résolution en pixels de l'image exportée. Normalement, la « carte de sélection » doit être générée avec la même résolution que l'image servant à représenter la scène.

Retour : la fonction retourne 1 si l'écriture s'est bien passée, 0 sinon.

FileExportSelectionImage (*SessionId* As Long, *SelectionImageFileName* As String, *XRes* As Long, *YRes* As Long) As Boolean

This function exports an image showing the **selected objects** of the current view (top view, elevation or perspective) as a .BMP, .GIF or .PNG file whose name (including the full access path) is specified in the *SelectionImageFileName* parameter.

This image is not intended to provide a realistic representation of the current view but, thanks to an images superimposition operation, to add on top of the current view the selection black squares corresponding to the selected objects.

If the .BMP format is chosen, the selection square points will be drawn with a white colour and a black background.

If the .GIF or .PNG format is chosen, the selection square points will be drawn with a black colour and a white background.

The *XRes* and *YRes* parameters specify the resolution of the exported image in pixels. Normally, the « **selection image** » should be generated with the same resolution than the image representing the current view.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportSelectionOutlineImage (*SessionId* As Long, *PixelXRot* As Long, *PixelYRot* As Long, *Angle* As Double, *ImageFileName* As String, *XRes* As Long, *YRes* As Long) As Boolean

This function exports an image showing the **outline of the selected objects** of the current view (top view, elevation or perspective) as a .BMP, .GIF or .PNG file whose name (including the full access path) is specified in the *ImageFileName* parameter.

This image is not intended to provide a realistic representation of the current view but, thanks to an images superimposition operation, to visualize the outline of the selected objects while moving them with the mouse. If the .BMP format is chosen, the selection square points will be drawn with a white colour and a black background.

If the .GIF or .PNG format is chosen, the selection square points will be drawn with a black colour and a white background.

The *PixelXRot* and *PixelYRot* parameters represent the position in image coordinates (pixels) of the pivot point of a possible rotation (for example the mouse cursor position) and the *Angle* parameter specifies the rotation angle of the outlines in degrees.

The *XRes* and *YRes* parameters specify the resolution of the exported image in pixels. Normally, the « **outline image** » should be generated with the same resolution than the image representing the current view.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportCatalogImage (*SessionId* As Long, *CatalogFileName* As String, *ArticleRef* As String, *HandingType* As Long, *Opened* As Long, *ViewMode* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *BackGroundColor* As String, *Transparent* As Boolean, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image corresponding to the article which reference is *ArticleRef* and having a *HandingType* handing in the *CatalogFileName* catalogue (short file name without extension).

<i>HandingType</i> value	Description
HANDINGTYPE_NONE = 0	No handing
HANDINGTYPE_LEFT = 1	Left direction
HANDINGTYPE_RIGHT = 2	Right direction

The article adopts the generic finishes corresponding to the presentation scene associated with the *CatalogFileName* catalogue.

If the *Opened* parameter equals 0, the article is shown closed; else it's shown opened as far as an opened version of the article has been designed.

The *ViewMode* parameter indicates what view of the article should be generated.

<i>ViewMode</i> value	Description
VIEWMODE_2D	Top view
VIEWMODE_ELEVATION	Wireframe elevation
VIEWMODE_REALELEVATION	Realistic elevation
VIEWMODE_3D	Wireframe perspective
VIEWMODE_REAL3D	Realistic perspective
VIEWMODE_PHOTO3D = 5	Photo-realistic perspective

The image is generated as a .JPG, .BMP, .GIF, .PNG, or .TIF file having a resolution of *XRes* x *YRes* whose name (including the full access path) is specified in the *ImageFileName* parameter.

The *BackGroundColor* parameter defines the colour of the image background. It's a characters string representing the decimal values (between 0 and 255) of the red component, then the green component and then the bleu component, separated with a coma like in the following example: « 255,128,0 ».

If the *BackGroundColor* parameter is empty, the background colour will be white.

If the *Transparent* parameter is set as True, the background colour will be considered as transparent (for the .GIF, .PNG and .TIF formats only).

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

<i>AntiAliasing</i> value	Description
1	No anti-aliasing
2	The image is first generated at a resolution of 2x <i>XRes</i> and 2x <i>YRes</i> and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.
3	The image is first generated at a resolution of 3x <i>XRes</i> and 3x <i>YRes</i> and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.

Return value: the function returns 1 if no problem occurred, 0 if *ArticleRef* has not been found in *CatalogFileName* or if any other problem occurred during the file writing.

FileExportCatalogOutlineImage (*SessionId* As Long, *CatalogFileName* As String, *ArticleRef* As String, *HandingType* As Long, *Width* As Long, *Depth* As Long, *Height* As Long, *Angle* As Double, *Opened* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long) As Boolean

This function exports an image corresponding to the outline of the article which reference is *ArticleRef* and having a *HandingType* handing in the *CatalogFileName* catalogue (short file name without extension).

This image corresponds to the current view of the current scene (top view, elevation or perspective).

It's generated as a .BMP, .GIF or .PNG file whose name (including the full access path) is specified in the *ImageFileName* parameter.

This image is not intended to provide a realistic representation of the article but, thanks to an images superimposition operation, to add on top of the current view the outline of the article which is being dragged. If the .BMP format is chosen, the selection square points will be drawn with a white colour and a black background.

If the .GIF or .PNG format is chosen, the selection square points will be drawn with a black colour and a white background.

The dimensions applied to the article which can be different than the default dimensions setup in the catalogue are specified in the *Width*, *Depth* and *Height* parameters.

If the *Opened* parameter equals 0, the article is shown closed; else it's shown opened as far as an opened version of the article has been designed.

The *XRes* and *YRes* parameters specify the resolution of the exported image in pixels.

Normally, the **article outline image** should be generated with the same resolution than the image representing the current view.

The reference point is located at the centre of the **article outline image**.

The *Angle* parameter specify in degrees a possible rotation angle applied to the article during the « drag & drop » operation.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileExportCatalogImageFromScript (*SessionId* As Long, *CatalogFileName* As String, *Script* As String, *HandingType* As Long, *Width* As Long, *Depth* As Long, *Height* As Long, *Opened* As Long, *ViewMode* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *BackGroundColor* As String, *Transparent* As Boolean, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image representing a block created from a script.

First of all, the function creates a temporary block in the *CatalogFileName* catalogue (short file name without extension) from the *Script* script. Then, it attaches an article to that block. The dimensions of this article are specified in the *Width*, *Depth* and *Height* parameters and the handing in the *HandingType* parameter.

If the *Opened* parameter equals 0, the article is shown closed; else it's shown opened as far as an opened version of the article has been designed.

The *ViewMode* parameter indicates what view of the article should be generated

(please report to the **FileExportCatalogImage** function described earlier for more information).

The article adopts the generic finishes corresponding to the presentation scene associated with the *CatalogFileName* catalogue.

The image is generated as a .JPG, .BMP, GIF, PNG or .TIF file having a resolution of *XRes* x *YRes* whose name (including the full access path) is specified in the *ImageFileName* parameter.

The *BackGroundColor* parameter defines the colour of the image background. It's a characters string representing the decimal values (between 0 and 255) of the red component, then the green component and then the bleu component, separated with a coma like in the following example: « 255,128,0 ».

If the *BackGroundColor* parameter is empty, the background colour will be white.

If the *Transparent* parameter is set as True, the background colour will be considered as transparent (for the .GIF, .PNG and .TIF formats only).

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileGetSelectionPoints (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long) As String

This function gives the selection points that is to say the outline points belonging to the current scene selected objects. This selection points list corresponds to the current view mode (top view, elevation or perspective).

The list is returned as a characters string.

The selection points coordinates are given in pixels in the reference corresponding to the image which resolution is *ImageResX* x *ImageResY* and that represents the current view.

The points coordinates are separated with a coma (",") and the points with a semi-column (";") as illustrated in the example below:

« x1,y1,z1;x2,y2,z2;...;xn,yn,zn;... », where « xn », « yn » and « zn » are the coordinates of the nth selection point.

Return value: the function returns the characters string listing the selection points.

FileGetSelectionOutlinePoints (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelXRot* As Long, *PixelYRot* As Long, *Angle* As Double) As String

This function gives the polylines composing the **outline** of the selected objects of the current scene.

This polylines list corresponds to the current view mode (top view, elevation or perspective).

The list is returned as a characters string.

The polylines points coordinates are given in pixels in the reference corresponding to the image which resolution is *ImageResX* x *ImageResY* and that represents the current view.

The points coordinates are separated with a coma (","), the points with a semi-column (";") and the polylines with a column (":") as illustrated in the example below:

« p1x1,p1y1,p1z1;p1x2,p1y2,p1z2;...;p1xn,p1yn,p1zn;...;p2x1,p2y1,p2z1;p2x2,p2y2,p2z2;...;p2xn,p2yn,p2zn;... », where « pmxn », « pmyn » and « pmzn » are the coordinates of the nth point of the mth polyline.

The *PixelXRot* and *PixelYRot* parameters represent the position in image coordinates (pixels) of the pivot point of a possible rotation (for example the mouse cursor position) and the *Angle* parameter specifies the rotation angle of the outlines in degrees.

Return value: the function returns the characters string listing the selection outline polylines points.

FileGetCatalogOutlinePoints (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *CatalogFileName* As String, *ArticleRef* As String, *HandingType* As Long, *Width* As Long, *Depth* As Long, *Height* As Long, *Angle* As Double, *Opened* As Long) As String

This function gives the polylines composing the **outline** of the article which reference is *ArticleRef* and having a *HandingType* handing in the *CatalogFileName* catalogue (short file name without extension).

The **outline** corresponds to the current view of the current scene (top view, elevation or perspective).

The list is returned as a characters string.

The polylines points coordinates are given in pixels in the reference corresponding to the image which resolution is *ImageResX* x *ImageResY* and that represents the current view.

The points coordinates are separated with a coma (","), the points with a semi-column (";") and the polylines with a column (":") as illustrated in the example below:

« p1x1,p1y1,p1z1;p1x2,p1y2,p1z2;...;p1xn,p1yn,p1zn;...;p2x1,p2y1,p2z1;p2x2,p2y2,p2z2;...;p2xn,p2yn,p2zn;...», where « pmxn », « pmy n » and « pmzn » are the coordinates of the nth point of the mth polyline.

The dimensions applied to the article which can be different than the default dimensions setup in the catalogue are specified in the *Width*, *Depth* and *Height* parameters.

If the *Opened* parameter equals 0, the article is shown closed; else it's shown opened as far as an opened version of the article has been designed.

The *XRes* and *YRes* parameters specify the resolution of the exported image in pixels.

Normally, the **article outline image** should be generated with the same resolution than the image representing the current view.

The *Angle* parameter specifies in degrees a possible rotation angle applied to the article during the « drag & drop » operation.

Return value: the function returns the characters string listing the article outline polylines points.

EditIsUndoPossible (*SessionId* As Long) As Boolean

Return value: the function returns 1 if the **EditUndo** function can « undo » the last operation that has been carried out; else it returns 0.

EditIsRedoPossible (*SessionId* As Long) As Boolean

Return value: the function returns 1 if the last operation that has been carried out is an « Undo » command and consequently if the **EditUndo** function will restore the last but one operation; else it returns 0.

EditUndo (*SessionId* As Long) As Boolean

This function « undoes » the last operation that has been carried out.

To call this function twice is equivalent to do nothing; there is only one undo level.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditDelete (*SessionId* As Long) As Boolean

This function deletes all the selected objects belonging to the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditPlaceObject (*SessionId* As Long, *CatalogFileName* As String, *ArticleRef* As String, *HandingType* As Long, *Width* As Long, *Depth* As Long, *Height* As Long, *X* As Long, *Y* As Long, *Altitude* As Long, *AltitudeType* as Long, *OXYAngle* As Long, *Opened* As Long, *Percussion* As Long, *Pool* As Long) As Long

This function places in the current scene the article coming from the *CatalogFileName* catalogue (short file name without extension) having the *ArticleRef* reference and the *HandingType* handing. The dimensions to be applied to the article are indicated in the *Width*, *Depth* and *Height* parameters. The coordinates of its reference point in the horizontal plane are *X*, *Y* and its angle in the scene relatively to the horizontal axis is *OXYAngle* degrees.

The altitude of the article, that is to say its position relatively to the vertical axis is defined by the *Altitude* parameter which specifies a value and by the *AltitudeType* parameter which indicates if *Altitude* is the article bottom position (ALTITUDETYPE_ON = 0) or the article top position (ALTITUDETYPE_UNDER = 1).

If the *Opened* parameter equals 1, the object is placed in its opened state.

If the *Percussion* parameter equals 1, the function eventually modifies the position of the object depending on its percussions with other objects in the current scene.

If the *Pool* parameter equals 1, the placed object is not visible in the graphical views (top view, elevation and perspective). At the contrary, it is present in the pool, that is to say in the list of the objects being placed while in pricing view mode (with no information about its position) and with the possibility to be located manually in the scene in a second stage thanks to the "Place | Pool" command.

If you use this function with the *Pool* parameter setup to 1, the *X*, *Y* and *Altitude* parameters are ignored.

Return value: the function returns the identifier of the placed object if everything went well; else it returns -1.

EditPlaceObjectPixel (*SessionId* As Long, *CatalogFileName* As String, *ArticleRef* As String, *HandingType* As Long, *Width* As Long, *Depth* As Long, *Height* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX* As Long, *PixelY* As Long, *Altitude* As Long, *AltitudeType* as Long, *OXYAngle* As Long, *Opened* As Long, *Percussion* As Long, *Pool* As Long) As Long

This function has the same purpose than the function described previously (**EditPlaceObject**); the difference is that the position of the object reference point (*PixelX* and *PixelY*) is given in pixels in the reference corresponding to the image whose resolution in pixels is *ImageResX* x *ImageResY* and that represents the current view.

If the view mode is VIEWMODE_2D, *PixelX* and *PixelY* represent the position of the object reference point in the horizontal plane.

If the view mode is VIEWMODE_ELEVATION or VIEWMODE_REALELEVATION, *PixelX* and *PixelY* represent the position of the object reference point in the current elevation plane.

Concerning the other parameters, please refer to the function described previously (**EditPlaceObject**).

Return value: the function returns the identifier of the placed object if everything went well; else it returns -1.

EditPlaceLinearObject (*SessionId* As Long, *CatalogFileName* As String, *ArticleRef* As String, *FinishesList* As String, *Place* as long) As Boolean

This function starts removing the linear objects of the same type (plinth, worktop, worktop edge, light pelmet, cornice) than the *ArticleRef* article which is already in the scene (if *ArticleRef* is empty and *Place* equals 0, all the linear objects in the scene are deleted any type they are).

Then, if the *Place* parameter equals 1, the function creates one or several shapes depending on the objects present in the current scene and on the type of the linear object to be placed (plinth, worktop, worktop edge, light pelmet, cornice).

The linear article having the *ArticleRef* reference in the *CatalogFileName* catalogue (short file name without extension with or without extension) is then placed along the generated shapes and in respect with the finishes mentioned in the *FinishesList* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditReplaceSelection (*SessionId* As Long, *CatalogFileName* As String, *ArticleRef* As String, *HandingType* As Long, *Width* As Long, *Depth* As Long, *Height* As Long, *Opened* As Long) As Boolean

This function replaces all the selected objects in the current scene with article having the *ArticleRef* reference and the *HandingType* handing in the *CatalogFileName* catalogue (short file name without extension). The replacing article is placed with the *Width*, *Depth* and *Height* dimensions and at the exact location and angle than the selected objects.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditSelectAll (*SessionId* As Long) As Boolean

This function selects all the objects belonging to the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditReverseSelection (*SessionId* As Long) As Boolean

This function inverts the selection state of all the objects belonging to the current scene. Consequently, the selected objects become unselected and reciprocally.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditSelectFromRectangle (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX1* As Long, *PixelY1* As Long, *PixelX2* As Long, *PixelY2* As Long, *MultipleSelection* As Long) As Boolean

This function selects all the objects of the current scene which are completely bounded by the rectangle that is defined by the two following points: (*PixelX1*, *PixelY1*) and (*PixelX2*, *PixelY2*). These coordinates are given in pixels in the reference corresponding to the image whose resolution in pixels is *ImageResX* x *ImageResY* and which represents the current view.

Return value: the function returns 1 if no problem occurred; else it returns 0.

EditGetSceneX (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX* As Long) As Long

Return value: the function returns the coordinate in scene measurement units which corresponds to the *PixelX* horizontal pixel coordinate considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

EditGetSceneY (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelY* As Long) As Long

Return value: the function returns the coordinate in scene measurement units which corresponds to the *PixelY* vertical pixel coordinate considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

EditGetPixelX (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *SceneX* As Long) As Long

Return value: the function returns the coordinate in pixels which corresponds to the *SceneX* horizontal scene measurement units coordinate considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

EditGetPixelY (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *SceneY* As Long) As Long

Return value: the function returns the coordinate in pixels which corresponds to the *SceneY* vertical scene measurement units coordinate considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

EditGetMagneticPoint (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX* As Long , *PixelY* As Long, *MagneticMargin* As Long) As String

This function gets the closest magnetic point in the current view to the point which is defined by the *PixelX* and *PixelY* coordinates considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

If no magnetic point is found in a square of side *MagneticMargin* pixels around the (*PixelX* , *PixelY*) point or if *MagneticMargin* =0 (disabled magnetism), **x2** = *PixelX*, **y2** = *PixelY*, **x1** and **y1** represent respectively the **x2** and **y2** values converted in scene measurement units and **x3**, **y3** and **z3** represent the coordinates in the scene of the (*PixelX* , *PixelY*) point.

Return value: the function returns a characters string having the following format: « **x1;y1;x2;y2;x3;y3;z3** »

where,

x1 = x coordinate in scene measurement units of the magnetic point projected into the plane corresponding to the current view (top view, elevation plane),

y1 = y coordinate in scene measurement units of the magnetic point projected into the plane corresponding to the current view (top view, elevation plane),

x2 = x coordinate in pixels of the magnetic point projected into the plane corresponding to the current view (top view, elevation plane) considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

y2 = y coordinate in pixels of the magnetic point projected into the plane corresponding to the current view (top view, elevation plane) considering the image representing the current view has a resolution in pixels of *ImageResX* x *ImageResY*.

x3 = x coordinate in scene measurement units of the magnetic point,

y3 = y coordinate in scene measurement units of the magnetic point,

z3 = z coordinate in scene measurement units of the magnetic point.

SelectionGetObjectsNb (*SessionId* As Long) As Long

Return value: the function returns the number of selected objects in the current scene.

SelectionGetObjectId (*SessionId* As Long, *SelectedObjectRank* As Long) As Long

Return value: the function returns the identifier of the selected object which rank in the current scene is *SelectedObjectRank*.

With this function it is possible to build a list of the selected objects looping on the *SelectedObjectRank* parameter from 1 to the value returned by the **SelectionGetNb** function described above.

SelectionMove (*SessionId* As Long, *SceneXOrig* As Long, *SceneYOrig* As Long, *SceneZOrig* As Long, *OXYAngleOrig* As Double, *OYZAngleOrig* As Double, *OZXAngleOrig* As Double, *SceneXExt* As Long, *SceneYExt* As Long, *SceneZExt* As Long, *OXYAngleExt* As Double, *OYZAngleExt* As Double, *OZXAngleExt* As Double, *Distance* As Long, *Duplicate* As Long, *Percussion* As Long) As Boolean

This function translates and/or rotates the selected objects of the current scene.

The translation axis is defined by the vector **in scene coordinates** having the (*SceneXOrig*, *SceneYOrig*, *SceneZOrig*) origin and the (*SceneXExt*, *SceneYExt*, *SceneZExt*) extremity.

The value (length) of the translation is defined by the *Distance* parameter given in the scene measurement unit. If the *Distance* parameter equals 0, then the value (length) of the translation is defined by the distance between the two points which coordinates are passed as parameters.

The rotations are defined by the differences between the *OXYAngleExt* and *OXYAngleOrig* angles, the *OYZAngleExt* and *OYZAngleOrig* angles and at last the *OZXAngleExt* et *OZXAngleOrig* angles.

If the *Duplicate* parameter equals 1, the function doesn't delete the selected objects located at their original position. So, the selected objects are duplicated.

If the *Percussion* parameter equals 1, the function eventually modifies the position of the object depending on its percussions with other objects in the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SelectionPixelMove (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelXOrig* As Long, *PixelYOrig* As Long, *OXYAngleOrig* As Double, *PixelXExt* As Long, *PixelYExt* As Long, *OXYAngleExt* As Double, *Distance* As Long, *Duplicate* As Long, *Percussion* As Long) As Boolean

This function translates and/or rotates the selected objects of the current scene.

The translation axis is defined by the vector **in pixel coordinates** having the (*PixelXOrig*, *PixelYOrig*) origin and the (*PixelXExt*, *PixelYExt*) extremity.

The value (length) of the translation is defined by the *Distance* parameter given in the scene measurement unit. If the *Distance* parameter equals 0, then the value (length) of the translation is defined by the distance between the two points which coordinates are passed as parameters.

The rotation in the horizontal plane is defined by the difference between the *OXYAngleExt* and *OXYAngleOrig* angles.

If the *Duplicate* parameter equals 1, the function doesn't delete the selected objects located at their original position. So, the selected objects are duplicated.

If the *Percussion* parameter equals 1, the function eventually modifies the position of the object depending on its percussions with other objects in the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SelectionRotate (*SessionId* As Long, *OXYAngle* As Double, *OYZAngle* As Double, *OZXAngle* As Double) As Boolean

This function rotates the selected objects of the current scene in one or several of the 3 rotation planes.

All the selected objects are rotated around the same pivot point which is located at the centre of the parallelepiped bounding all the selected objects.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SelectionDimension (*SessionId* As Long) As Boolean

This function launch the automatic generation of dimensions for the selected objects of the current scene. If there is no selected object in the scene, dimensions are generated for all the objects of the scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SelectionOpen (*SessionId* As Long) As Boolean

This function « opens » all the selected objects of the scene as far as the objects have been designed to be opened.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SelectionClose (*SessionId* As Long) As Boolean

This function « closes » all the selected objects of the scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectGetKeywordInfo (*SessionId* As Long, *ObjectId* As Long, *Keyword* As String) As String

This function gets the information specified in the *KeyWord* characters string concerning the object in the scene whose identifier is *ObjectId*.

If the *ObjectId* parameter equals -2, the function applies to the active object of the current scene.

The values that *KeyWord* can take are described in the « Creation of customized WORD documents » document that can be downloaded from this link: www.kitchendraw.com/FTP/worddoceng.rtf.

Any « Object » key word can be used.

Return value: the function returns the requested information as a characters string. It returns an empty characters string if *ObjectId* equals -1 or doesn't correspond to any object in the scene or if the *KeyWord* parameter is unknown.

ObjectSetKeywordInfo (*SessionId* As Long, *ObjectId* As Long, *Value* As String, *KeyWord* As String) As Boolean

This function writes the *Value* characters string in the data filed specified in the *KeyWord* parameter of the object whose identifier is *ObjectId*

If the *ObjectId* parameter equals -2, the function applies to the active object of the current scene.

The values that *KeyWord* can take are described in the « Creation of customized WORD documents » document that can be downloaded from this link: www.kitchendraw.com/FTP/worddoceng.rtf.

Any « Object » key word can be used.

Return value: the function returns 1 if the write operation went well. Else, it returns 0 (for example if *ObjectId* equals -1 or doesn't correspond to any object in the scene or if the *KeyWord* parameter is unknown).

ObjectGetInfo (*SessionId* As Long, *ObjectId* As Long, *InfoType* As Long) As String

This function gets the information specified in the *InfoType* numerical parameter concerning the object in the scene whose identifier is *ObjectId*.

If the *ObjectId* parameter equals -2, the function applies to the active object of the current scene.

The values that can take the *InfoType* parameter are listed in the following table:

<i>InfoType</i> values	Description
OBJINFO_CATALOGFILENAME = 0	Catalogue file name of the catalogue the object comes from (8 characters maxi)
OBJINFO_CATALOGNAME = 1	Catalogue name of the catalogue the object comes from (30 characters maxi)
OBJINFO_CATALOGCODE = 2	Catalogue code of the catalogue the object comes from (8 characters maxi)
OBJINFO_SECTIONNAME = 3	Section name of the section the object comes from (30 characters maxi)
OBJINFO_BLOCKCODE = 4	Block code of the block the object comes from (10 characters maxi)
OBJINFO_NAME = 5	Name of the object (30 characters maxi)
OBJINFO_KEYREF = 7	Key reference of the object (30 characters maxi) coming from the "Articles" table of the catalogue and that is unique
OBJINFO_REF = 8	Reference of the object (30 characters maxi) coming from the "References" table of the catalogue. If the cell corresponding to this article and to its model or finish in the "References" table is empty, the OBJINFO_KEYREF is returned
OBJINFO_USERREF = 9	« User » reference of the object. This reference is eventually attached to the object in KitchenDraw by the user himself through the "Object Attributs" dialog box.
OBJINFO_CODE = 6	Code of the object not to be mistaken for the reference. This code is a kind of secondary reference setup in case of double referencing
OBJINFO_HANDING = 10	Handing (or hinges position) of the object as a characters string (« L » or « R » in English but depending on the current language)
OBJINFO_HANDINGTYPE = 11	Handing (or hinges position) of the object as a long integer: HANDINGTYPE_NONE = 0

	HANDINGTYPE_LEFT = 1 HANDINGTYPE_RIGHT = 2
OBJINFO_DIMX = 12	Width of the object in the scene measurement units
OBJINFO_DIMY = 13	Depth of the object in the scene measurement units
OBJINFO_DIMZ = 14	Height of the object in the scene measurement units
OBJINFO_ISDIMXVAR = 15	Width « variability » flag of the object. This flag indicates if the object width can be changed by the user in KitchenDraw or not. It can be used to grey or not the « Width » editable text area corresponding to the active object of the current scene
OBJINFO_ISDIMYVAR = 16	Depth « variability » flag of the object
OBJINFO_ISDIMZVAR = 17	Height « variability » flag of the object
OBJINFO_ORDERDIMX = 18	Order width of the object (width in the supplier order). This value is always equal to 0 if the LC0 parameter is present in the block script; it's equal to the corresponding "catalog dimension" if the LCL, LCP or LCH parameter is present in the block script; it's equal to the corresponding "scene dimension" if the LCLV, LCPV or LCHV parameter is present in the block script.
OBJINFO_ORDERDIMY = 19	Order depth of the object (depth in the supplier order). This value is always equal to 0 if the PC0 parameter is present in the block script; it's equal to the corresponding "catalog dimension" if the PCL, PCP or PCH parameter is present in the block script; it's equal to the corresponding "scene dimension" if the PCLV, PCPV or PCHV parameter is present in the block script.
OBJINFO_ORDERDIMZ = 20	Order height of the object (height in the supplier order). This value is always equal to 0 if the HC0 parameter is present in the block script; it's equal to the corresponding "catalogue dimension" if the HCL, HCP or HCH parameter is present in the block script; it's equal to the corresponding "scene dimension" if the HCLV, HCPV or HCHV parameter is present in the block script.
OBJINFO_POSX = 21	Position on the X axis (width of the screen) of the object in scene measurement units. A value of 0 means the position is located in the centre of the dotted green rectangle representing the scene dimensions.
OBJINFO_POSY = 22	Position on the Y axis (height of the screen) of the object in scene measurement units. A value of 0 means the position is located in the centre of the dotted green rectangle representing the scene dimensions.
OBJINFO_POSZ = 23	Position on the Z axis (altitude) of the object in scene measurement units. A value of 0 means the position is located in bottom face of the dotted green parallelepiped representing the scene dimensions.
OBJINFO_ON_OR_UNDER = 55	Reference altitude of the object: ALTITUDETYPE_ON = 0 ALTITUDETYPE_UNDER = 1
OBJINFO_ANGLEXY = 24	Angle in the horizontal plane (0, X, Y) of the object in degrees
OBJINFO_ANGLEYZ = 25	Angle in the (0, Y, Z) plane of the object in degrees
OBJINFO_ANGLEZX = 26	Angle in the (0, Z, X) plane of the object in degrees
OBJINFO_DISTLEFTWALL = 27	Distance between the left side of the object and the left wall corner
OBJINFO_DISTRIGHTWALL = 28	Distance between the right side of the object and the right wall corner
OBJINFO_UNITTEXT = 29	Text representing the measurement unit of the object (that is defined in the catalogue and that can be different from the scene unit of measurement)
OBJINFO_UNITVALUE = 30	Measurement unit value of the object in millimetres

OBJINFO_ID = 31	Constant and unique numerical identifier of the object	
OBJINFO_TYPE = 32	Object type which possible values are listed below:	
	OBJINFO_TYPE value	Description
	OBJTYPE_STANDARD = 0	Standard object from a catalogue
	OBJTYPE_DOOR = 1	Door
	OBJTYPE_WINDOW = 2	Window
	OBJTYPE_RECESS = 3	Niche, recess
	OBJTYPE_PLAN = 4	« Plane object » placed inside a shape drawn freely by the user
	OBJTYPE_PLANARTICLE = 5	« Plane object » placed inside a shape that is predefined in the catalogue but that can be « cut » by the user
	OBJTYPE_LINEAR = 6	« Linear object » placed along a shape (plinth, cornice, etc.)
	OBJTYPE_WALL = 7	Wall
	OBJTYPE_TEXT = 8	Text
	OBJTYPE_LINEARDIMENSION = 9	Simple or multiple linear dimension, placed manually or automatically
	OBJTYPE_ARROW = 10	Arrow
	OBJTYPE_PLANNINGAREA = 11	Zone
	OBJTYPE_ELEVATIONSYMBOL = 12	Elevation symbole
	OBJTYPE_ANGULARDIMENSION = 13	Angular dimension
	OBJTYPE_DXF3D = 14	Imported DXF3D file
	OBJTYPE_WMF = 15	Imported WMF file
	OBJTYPE_GROUP = 16	Object that is a header of a set of grouped objects
	OBJTYPE_OPEN = 17	Object which doesn't belong to a catalogue and that is placed thanks to a « Place Open article » command
	OBJTYPE_METAFILE = 18	Object that is created from an « Edit paste » command
	OBJTYPE_DXF2D = 19	Imported DXF2D file
	OBJTYPE_BMP = 20	Imported BMP file
	OBJTYPE_ENVELOPE = 21	Object that is a header of a catalogue block set
	OBJTYPE_TECHNICALSYMBOL = 22	Technical symbole placed in elevation
	OBJTYPE_LIGHTSOURCE = 23	Light source
	OBJTYPE_TILE = 24	Floor or wall tile
OBJINFO_COMPONENTTYPE = 33	Component type which possible values are listed below:	
	OBJINFO_COMPTYPE value	Description
	COMPTYPE_OPTION = 0	« Normal » component that is to say, option or accessory of the parent object or member of a block subset associated

		to the parent object for placing convenience reasons
	COMPTYPE_INTERMEDIATE = 1	Intermediate component for cutting lists (see the MobiScript 2 documentation for more information)
	COMPTYPE_CARCASEPANEL = 2	Carcase panel for cutting lists
	COMPTYPE_FRONTPANEL = 3	Front panel for cutting lists
	COMPTYPE_PROFILE = 4	Profile or linear part for Bill Of Material (edge, etc.)
	COMPTYPE_PART = 5	Part for Bill Of Material (legs, hinges, etc.)
	COMPTYPE_LIQUIDPRODUCT = 6	Liquid product for Bill Of Material (glue, varnish, etc.)
	COMPTYPE_LABOR = 7	Manpower for Bill Of Material
	COMPTYPE_SUBSET = 8	Subset for cutting lists (see the MobiScript 2 documentation for more information)
	COMPTYPE_NOTACOMPONENT = -1	The object is not a component (object of highest level)
OBJINFO_NUMBER = 34	Number of the object as it may appear in the pricing table or in the top or elevation views when the « sequential number » mark type is selected in the « Scene Mark » dialog box	
OBJINFO_QUANTITY = 35	Quantity of the object. This information can be different than 1 only for linear objects, plane objects or objects that displays a dialog box asking for a quantity when they are placed. ATTENTION: Apart from the objects mentioned above, several identical objects present in the same scene (even if they don't have any graphics) will not be grouped as one object having a quantity higher than 1	
OBJINFO_LENGTH = 36	Sum of segments length of a linear object placed along one or several shapes. This information can be used to calculate the price of a linear object having a "length dependent" price type.	
OBJINFO_AREA = 37	Net surface for a plane object	
OBJINFO_LEFTCUTANGLE = 38	Left cut out angle for a plane object	
OBJINFO_RIGHTCUTANGLE = 39	Right cut out angle for a plane object	
OBJINFO_LEFTCUTLENGTH = 40	Left front cut out length for a plane object	
OBJINFO_RIGHTCUTLENGTH = 41	Right front cut out length for a plane object	
OBJINFO_LAYER = 42	Layer where the object is located This is a read only information	
OBJINFO_PRICE = 43	Price of the object. Returns an empty characters string if the object is not « billed » or it's not a valid object. At the contrary, the price is returned even if the object being a component is not « placed » in the scene (unchecked in the "Object Components" dialog box)	
OBJINFO_GROUPNAME = 44	Name of the group the object belongs to	
OBJINFO_GROUPNUMBER = 45	Number of the group the object belongs to	
OBJINFO_PRIORITY = 46		
OBJINFO_PAGE = 47	Page number in the paper catalogue where the object is	

	located
OBJINFO_VATTYPE = 48	Tax rate type that is applicable to the object (it comes from the origin catalogue)
OBJINFO_DESCRIPTION = 49	Long description of the object as it appears in the estimate
OBJINFO_COMMENT_CUSTOMER = 50	Comment intended to the end customer (appears in the estimate)
OBJINFO_COMMENT_SUPPLIER = 51	Comment intended to the supplier (appears in the supplier order).
OBJINFO_COMMENT_FITTER = 52	Comment intended to the fitter (appears in the fitting note or any document intended to the fitter)
OBJINFO_XML = 53	XML data area used only by KitchenDraw « Plug-In » functions and allowing them to store and retrieve specific data into the objects
OBJINFO_VALUE = 54	
OBJINFO_PRICETYPE = 56	Price type
OBJINFO_PRICEPERQUANTITY = 57	
OBJINFO_ISVALID = 58	« Valid » flag indicates if the object is valid. An object is declared as being invalid if one of its dimensions is not allowed in the catalogue or if the object doesn't exist in the front model or the finishes that have been applied to it (empty price in the catalogue "Prices" table) Returns the « 1 » characters string if the object is "valid"; else it returns « 0 » This is a read only information
OBJINFO_ISPLACED = 59	« Placed » flag (for a component) indicates if the component is « placed » that is to say is checked in the « Object Components » dialog box. Returns the « 1 » characters string if the object is "placed"; else it returns « 0 »
OBJINFO_ISSELECTED = 60	« Selected » flag (object having black squares in place of outline points or not). Returns the « 1 » characters string if the object is selected; else it returns « 0 »
OBJINFO_ISACTIVE = 61	« Active » flag (object having blinking outline points or not). Returns the « 1 » characters string if the object is the active object of the current scene ; else it returns « 0 »
OBJINFO_ISOPEN = 62	« Open » flag returns the « 1 » characters string if the object is opened or the « 0 » characters string if it's closed
OBJINFO_SUPPLIERID = 63	Supplier identifier of the object origin catalogue. This is a read only information
OBJINFO_MODELNAME = 64	Front model name which is applied to the object if the object is model dependant
OBJINFO_MODELCODE = 65	Front model code which is applied to the object if the object is model dependant
OBJINFO_MODELPRICECOLUMN = 66	Front model price column which is applied to the object if the object is model dependant
OBJINFO_MODELFINISHTYPE... OBJINFO_MODELFINISHTYPE+15	
OBJINFO_MODELFINISHNAME... OBJINFO_MODELFINISHNAME+15	
OBJINFO_MODELFINISHCODE... OBJINFO_MODELFINISHCODE+15	
OBJINFO_FAMILYFINISHTYPE... OBJINFO_FAMILYFINISHTYPE+15	
OBJINFO_FAMILYFINISHNAME... OBJINFO_FAMILYFINISHNAME+15	
OBJINFO_FAMILYFINISHCODE... OBJINFO_FAMILYFINISHCODE+15	
OBJINFO_TEXTURE... OBJINFO_TEXTURE+8	

OBJINFO_COMPONENTLEVEL = 171	Position of the object in the components hierarchy. A value of 1 indicates that the object is not a component but an object of the highest level possible.
OBJINFO_BASEPRICE = 172	Base purchase price of the object coming from the catalogue.
OBJINFO_GROSSELLINGPRICE = 173	Gross selling price of the object including taxes.

Return value: the function returns the requested information as a characters string or it returns an empty characters string if *ObjectId* equals -1 or if *ObjectId* doesn't correspond to any object in the current scene or if *InfoType* is unknown.

ObjectSetInfo (*SessionId* As Long, *ObjectId* As Long, *Value* As String, *InfoType* As Long) As Boolean

This function writes the *Value* characters string in the data field which is specified by the *InfoType* parameter in the object which identifier is specified by the *ObjectId* parameter.

If *ObjectId* equals -2, the function applies to the active object of the current scene.

Please, refer to the **ObjectGetInfo** function described above to get the *InfoType* values list.

Return value: the function returns 1 if the write operation went well. Else, it returns 0 (for example if *ObjectId* equals -1 or doesn't correspond to any object in the scene or if the *InfoType* parameter is unknown).

ObjectGetScript (*SessionId* As Long, *ObjectId* As Long) As String

If *ObjectId* equals -2, the function applies to the active object of the current scene.

Return value: the function returns the script corresponding to the object that is identified by the *ObjectId* parameter.

ObjectSetScript (*SessionId* As Long, *ObjectId* As Long, *Script* As String) As Boolean

This function replaces and recompiles the script of the object whose identifier is *ObjectId* in the current scene. This often results in a change in the object's top view, 3D model or description, etc.

If *ObjectId* equals -2, the function applies to the active object of the current scene.

This function is used while developing wizards in the validation part of them.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectGetCustomInfo (*SessionId* As Long, *ObjectId* As Long, *InfoKey* As String) As String

This function reads the characters string identified by the *InfoKey* parameter that is contained in the data area dedicated to the application extensions (wizards and « plug-in » functions) and which corresponds to the object identified by the *ObjectId* parameter.

This characters string must have been placed in this area previously using the **SceneSetCustomInfo** function as well as the same value of the *InfoKey* parameter.

The characters strings allow a wizard or a « plug-in » function storing in the scene some custom configuration variables. Thanks to the *InfoKey* parameter each wizard or « plug-in » function can isolate its own variables from the ones of the others.

These specific characters strings appear in the XML file that is generated by the "File | Export | Management data (.XML)" command; each one under a tag equal to the value of the corresponding *InfoKey* parameter.

If *ObjectId* equals -2, the function applies to the active object of the current scene.

Return value: the function returns the requested information as a characters string.

ObjectSetCustomInfo (*SessionId* As Long, *ObjectId* As Long, *Value* As String, *InfoKey* As String) As Boolean

This function writes the *Value* characters string into the data area dedicated to the application extensions (wizards and « plug-in » functions) and which corresponds to the object identified by the *ObjectId* parameter.

This characters string is identified by the *InfoKey* parameter.

If *ObjectId* equals -2, the function applies to the active object of the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectGetParentId (*SessionId* As Long, *ObjectId* As Long) As Long

Return value: the function returns the identifier of the parent object corresponding to the object whose identifier is *ObjectId* if this object is a component; else it returns -1.
If *ObjectId* equals -2, the function applies to the active object of the current scene.

ObjectGetChildrenNb (*SessionId* As Long, *ObjectId* As Long, *AllComponents* As Long) As Long

Return value: the function returns the number of components held by the object whose identifier is *ObjectId*. The components that are taken into account are components of any type being "placed" or not, being valid or not.

If *AllComponents* equals 1, this number represents all the components of the object whatever their position can be in the components hierarchy. If *AllComponents* equals 0, this number represents only the components pertaining to the lower level in the components hierarchy.

If *ObjectId* equals -2, the function applies to the active object of the current scene.

ObjectGetChildId (*SessionId* As Long, *ObjectId* As Long, *ChildRank* As Long, *AllComponents* As Long) As Long

Return value: the function returns the identifier of the component whose rank is *ChildRank* (starting from 1) being held by the object whose identifier is *ObjectId*.

Be careful : the *ChildRank* parameter will point to a different component according to the value of the *AllComponents* parameter. Please, refer to the description of the **ObjectGetChildrenNb** function for more information about the exact meaning of this parameter.

The number of components held by the *ObjectId* object is returned by the **ObjectGetChildrenNb** function which is described earlier.

Normally, the *AllComponents* parameter mentioned in the function should have the same value than the one indicated in the **ObjectGetChildrenNb** function.

If *ObjectId* equals -2, the function applies to the active object of the current scene.

ObjectSelect (*SessionId* As Long, *ObjectId* As Long, *MultipleSelection* As Long) As Long

This function selects the object whose identifier is *ObjectId*.

If the *MultipleSelection* parameter equals 1, the objects that were already selected will remain selected.

Return value: the function returns the number of selected objects after the function is performed.

ObjectGetFinishesConfig (*SessionId* As Long, *ObjectId* As Long) As String

Return value: the function returns a characters string representing the current front model and finishes configuration corresponding to the object whose identifier is *ObjectId*.

The characters string has the following format:

«-1,10005,10006,10007,20022,20023;;1,2,1,4,9,7».

At the left hand side of « ; ; », we can find the finishes types separated with a coma. It's possible to get the name of the finishes types corresponding to these values thanks to the **CatalogGetFinishTypeName** function of the **Appli** class.

The -1 code represents the front model; the 1XXXX codes represent the model finishes types and the 2XXXX codes represent the family finishes types.

At the right hand side of « ; ; », we can find the ranks (numbered from 0 and separated with a coma) of the finishes corresponding respectively to the finishes types listed previously. Each value represents the rank of the selected finish in the list of the available finishes for the corresponding finish type.

ObjectSetFinishesConfig (*SessionId* As Long, *ObjectId* As Long, *FinishesList* As String) As Boolean

This function modifies the current front model and finishes configuration corresponding to the object whose identifier is *ObjectId*.

This is done thanks to the *FinishesList* parameter that represents the characters string listing in the right order the front model followed eventually by the finishes ranks to be applied to the object. The finishes ranks are numbered starting from 0 and are separated with a coma.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectModifyFinishesConfig (*SessionId* As Long, *ObjectId* As Long, *ModifiedLine* As Long, *NewFinish* As Long, *Modify* as Long) As String

Return value: this function returns a characters string representing the finishes configuration to be presented following the configuration modification of the object passed in the *ObjectId* parameter by a finish change (*NewFinish*) carried out at the level of the finish type of rank *ModifiedLine*.

The characters string has the following format:

«-1,10005,10006,10007,20022,20023;;1,2,1,4,9,7».

At the left hand side of « ; ; », we can find the finishes types separated with a coma. It's possible to get the name of the finishes types corresponding to these values thanks to the **CatalogGetFinishTypeName** function of the **Appli** class.

The -1 code represents the front model; the 1XXXX codes represent the model finishes types and the 2XXXX codes represent the family finishes types.

At the right hand side of « ; ; », we can find the ranks (numbered from 0 and separated with a coma) of the finishes corresponding respectively to the finishes types listed previously. Each value represents the rank of the selected finish in the list of the available finishes for the corresponding finish type.

If the *Modify* parameter equals 1, the configuration of the object finishes will be actually modified.

ObjectGetFinishCodeAndName (*SessionId* As Long, *ObjectId* As Long, *FinishTypeRank* As Long, *FinishRank* As Long) As String

Return value: this function returns a characters string containing the code and the name of the *FinishRank* finish rank (starting from 0) taken in the finish type of rank *FinishTypeRank*.

The code and the name are separated with the « ; ; » characters string.

ObjectExportImage (*SessionId* As Long, *ObjectId* As Long, *Opened* As Long, *ViewMode* As Long, *ImageFileName* As String, *XRes* As Long, *YRes* As Long, *BackGroundColor* As String, *Transparent* As Boolean, *JPEGQuality* As Long, *AntiAliasing* As Long) As Boolean

This function exports an image corresponding to the object whose identifier is *ObjectId*. The generated file can be a .JPG, .BMP, .GIF, .PNG or .TIF file whose name (including the full access path) is specified in the *ImageFileName* parameter.

The *XRes* and *YRes* parameters specify the resolution in pixels of the exported image.

If the *Opened* parameter is equal to 0, the object is shown closed; else it's shown opened.

The *ViewMode* parameter indicates what kind of view type must be generated.

<i>ViewMode</i> value	Description
VIEWMODE_2D	Top view
VIEWMODE_ELEVATION	Wireframe elevation
VIEWMODE_REALELEVATION	Realistic elevation
VIEWMODE_3D	Wireframe perspective
VIEWMODE_REAL3D	Realistic perspective
VIEWMODE_PHOTO3D = 5	Photo-realistic perspective

The *BackGroundColor* parameter defines the colour of the image background. It's a characters string representing the decimal values (between 0 and 255) of the red component, then the green component and then the bleu component, separated with a coma like in the following example: « 255,128,0 ».

If the *BackGroundColor* parameter is empty, the background colour will be white.

If the *Transparent* parameter is set as True, the background colour will be considered as transparent (for the .GIF, .PNG and .TIF formats only).

For the .JPG format only, it is possible to control the quality level of the image with the *JPEGQuality* parameter (its value must be between 0 and 100, with a recommended value of 75).

The *AntiAliasing* parameter indicates the anti-aliasing level used to reduce the stairs effect on the oblique lines. For speed reasons, it's not advised to use an anti-aliasing level higher than 3.

<i>AntiAliasing</i> value	Description
1	No anti-aliasing
2	The image is first generated at a resolution of 2xXRes and 2xYRes and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.
3	The image is first generated at a resolution of 3xXRes and 3xYRes and then the final image is obtained calculating each pixel colour as the average colour of 4 pixels.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectGetShape(*SessionId* As Long, *ObjectId* As Long) As String

Return value: the function returns the characters string which lists the points belonging to the shape corresponding to the *ObjectId* object.

The coordinates are given in decimal values and are separated with a coma. The points are separated with a semi column like in the following example:

« x1,y1,z1,t1;x2,y2,z2,t2;...;xn,yn,zn,tn;... », where « xn », « yn » and « zn » are the scene coordinates of the nth point of the shape and « tn » its property value.

The different shape point properties are listed in the following table:

Shape point properties	Description
SHAPEPOINTPROP_SELECTED = 1	Selection flag of the point
SHAPEPOINTPROP_ARC = 2	Circle arc flag : indicates the point defines a circle arc along with the previous point of the shape and the following point.

The value mentioned after the coordinates of a shape point is the sum of the properties values corresponding to this point. This way, a point that is selected and that is at the same time a circle arc point will have a property value of 3 (1 + 2).

ObjectSetShape(*SessionId* As Long, *ObjectId* As Long, *ShapePointsList* As String) As Boolean

This function modifies the points composing the shape of the object which identifier is *ObjectId*.

This is done through the *ShapePointsList* parameter that represents the characters string listing the shape points in the right order.

The format of the characters string is specified in the description of the **ObjectGetShape** function above.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectWizard(*SessionId* As Long, *ObjectId* As Long) As Boolean

This function runs the wizard that is eventually associated with the object whose identifier is *ObjectId*.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ObjectValueToIncrements(*SessionId* As Long, *ObjectId* As Long, *Value* As String) As Long

This function converts the dimension or altitude values that are return in catalogue units of measure (for example by the **ObjectGetInfo** function) into a number of length increments (smallest length quantity) from the object origin catalogue.

The length increment can be the same as the catalogue unit of measure for example when working with imperial units of measure or when the unit of measure is « 1mm ». It can also be different for example when the unit of measure is « 1mm (1.0 mm) ». In that case, the unit of measure is one millimetre whereas the length increment is one tenth of millimetre.

This function is useful when developing wizards or « plug-in » functions that must work in both metric and imperial measurements.

In that case, the dimensions of the objects must be converted into length increments, then they can be processed, and finally they must be converted in characters strings (by the **ObjectIncrementsToValue** function which is described below) before being displayed or modified by the **ObjectSetInfo** function. The sloping ceiling wizard (« sdk_slop.dll » illustrates the use of these functions.

Notice : the objects coming from standard catalogues like the constraints catalogues or the decoration catalogues return dimension values in scene measurement units. In that case, the function assumes the catalogue measurement unit is the same as the current scene measurement unit.

Return value: the function returns the number of length increments corresponding to the *Value* value.

ObjectIncrementsToValue (*SessionId* As Long, *ObjectId* As Long, *IncrementsNumber* As String) As String

This function formats a number of length increments passed in the *IncrementsNumber* parameter to a characters string in catalogue measurement units (measurement unit of the catalogue that contains the object whose identifier is *ObjectId* in the current scene).

For example, a length increment number of 67 for an object coming from a catalogue having the “1/32 inch” measurement unit will give the “2 3/32” characters string.

Please, refer to the description of the **ObjectValueToIncrements** function to get more information.

Return value: the function returns the characters string representing the display value in catalogue measurement units corresponding to the number of length increments in the *IncrementsNumber* parameter.

ZoomEnlarge (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX1* As Long, *PixelY1* As Long, *PixelX2* As Long, *PixelY2* As Long) As Boolean

This function carries out a zoom in operation on the current view based on a rectangle.

The *PixelX1* and *PixelY1* parameters determine the position of one of the rectangle corner whereas *PixelX2* and *PixelY2* determine the position of the opposite corner.

The *PixelX1*, *PixelY1*, *PixelX2* and *PixelY2* parameters are given in pixels in the reference corresponding to the image which resolution is *ImageResX* x *ImageResY* and which represents the current view.

Return value: the function returns always 1.

ZoomReduce (*SessionId* As Long) As Boolean

This function carries out a zoom out operation on the current view dividing the sizes by two on the screen.

Return value: the function returns always 1.

ZoomInitial (*SessionId* As Long) As Boolean

This function sets up the zoom of the current view to its default value.

In the top view mode, the green dotted rectangle is shown with a small margin around.

In elevation view mode, the front view of the active object is shown with a small margin around.

In perspective view mode, the whole field of vision is shown.

Return value: the function returns always 1.

ZoomAdjusted (*SessionId* As Long) As Boolean

This function sets up the largest zoom possible which allows showing all the objects of the scene.

Return value: the function returns always 1.

ZoomFromPoint(*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX1* As Long, *PixelY1* As Long, *ZoomPercentage* As Long) As Boolean

This function carries out a zoom operation that is centered on the point which is defined by the *PixelX1* and *PixelY1* parameters.

The *PixelX1* and *PixelY1* parameters are given in pixels in the reference corresponding to the image which resolution is *ImageResX* x *ImageResY* and which represents the current view.

If the *ZoomPercentage* value is positive, the operation is a zoom in operation (you see the objects bigger on the screen). The zoom factor is given by the *ZoomPercentage* value. For example a *ZoomPercentage* value of 100 doubles the distances in pixels on each axis.

Reciprocally, a negative value of *ZoomPercentage* involves a reduction of the zoom factor (you see more objects but they look smaller on the screen). For example a value of -100 divides by 2 the distances in pixels on each axis.

This function can be used to manage a zoom in/zoom out function from the mouse wheel just like in KitchenDraw.

Return value: the function returns always 1.

ZoomMove(*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *StartPixelX* As Long, *StartPixelY* As Long, *EndPixelX* As Long, *EndPixelY* As Long) As Boolean

This function doesn't affect the zoom factor but it moves the current view so that the point which is defined by the *StartPixelX* and *StartPixelY* is moved to the position defined by *EndPixelX* and *EndPixelY*.

The *StartPixelX*, *StartPixelY*, *EndPixelX*, and *EndPixelY* parameters are given in pixels in the reference corresponding to the image which resolution is *ImageResX* x *ImageResY* and which represents the current view.

Return value: the function returns always 1.

ViewSetMode (*SessionId* As Long, *ViewMode* As Long) As Boolean

This function sets up the active view type.

The *ViewMode* parameter indicates what view of the current scene should be active.

<i>ViewMode</i> value	Description
VIEWMODE_2D	Top view
VIEWMODE_ELEVATION	Wireframe elevation
VIEWMODE_REALELEVATION	Realistic elevation
VIEWMODE_3D	Wireframe perspective
VIEWMODE_REAL3D	Realistic perspective
VIEWMODE_PHOTO3D = 5	Photo-realistic perspective

Return value: the function returns always 1.

ViewGetObserverHAngle (*SessionId* As Long) As Long

Return value: the function returns the value in degrees of the angle in the horizontal plane corresponding to the direction of the observer's look.

ViewGetObserverVAngle (*SessionId* As Long) As Long

Return value: the function returns the value in degrees of the angle in the vertical plane corresponding to the direction of the observer's look.

A positive value indicates the observer looks upwards.

ViewSetObserverHAngle (*SessionId* As Long, *Angle* As Long) As Boolean

This function fixes the direction of the observers's look in the horizontal plane with the *Angle* parameter (in degrees). A value of 0 forces the observer to look in the direction of the scene width (towards the right when looking at the screen in top view).

Return value: the function returns 1 if no problem occurred; else it returns 0.

ViewSetObserverVAngle (*SessionId* As Long, *Angle* As Long) As Boolean

This function fixes the direction of the observer's look in the vertical plane with the *Angle* parameter (between -90 and +90 degrees). A positive value forces the observer to look upwards.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ViewGetObserverX(*SessionId* As Long) As Long

Return value: the function returns the observer's position on the Ox axis in scene measurement units.

ViewSetObserverX(*SessionId* As Long, *Value* As Long) As Boolean

This function sets up the observer's position on the Ox axis from the *Value* parameter given in scene measurement units.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ViewGetObserverY(*SessionId* As Long) As Long

Return value: the function returns the observer's position on the Oy axis in scene measurement units.

ViewSetObserverY(*SessionId* As Long, *Value* As Long) As Boolean

This function sets up the observer's position on the Oy axis from the *Value* parameter given in scene measurement units.

Return value: the function returns 1 if no problem occurred; else it returns 0.

ViewGetObserverZ(*SessionId* As Long) As Long

Return value: the function returns the observer's position on the Oz vertical axis in scene measurement units.

ViewSetObserverZ(*SessionId* As Long, *Value* As Long) As Boolean

This function sets up the observer's position on the Oz vertical axis from the *Value* parameter given in scene measurement units.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneGetKeywordInfo (*SessionId* As Long, *Keyword* As String) As String

This function gets the information specified in the *Keyword* characters string concerning the current scene. The values that *Keyword* can take are described in the « Creation of customized WORD documents » document that can be downloaded from this link: www.kitchendraw.com/FTP/worddoceng.rtf. Any « Supplier », « Scene », « Heading » and « Base » key word can be used. The *Keyword* characters string must include the « @ » character at the beginning and the parenthesis at the end.

Return value: the function returns the requested information as a characters string. It returns an empty characters string if the *Keyword* parameter is unknown.

SceneSetKeywordInfo (*SessionId* As Long, *Value* As String, *Keyword* As String) As Boolean

This function writes the *Value* characters string in the data field of the current scene specified in the *KeyWord* parameter.

The values that *KeyWord* can take are described in the « Creation of customized WORD documents » document that can be downloaded from this link: www.kitchendraw.com/FTP/worddoceng.rtf.

Any « Supplier », « Scene », « Heading » and « Base » keyword can be used. The *KeyWord* characters string must include the « @ » character at the beginning and the parenthesis at the end.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneGetInfo(*SessionId* As Long, *InfoType* As Long) As String

This function gets the information specified by the *InfoType* parameter concerning the current scene. The values that can get the *InfoType* parameter are listed in the following table:

<i>InfoType</i> values	Description
SCNINFO_AUTO_2D_DIMENSIONS = 0	Top view automatic dimensioning flag Returns the « 1 » characters string if the top view automatic dimensioning facility is enabled or the « 0 » characters string if it's disabled.
SCNINFO_AUTO_ELEVATION_DIMENSIONS = 1	Elevation automatic dimensioning flag Returns the « 1 » characters string if the elevation automatic dimensioning facility is enabled or the « 0 » characters string if it's disabled.
SCNINFO_UNITTEXT = 2	Text representing the measurement unit of the scene
SCNINFO_UNITVALUE = 3	Value of the measurement unit of the scene in millimetres
SCNINFO_LASTOBJDROPPOSX = 4	Position on the Ox axis (width of the screen) of the last object that has been dropped in the scene <u>before any automatic positioning</u> . The value is given in scene coordinates. A value of 0 means the drop position is located in the middle of the dotted green rectangle representing the dimensions of the scene.
SCNINFO_LASTOBJDROPPOSY = 5	Position on the Oy axis (height of the screen) of the last object that has been dropped in the scene <u>before any automatic positioning</u> . The value is given in scene coordinates. A value of 0 means the drop position is located in the middle of the dotted green rectangle representing the dimensions of the scene.
SCNINFO_LASTOBJDROPPOSZ = 6	Position on the Oz axis (altitude) of the last object that has been dropped in the scene <u>before any automatic positioning</u> . The value is given in scene coordinates. A value of 0 means the object has been dropped on the floor (lower face of the dotted green parallelepiped representing the dimensions of the scene).
SCNINFO_LASTOBJDROPANGLEXY = 7	Angle in the horizontal plane (0, X, Y) of the last object that has been dropped in the scene <u>before any automatic positioning</u> . The value is given in degrees.

Return value: the function returns the requested information as a characters string. It returns an empty characters string if the *InfoType* parameter is unknown.

SceneSetInfo(*SessionId* As Long, *Value* As String, *InfoType* As Long) As Boolean

This function writes the *Value* characters string into the scene field that is specified by the *InfoType* parameter. Please refer to the description of the **SceneGetInfo** function to know all the values that can get the *InfoType* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneGetCustomInfo (*SessionId* As Long, *InfoKey* As String) As String

This function reads the characters string identified by the *InfoKey* parameter that is contained in the data area of the current scene dedicated to the application extensions (wizards and « plug-in » functions).

This characters string must have been placed in this area previously using the **SceneSetCustomInfo** function as well as the same value of the *InfoKey* parameter.

The characters strings allow a wizard or a « plug-in » function storing in the scene some custom configuration variables. Thanks to the *InfoKey* parameter each wizard or « plug-in » function can isolate its own variables from the ones of the others.

These specific characters strings appear in the XML file that is generated by the “File | Export | Management data (.XML)” command; each one under a tag equal to the value of the corresponding *InfoKey* parameter.

Return value: the function returns the requested information as a characters string.

SceneSetCustomInfo (*SessionId* As Long, *Value* As String, *InfoKey* As String) As Boolean

This function writes the *Value* characters string into the data area of the current scene dedicated to the application extensions (wizards and « plug-in » functions). This characters string will be identified by the *InfoKey* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneGetObjectsNb (*SessionId* As Long) As Long

Return value: the function returns the number of objects in the current scene whatever their type is (wall, dimension, standard object from a catalogue, linear object, etc.).

If the object is a component it will be counted only if its type is COMPTYPE_OPTION (component that is visible in the “Object | Components” dialog box) and if it’s “placed”.

Be careful: the components that are counted here will also be counted in the objects components number returned by the **ObjectGetChildrenNb** function.

Non “placed” components or invalid components are ignored.

A non “placed” component is a component that is arrived unchecked in the “Object | Components” dialog box when its parent object has been placed or that has been unchecked by the user in the “Object | Components” dialog box.

A component is declared as being invalid if one of its dimensions is not allowed in the catalogue or if the component doesn’t exist in the front model or the finishes that have been applied to it (empty price in the catalogue “Prices” table).

SceneGetObjectId (*SessionId* As Long, *Rank* As Long) As Long

Return value: the function returns the identifier of the object which rank in the scene is specified in the *Rank* parameter (starting from 1).

SceneValueToIncrements(*SessionId* As Long, *Value* As String) As Long

This function converts the dimension or altitude values that are return in scene units of measure (for example by the **SceneGetInfo** function) into a number of length increments (smallest length quantity) from the current scene.

The length increment can be the same as the scene unit of measure for example when working with imperial units of measure or when the unit of measure is « 1mm ». It can also be different for example when the unit of measure is « 1mm (1.0 mm) ». In that case, the unit of measure is one millimetre whereas the length increment is one tenth of millimetre.

This function is useful when developing wizards or « plug-in » functions that must work in both metric and imperial measurements.

In that case, the positions and dimensions of the objects must be converted into length increments, then they can be processed, and finally they must be converted in characters strings (by the **SceneIncrementsToValue** function which is described bellow) before being displayed.
The sloping ceiling wizard (« sdk_slop.dll » illustrates the use of these functions.

Return value: the function returns the number of length increments corresponding to the *Value* value.

SceneIncrementsToValue (*SessionId* As Long, *IncrementsNumber* As String) As String

This function formats a number of length increments passed in the *IncrementsNumber* parameter to a characters string in scene measurement units.

For example, a length increment number of 67 for a scene having the “1/32 inch” measurement unit will give the “2 3/32” characters string.

Please, refer to the description of the **SceneValueToIncrements** function to get more information.

Return value: the function returns the characters string representing the display value in scene measurement units corresponding to the number of length increments in the *IncrementsNumber* parameter.

SceneGetGenericsNb(*SessionId* As Long) As Long

Return value: the function returns the number of generics belonging to the current scene.

SceneAddGeneric(*SessionId* As Long, *CatalogFileName* As String) As Boolean

This function adds a generic to the current scene.

The catalogue that is used in the new generic is specified in the *CatalogFileName* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneDeleteGeneric(*SessionId* As Long, *GenericRank* As Long) As Boolean

This function deletes the generic whose rank (starting from 1) is *GenericRank* in the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneGetShapesNb(*SessionId* As Long) As Long

Return value: the function returns the number of shapes in the current scene.

SceneGetShape(*SessionId* As Long, *ShapeRank* As Long) As String

Return value: the function returns the characters string which lists the points belonging to the shape whose rank (starting from 1) is *ShapeRank* in the current scene.

The coordinates are given in decimal values and are separated with a coma. The points are separated with a semi column like in the following example:

« x1,y1,z1,t1;x2,y2,z2,t2;...;xn,yn,zn,tn;... », where « xn », « yn » and « zn » are the scene coordinates of the nth point of the shape and « tn » its property value.

The different shape point properties are listed in the following table:

Shape point properties	Description
SHAPEPOINTPROP_SELECTED = 1	Selection flag of the point
SHAPEPOINTPROP_ARC = 2	Circle arc flag: indicates the point defines a circle arc along with the previous point of the shape and the following point.

The value mentioned after the coordinates of a shape point is the sum of the properties values corresponding to this point. This way, a point that is selected and that is at the same time a circle arc point will have a property value of 3 (1 + 2).

SceneSetShape(*SessionId* As Long, *ShapeRank* As Long, *ShapePointsList* As String) As Boolean

This function modifies the points composing the shape whose rank (starting from 1) is *ShapeRank* in the current scene.

This is done through the *ShapePointsList* parameter that represents the characters string listing the shape points in the right order.

The format of the characters string is specified in the description of the **SceneGetShape** function above.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneAddShape(*SessionId* As Long, *ShapePointsList* As String) As Boolean

This function adds a shape to the current scene.

The shape points are listed in the *ShapePointsList* parameter.

The format of the *ShapePointsList* characters string is specified in the description of the **SceneGetShape** function above.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneDeleteShape(*SessionId* As Long, *ShapeRank* As Long) As Boolean

This function deletes the shape whose rank (starting from 1) is *ShapeRank* in the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneDeleteAllShapes(*SessionId* As Long) As Boolean

This function deletes all the shapes in the current scene.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SceneGetActiveShape(*SessionId* As Long) As Long

Return value: the function returns the rank (starting from 1) of the active shape in the current scene, or 0 if there is no active shape in the current scene.

SceneSetActiveShape(*SessionId* As Long, *ShapeRank* As Long) As Boolean

This function sets the active shape of the current scene thanks to the *ShapeRank* parameter (starting from 1).

Return value: the function returns 1 if no problem occurred; else it returns 0.

GenericGetCatalogFileName (*SessionId* As Long, *GenericRank* As Long) As String

Return value: the function returns the file name of the catalogue corresponding to the generic which rank is *GenericRank*. Even if most of the time the scenes owns only one generic (front model and generic finishes specification), a scene can have up to 4 generics.

GenericSetCatalogFileName (*SessionId* As Long, *GenericRank* As Long, *CatalogFileName* As String) As Boolean

This function changes the catalogue of the generic which rank is *GenericRank*.

Return value: the function returns 1 if no problem occurred; else it returns 0.

GenericGetFinishesConfig (*SessionId* As Long, *GenericRank* As Long) As String

Return value: the function returns a characters string representing the current front model and finishes configuration corresponding to the generic which rank is *GenericRank* (starting from 1) in the current scene. The characters string has the following format:

«-1,10005,10006,10007,20022,20023;;1,2,1,4,9,7».

At the left hand side of « ; ; », we can find the finishes types separated with a coma. It's possible to get the name of the finishes types corresponding to these values thanks to the **CatalogGetFinishTypeName** function of the **Appli** class.

The -1 code represents the front model; the 1XXXX codes represent the model finishes types and the 2XXXX codes represent the family finishes types.

At the right hand side of « ; ; », we can find the ranks (numbered from 0 and separated with a coma) of the finishes corresponding respectively to the finishes types listed previously. Each value represents the rank of the selected finish in the list of the available finishes for the corresponding finish type.

GenericSetFinishesConfig (*SessionId* As Long, *GenericRank* As Long, *FinishesList* As String) As Boolean

This function modifies the current front model and finishes configuration corresponding to the generic which rank is *GenericRank* (starting from 1) in the current scene.

This is done thanks to the *FinishesList* parameter that represents the characters string listing in the right order the front model followed eventually by the finishes ranks to be applied to the generic.

The finishes ranks are numbered starting from 0 and are separated with a coma.

Return value: the function returns 1 if no problem occurred; else it returns 0.

GenericModifyFinishesConfig (*SessionId* As Long, *GenericRank* As Long, *FinishesList* As String, *ModifiedLine* As Long, *NewFinish* As Long, *Modify* as Long) As String

Return value: this function returns a characters string representing the finishes configuration to be presented following the configuration modification of the generic which rank is *GenericRank* (starting from 1) in the current scene by a finish change (*NewFinish*) carried out at the level of the finish type of rank *ModifiedLine*.

The characters string has the following format:

«-1,10005,10006,10007,20022,20023;;1,2,1,4,9,7».

At the left hand side of « ; ; », we can find the finishes types separated with a coma. It's possible to get the name of the finishes types corresponding to these values thanks to the **CatalogGetFinishTypeName** function of the **Appli** class.

The -1 code represents the front model; the 1XXXX codes represent the model finishes types and the 2XXXX codes represent the family finishes types.

At the right hand side of « ; ; », we can find the ranks (numbered from 0 and separated with a coma) of the finishes corresponding respectively to the finishes types listed previously. Each value represents the rank of the selected finish in the list of the available finishes for the corresponding finish type.

If the *Modify* parameter equals 1, the configuration of the generic finishes will be actually modified.

HeadingGetObjectsNb (*SessionId* As Long, *HeadingRank* As Long, *Factorized* as Long) As Long

Return value: the function returns the number of objects belonging to the heading which rank (starting from 1) is specified in the *HeadingRank* parameter. The counted objects are those that appear in the KitchenDraw pricing table corresponding to that heading.

If *HeadingRank* equals *OBJLIST_ALLHEADINGS* (that is to say 0), the function returns the number of objects belonging to any of the scene headings.

If the *Factorized* parameter equals 1 the objects that are strictly identical are factorized that is to say are counted once. Then, the **HeadingGetObjectOccurrencesNb** function allows retrieving the occurrences number of the object.

Notice: only the objects having a price, that are valid and "placed" (for the components) belong to a heading.

HeadingGetObjectId (*SessionId* As Long, *HeadingRank* As Long, *Factorized* as Long, *Rank* As Long) As Long

Return value: the function returns the identifier of the object which rank in the *HeadingRank* heading (starting from 1) is specified in the *Rank* parameter (starting from 1).

If the *Factorized* parameter equals 1 the *Rank* parameter must take into account a possible factorisation of the strictly identical objects.

HeadingGetObjectOccurrencesNb (*SessionId* As Long, *HeadingRank* As Long, *Factorized* as Long, *Rank* As Long) As Long

Return value: the function returns the occurrences number of the object which rank in the *HeadingRank* heading (starting from 1) is specified in the *Rank* parameter (starting from 1).
If the *Factorized* parameter equals 0 the function returns 1.

SupplierGetObjectsNb (*SessionId* As Long, *SupplierID* As String, *HeadingRank* As Long) As Long

Return value: the function returns the number of objects in the heading specified by the *HeadingRank* parameter coming from a catalogue affected to the supplier specified in the *SupplierID* parameter.
If *HeadingRank* equals *OBJLIST_ALLHEADINGS* (that is to say 0), the function returns the number of objects belonging to any of the scene headings but coming from a catalogue affected to the supplier specified in the *SupplierID* parameter.
Objects which "To be ordered" attribute is not checked in the "Object | Attributs" dialog box are ignored.

Notice: only the objects having a price, that are valid and "placed" (for the components) belong to a heading.

SupplierGetObjectId (*SessionId* As Long, *SupplierID* As String, *HeadingRank* As Long, *Rank* As Long) As Long

Return value: the function returns the identifier of the object which rank in the *HeadingRank* heading (starting from 1) is specified in the *Rank* parameter (starting from 1 in the list of the objects coming from a catalogue affected to the supplier specified in the *SupplierID* parameter).

LayersSetLoad (*SessionId* As Long, *LayersFileName* As String, *LayersSetName* As String) As Long

This function loads the layers set which name is *LayersSetName* from the *LayersFileName* layers set file.
Layers sets are used to specify if objects belonging to a given layer should be displayed or not, and if they should be displayed, what drawing style should be applied to them (surface style, outline style), if the objects can be selected or not, etc.

Return value: the function returns 1 if no problem occurred; else it returns 0.

GetPointedObject (*SessionId* As Long, *ImageResX* As Long, *ImageResY* As Long, *PixelX* As Long, *PixelY* As Long, *PreviousObjectId* As Long) As Long

Return value: the function returns the identifier of the object of the scene that is located "under" the point which coordinates in pixels are *PixelX*, *PixelY*. If several objects are located "under" this point (at different altitudes), a priority list from the topmost to the topless is established.
Looping on this function changing the *PreviousObjectId* parameter makes it possible to go through that list.
For the first call to the function, the *PreviousObjectId* should be equal to -1.
The function return -1 if no object is located « under » the point which coordinates in pixels are *PixelX*, *PixelY*.

IsLoaded(*SessionId* As Long) As Boolean

This function indicates if a scene is loaded into memory and if it's possible to call functions that apply to a scene that is loaded into memory (most of the functions belonging to the Scene class).

Return value: the function returns 1 if a scene is loaded into memory and 0 if no scene is loaded.

The « Dico » class

The functions of the **Dico** class manage the dictionary files used by multilingual applications like KitchenDraw.

The **Dico** class functions list is the following:

FileNew (*SessionId* As Long, *KeyLanguageCode* As String) As Boolean

This function creates a new dictionary whose key language (the one which is used to write the basic texts) is specified in the *KeyLanguageCode* parameter.

The different language codes that can be used are listed in the « Language » combo box in the « Setup | System » KitchenDraw dialog box.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileLoad (*SessionId* As Long, *DicoFileName* As String) As Boolean

This function loads into memory a dictionary which file name (including the full access path) is specified in the *DicoFileName* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileSave (*SessionId* As Long, *DicoFileName* As String) As Boolean

This function saves the dictionary which is loaded into memory to a file which name (including the full access path) is specified in the *DicoFileName* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

FileInsertLine (*SessionId* As Long, *Ident* As String, *KeyString* As String) As Boolean

This function inserts a new line in the dictionary which is loaded into memory.

The *Ident* parameter contains the characters string which will appear in the « Ident » column of the dictionary.

The *KeyString* parameter is the text corresponding to this new entry. It will be placed in the dictionary column corresponding to the “key language”.

Return value: the function returns 1 if no problem occurred; else it returns 0.

SetStringFromKey (*SessionId* As Long, *KeyString* As String, *KeyLanguageCode* As String, *TranslatedString* As String) As Boolean

This function writes the *TranslatedString* characters string in a cell of the dictionary which is loaded into memory. This cell corresponds to the translation of the characters string defined by the *KeyString* parameter into the language specified in the *KeyLanguageCode* parameter.

Return value: the function returns 1 if no problem occurred; else it returns 0.

GetStringFromKey (*SessionId* As Long, *KeyString* As String, *KeyLanguageCode* As String) As String

Return value: the function returns the characters string corresponding to the translation of the characters string defined by the *KeyString* parameter into the language specified in the *KeyLanguageCode* parameter.

“Plug-in” functions

Functions located in DLL libraries and called “plug-in” functions are automatically fired when certain commands or actions are carried out in KitchenDraw.

These functions must have a specific name depending on the event that fired them.

For example, the *OnSceneInformationBefore()* function will be fired when the user runs the “Scene | Information” command just before the dialog box appears.

To be active, a “plug-in” function must be registered in the SPACE.INI configuration file.

Two methods can be employed concurrently:

The first one is declaring the DLL libraries in the [plugins] section of the SPACE.INI file.

```
[Plugins]
plugin1.dll=
plugin2.dll=
```

In that case, KitchenDraw analyses the content of the different registered DLL libraries and automatically registers all the individual “plug-in” functions that are included.

The second method is declaring the DLL libraries in the sections that are dedicated to each individual function. This method is more flexible since it allows registering only some functions of the DLL libraries and not all of them and also because if several DLL libraries are connected to the same event it's possible to control the processing order independently.

The function included in the DLL library that appears in first position in the list will be run first, the second then and so on.

```
[OnAppStartBefore]
mykdext.dll=
[OnFileSaveEnd]
mykdext.dll=
prodext.dll=
[OnFileOpenBefore]
mykdext.dll=
```

The “plug-in” functions list is following:

```
OnAppStartBefore,
OnAppStartAfter,
OnAppQuitBefore,
OnAppQuitAfter,

OnFileNewBefore,
OnFileNewAfter,
OnFileOpenBefore,
OnFileOpenAfter,
OnFileSaveBefore,
OnFileSaveAfter,
OnFileSaveAsBefore,
OnFileSaveAsAfter,
OnFileSaveVersionBefore,
OnFileSaveVersionAfter,
OnFileCloseBefore,
OnFileCloseAfter,
OnFileGenerateOrdersBefore,
OnFileGenerateOrders,
OnFileGenerateOrdersAfter,

OnSceneInformationBefore,
OnSceneInformation,
OnSceneInformationAfter,
OnSceneInformationDlgInit,
OnSceneInformationDlgValidation,
OnSceneInformationDlgOk,
OnSceneSpaceBefore,
OnSceneSpace,
OnSceneSpaceAfter,
OnSceneGenericFinishesBefore,
OnSceneGenericFinishes,
OnSceneGenericFinishesAfter,
OnScenePartsListBefore,
OnScenePartsList,
OnScenePartsListAfter,
OnSceneUpdatePricesBefore,
```

OnSceneUpdatePrices,
OnSceneUpdatePricesAfter,
OnSceneVATRatesBefore,
OnSceneVATRates,
OnSceneVATRatesAfter,
OnSceneCommentBefore,
OnSceneComment,
OnSceneCommentAfter,
OnSceneRenumberBefore,
OnSceneRenumber,
OnSceneRenumberAfter,
OnSceneCheckingBefore,
OnSceneChecking,
OnSceneCheckingAfter,
OnSceneLayersBefore,
OnSceneLayers,
OnSceneLayersAfter,
OnSceneDrawingStyleBefore,
OnSceneDrawingStyle,
OnSceneDrawingStyleAfter,
OnSceneMarkBefore,
OnSceneMark,
OnSceneMarkAfter,
OnSceneGridBefore,
OnSceneGrid,
OnSceneGridAfter,
OnSceneAutomaticLinearArticlesBefore,
OnSceneAutomaticLinearArticlesAfter,
OnObjectPlaceBefore,
OnObjectPlaceAfter,
OnObjectDeleteBefore,
OnObjectDeleteAfter,
OnObjectMoveBefore,
OnObjectMoveAfter,
OnObjectAttributesBefore,
OnObjectAttributes,
OnObjectAttributesAfter,
OnObjectAttributesDlgInit,
OnObjectAttributesDlgValidation,
OnObjectAttributesDlgOk,
OnObjectComponentsBefore,
OnObjectComponents,
OnObjectComponentsAfter,
OnObjectFinishesBefore,
OnObjectFinishes,
OnObjectFinishesAfter,
OnObjectPricesBefore,
OnObjectPrices,
OnObjectPricesAfter,
OnObjectSpecialTermsBefore,
OnObjectSpecialTerms,
OnObjectSpecialTermsAfter,
OnObjectCommentsBefore,
OnObjectComments,
OnObjectCommentsAfter,
OnObjectWizardBefore,
OnObjectWizard,
OnObjectWizardAfter,
OnObjectCatalogPrices,
OnObjectGrossSellingPrice.

All the « plug-in » functions must handle a unique parameter which type is Long Integer. This parameter is used to access the current scene as well as the application resources (window, ...).

All the « plug-in » functions return a Boolean value.

Here is the skeleton of a « plug-in » function written in VB6:

```
'=====
Public Function OnFileSaveAfter(ByVal lCallParamsBlock As Long) As Boolean
'=====
oAppli As Object
oScene As Object
lSessionId As Long
bOk As Boolean
lNbObjects As Long

Set oAppli = CreateObject("KDSDK.Appli")
Set oScene = CreateObject("KDSDK.Scene")

lSessionId = oAppli.StartSessionFromCallParams(lCallParamsBlock)

lNbObjects = oScene.SceneGetObjectsNb(oAppli)
MsgBox "Nombre d'objets dans la scène : " & lNbObjects

bOk = oAppli.End Session(lSessionId)

Set oAppli = Nothing
Set oScene = Nothing

OnFileSaveAfter = True
End Function
```

You can see that most of the « plug-in » functions exist in 2 or 3 or even 6 versions: one version which name ends with « Before », another one which name ends with « After » and in case the command involves the use of a dialog box a third without any ending.

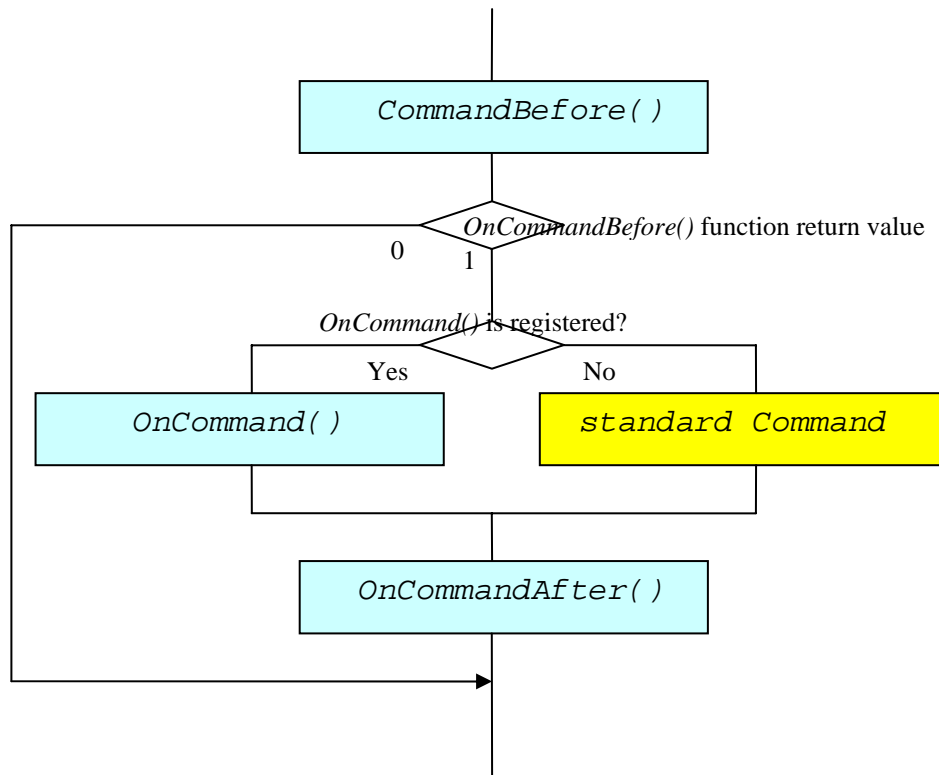
In fact, when a user runs an application command, two or three events (depending on the command) are fired consecutively.

The first event (« Before ») is fired before the execution of the standard function or the display of the dialog box. The value returned by the possible « plug-in » function conditions the command continuation: a return value of “True” allows continuing the processing whereas a return value of “False” stops it. This version of « plug-in » function allows preparing the scene prior to the standard command is run or preventing the command to be run if some conditions are not met.

If the command involves a dialog box, the second event is fired just before the display of the dialog box and only if a « plug-in » function without ending is registered. In that case, the « plug-in » function without ending will be run instead of the standard dialog box. This kind of « plug-in » functions allows replacing standard KitchenDraw dialog boxes with your own specific dialog boxes.

At last, a third event (« After ») is fired once the standard or the specific command is complete or when the standard or the specific dialog box is validated (click on “OK” or press on ENTER). The « After » version of « plug-in » functions allows running a post-processing. For example, after placing an object in the scene, a « plug-in » function could scan the articles already placed in the scene to control their compatibility or eventually place complementary objects. As a matter of fact, except in the case of components, KitchenDraw doesn't have any standard function that is conditioned by the objects in the scene. The « plug-in » functions are a flexible and powerful way to give intelligence to a catalogue.

Here is the graph representing the sequence of the various « plug-in » functions associated to a command:



For the OnSceneInformation... and the OnObjectAttributs... functions, 3 other versions exist: the first one with the « DlgInit » ending, the second one with the « DlgValidation » ending and the third one with the « DlgOk » ending.

The « DlgInit » event is fired in the initialisation part of the standard dialog box just before being displayed. The corresponding « plug-in » function could setup some information in the dialog box or even change the layout of the dialog box adding or removing controls.

The « DlgValidation » event is fired when the user has clicked on the “Ok” button or pressed on the ENTER key. The corresponding « plug-in » function could check the values of the controls in the dialog box. A False value returned by the function will prevent the dialog box to be closed.

The « DlgOk » event is fired after the « DlgValidation » event except if the « DlgValidation » function returns False. The corresponding « plug-in » function could use the values of the controls to do something.