ISA Dialog Manager

C Interface - Functions

A.06.03.c

This manual explains all C API (application programming interface) functions of the ISA Dialog Manager. It contains the function definitions with their parameters and return values.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Germany

Notation Conventions

DM will be used as a synonym for Dialog Manager.

The notion of UNIX in general comprises all supported UNIX derivates, otherwise it will be explicitly stated.

< >	to be substituted by the corresponding value
color	keyword
.bgc	attribute
{ }	optional (0 or once)
[ ]	optional (0 or n-times)
<A> | <B>	either <A> or <B>

Description Mode

All keywords are bold and underlined, e.g.

variable **integer** **function**

Indexing of Attributes

Syntax for indexed attributes:

[I]

[I,J] meaning [row, column]

Identifiers

Identifiers have to begin with an uppercase letter or an underline ('_'). The following characters may be uppercase or lowercase letters, digits, or underlines.

Hyphens ('-') are ***not*** permitted as characters for specifying identifiers.

The maximal length of an identifier is *31* characters.

Description of the permitted identifiers in the Backus-Naur form (BNF)

<identifier>	::=	<first character>{<character>}
<first character>	::=	_ | <uppercase>
<character>	::=	_ | <lowercase> | <uppercase> | <digit>

<digit> ::= 1 | 2 | 3 | … 9 | 0

<lowercase> ::= a | b | c | … x | y | z

<uppercase> ::= A | B | C | … X | Y | Z

Table of Contents

1 Introduction

This manual describes the **Application Programming Interface** (API), provided by Dialog Manager (DM) for application programs which have been written in C.

This manual explains how to implement and use the DM-API for C programs which have been written with the system-specific C compiler.

Apart from certain control functions which influence the complex DM handling the API offers the usual functionality available in the Rule Language of the application program.

This manual includes only those DM functions which are used to access or manipulate values. The basic structure of a C program is explained in the manual “C Interface - Basics”.

2 Functions of the DM Interface

In this chapter the functions of the DM interface are described. After a short summary (chapter "Overview of Functions") the different function types are introduced (chapter "Access Functions and Error Handling"). In the chapter "Functions in Alphabetical Order" you will find a complete list including the functions described in detail.

For the description of the parameters we use the following characters:

-> input parameter

<- output parameter

<-> in- and output parameter

2.1 Overview of Functions

Function Name	Short Description
AppFinish	finish routine (distributed application)
AppInit	start routine (distributed application)
AppMain	main routine of application
DM_ApplyFormat	formatting a string
DM_BindCallBacks	transferring function tables
DM_BindFunctions	transferring function tables
DM_BootStrap	first initialization of DM
DM_CallFunction	calling function in any part of application (distributed application)
DM_CallMethod	calling a method of an object
DM_Calloc	allocation of memory
DM_CallRule	calling rules by parameters
DM_Control	triggering an action
DM_ControlEx	triggering an action, extended by an additional action
DM_CreateObject	creating objects

Function Name	Short Description
DM_DataChanged	signaling that the value of an attribute has changed on a Data Model
DM_Destroy	deleting any DM objects
DM_DialogPathToID	converting external object names into internal DM_ID **deprecated, replaced by DM_ParsePath**
DM_DispatchHandler	user-defined function for handling *XEvents* at the X level (IDM FOR MOTIF only)
DM_DumpState	outputs status information into the log or trace file
DM_ErrMsgText	text belonging to an error code
DM_ErrorHandler	user-defined function for handling errors recognized by the rule interpreter
DM_EventLoop	starting dialog processing
DM_ExceptionHandler	user-defined function for handling “asserts” of the IDM
DM_Execute	starting another program
DM_FatalAppError	error handling
DM_FmtDefaultProc	format
DM_Free	releasing memory capacity
DM_FreeContent	deleting object contents (e.g. listbox)
DM_FreeVectorValue	releasing allocated memory
DM_GetContent	querying object contents (e.g. listbox)
DM_GetMultiValue	querying several DM attributes in one call
DM_GetToolkitData	querying toolkit data
DM_GetValue	querying DM attributes
DM_GetValueIndex	querying DM attributes (indication of index datatype)
DM_GetVectorValue	querying vectorial DM attributes
DM_IndexReturn	safe returning of local index values from a function
DM_Initialize	initialization of DM

Function Name	Short Description
DM_InitMSW	parsing of command line for Windows
DM_InputHandler	querying additional input sources
DM_InstallNlsHandler	retrieving text from external text catalog
DM_LoadDialog	loading dialog in DM
DM_LoadProfile	loading user-specific settings
DM_Malloc	allocation of memory
DM_NetHandler	user-defined function for manipulating data that the DDM sends over a network
DM_OpenBox	opens a messagebox or dialogbox (window with *.dialogbox* = *true*)
DM_ParsePath	search for an object in any dialog and return its object ID
DM_PathToID	converting external object name into internal DM ID **deprecated, replaced by DM_ParsePath**
DM_PictureHandler	custom function for loading images in graphic formats not supported by the IDM (graphic handlers)
DM_PictureReaderHandler	registering custom functions for loading images (graphic handlers)
DM_ProposeInputHandlerArgs	querying an unassigned message number (IDM FOR WINDOWS only)
DM_QueryBox	opening messagebox
DM_QueryError	querying errors
DM_QueueExtEvent	putting external events in the event queue
DM_Realloc	allocation of memory
DM_ResetMultiValue	resetting of DM objects to model values in one call
DM_ResetValue	resetting of DM objects to model or default values
DM_ResetValueIndex	resetting tablefield attributes to values of corresponding default attribute
DM_SaveProfile	saves the current values of configurable records and variables in a configuration file
DM_SendEvent	putting external events in the event queue

Function Name	Short Description
DM_SendMethod	asynchronous method invocation
DM_SetContent	setting object contents (e.g. listbox).
DM_SetMultiValue	changing several DM object attributes in one call
DM_SetToolkitData	setting specific window-system data
DM_SetValue	changing DM object attributes
DM_SetValueIndex	changing DM attributes (indication of index data type)
DM_SetVectorValue	changing vectorial DM attributes
DM_ShutDown	regular closing of DM
DM_StartDialog	starting actual dialog application
DM_StopDialog	finishing a dialog
DM_StrCreate	creating a string with a given character encoding
DM_Strdup	duplicating strings
DM_StringChange	modifying a string that is managed by the IDM
DM_StringInit	converting a string into string managed by the IDM
DM_StringReturn	safe returning of local strings from a function
DM_TraceMessage	writing protocol messages of the application in the DM protocol file
DM_ValueChange	replacing the entire value or a single element value of a collection
DM_ValueCount	returns the number of values, the index type or the highest index value of a collection
DM_ValueGet	retrieving a single element value that belongs to a defined index from collections
DM_ValueIndex	determines the corresponding index for a position in a collection
DM_ValueInit	converting a value reference into a local or global value reference managed by the IDM
DM_ValueReturn	safe returning of local **DM_Value** values from a function
DM_WaitForInput	waiting for a special event

Function Name	Short Description
YiRegisterUserEventMonitor	installation of event monitors (IDM FOR WINDOWS only) **Use not recommended!**

2.2 Initializing and Starting Dialog Manager

The functions which are described next are necessary for initializing and starting a dialog. You have to install the following functions in your main program:

- DM_BindCallBacks
  This function transfers all addresses of your functions to Dialog Manager so that Dialog Manager can call these functions.
- DM_BindFunctions
  This function transfers all addresses of your functions to the Dialog Manager so that Dialog Manager can call these functions. DM_BindFunctions is needed if the function is defined in modules.
- DM_EventLoop
  This function starts the processing in Dialog Manager. Without this call no interaction with the user interface would be possible.
- DM_Initialize
  This function transfers the parameter from the command line to Dialog Manager and initializes Dialog Manager.
- DM_LoadDialog
  This function loads a dialog into Dialog Manager. This dialog description may be an ASCII file or a binary file.
- DM_LoadProfile
  This function loads the user-dependent configuration file with the help of which specifically marked variables are to be changed.
- DM_StartDialog
  This function starts the dialog. It initializes the window system and makes visible the windows defined as visible in the file. The function also executes the dialog-start rule.

2.3 Access Functions

The following functions help you manipulate the objects and object attributes of Dialog Manager. You should always access the object structures by means of these functions.

2.3.1 Access to Dialog Manager Identifiers

In order to identify the individual objects, you will need the names you have given the objects in the dialog script. By these names you can query the object's internal ID of Dialog Manager. You can use this ID to inform Dialog Manager about the attribute of the object you want to query or change.

- DM_DialogPathToID **deprecated, replaced by DM_ParsePath**
  This function returns the Dialog Manager identifier of an object. DM_DialogPathToID has more abilities than just processing a dialog.
- DM_ParsePath
  This function searches for an object in any dialog and returns its object ID.
- DM_PathToID **deprecated, replaced by DM_ParsePath**
  This function returns the Dialog Manager identifier. This ID has to be used for all other calls to Dialog Manager.

2.3.2 Access to Object Attributes

To be able to access any object attribute, you need the object identifier, the data type and the attribute identifier.

- DM_DataChanged
  This function is used to signal that the value of the specified attribute (Model attribute) has changed on a particular Data Model (Model component).
- DM_GetMultiValue
  This function returns the value of several desired attributes in one call.
- DM_GetValueIndex
  This function returns the value of the desired attribute of the given object. In contrast to DM_GetValue this function operates with two indices.
- DM_GetValue
  This function returns the value of the desired attribute of each object.
- DM_ResetMultiValue
  This function resets several attributes to the value of the object model or object default in one call.
- DM_ResetValueIndex
  This function resets the tablefield attribute to the value of the relevant default attribute (Index 0).
- DM_ResetValue
  This function resets the attribute to the value of the object model.
- DM_SetMultiValue
  This function changes in one call several attributes to the transferred value.
- DM_SetValueIndex
  This function changes the attribute to the transferred value. In contrast to DM_SetValue this function operates with two indices.
- DM_SetValue
  This function changes the attribute to the transferred value.

2.3.3 Handling Vectorial Attributes

These functions help you in using so-called "vector attributes", i.e. attributes occurring several times at an object.

» DM_FreeVectorValue
By using this function you can release allocated memory space.

» DM_GetVectorValue
This function can return several attribute values.

» DM_SetVectorValue
By using this function you can set several attribute values.

2.3.4 Handling Complex Vectorial Attributes

To process the contents of a tablefield or a listbox efficiently, you can use the following functions:

» DM_FreeContent
This function releases the allocated memory space for an object contents which has been returned due to a call of DM_GetContent.

» DM_GetContent
This function returns the present contents of an object.

» DM_SetContent
This function replaces the present contents with a new contents.

2.3.5 Creating and Destroying Objects

Objects can be dynamically created or destroyed by using the following two functions.

» DM_CreateObject
This function creates an object of a specific class and returns its identifier. After the object having been successfully created, a DM function can access the object.

» DM_Destroy
This function destroys each object. After a call of this function no DM function can access these objects any more.

2.3.6 Memory Administration Functions

By using the memory administration functions described next you can allocate and release portable memory without having to access the functions optimal for the relevant operation system. If these functions are used in an application, you absolutely have to take care that the memory space which has been allocated by the functions described here, may only be released by these function. In principle, you can mix different methods of memory allocation; once you have allocated memory, you can only process it by using the same kind of functions.

» DM_Calloc
This function allocates memory in the given size and given number.

» DM_Free
This function releases allocated memory.

- DM_Malloc
  This function allocates memory in the given size.
- DM_Realloc
  This function allocates memory in a given size.

2.3.7 Service Functions (Utilities)

- DM_CallMethod
  By using this function you can call a method of an object from a program.
- DM_CallRule
  By using this function you can call named rules with parameters from an application.
- DM_Control
  By using this function the application can rebuild the screen, switch the code page currently used, or lock the keyboard.
- DM_ControlEx
  This function can be used to change general settings in the ISA DIALOG MANAGER or to trigger actions.
- DM_Execute
  This function starts another program.
- DM_DumpState
  This function writes status information into the log or trace file.
- DM_FmtDefaultProc
  If an editable text has a format, this function will be called for all occurring tasks (e.g. setting the format, input control, navigation etc.).
- DM_IndexReturn
  This function is used to safely return local index values (**DM_Index**) from a function.
- DM_OpenBox
  This function opens a messagebox or a dialogbox (window with attribute *.dialogbox* = *true*).
- DM_ProposeInputHandlerArgs
  With this function a still unassigned message number for DM_InputHandler can be queried (IDM FOR WINDOWS only).
- DM_QueryBox
  By using this function you can open messageboxes.
- DM_QueueExtEvent
  By using this function you can put an event in a queue processing a rule which is assigned to an external event.
- DM_SaveProfile
  This function writes the current values of all configurable ***record*** instances and **global variables** of a dialog or module into a configuration file (profile).
- DM_SendEvent

By using this function you can put an event in a queue processing a rule which is assigned to an external event.
In contrast to **DM_QueueExtEvent**, the external event is indicated through a **DM_Value** structure, which facilitates the utilization of message resources.

- DM_SendMethod
  This function puts a method call into the event queue to be executed asynchronously from the event loop.
- DM_TraceMessage
  By using this function the application can write strings in the tracefile.
- DM_ValueReturn
  This function is used to safely return local **DM_Value** values from a function.

See also

Chapter “Integrating Custom Functions (Handlers)”

2.3.8 Special Functions

These functions are normally not used, i.e. only in exceptions, since they are called automatically by Dialog Manger via the modules **startup.o** or **startup.obj**. Only if one of these modules is replaced (with one of its own), these functions have to be called.

- DM_BootStrap
  By using this function you can initialize Dialog Manager. DM_BootStrap has to be the first function to be called in DM.
- DM_InitMSW
  By using this function the command line of a Windows program can be broken down into its individual constituents.
- DM_ShutDown
  By using this function DM can be closed properly.

2.3.9 Linking a Window System

To link the application to the window system, the following functions are available:

- DM_GetToolkitData
  This function returns the window-system-specific data of an attribute.
- DM_SetToolkitData
  By using this function you can overwrite window-system-specific data of an attribute.

See also

Chapter “Integrating Custom Functions (Handlers)”

2.4 Error Processing

If a function call to Dialog Manager returns FALSE, the application can query the error with help of the functions which are described in this chapter:

- DM_ErrMsgText
  This function returns the text string to an error code.
- DM_FatalAppError
  This function should be called if the application has discovered a serious error and cannot continue operating any more. It finishes any work in Dialog Manager.
- DM_QueryError
  This function returns the number of actual fields together with the error codes.

See also

Chapter “Integrating Custom Functions (Handlers)”

2.4.1 Information in the Error Code

The error code is a combination of three kinds of information:

- the severity of the error
- the error code itself
- the module in which the error occurs.

To extract the error from the error code variable, you can use the macro

```
DM_ExtractSev(ERRNO)
```

The following error degrees are possible:

DM_SeveritySuccess	no error
DM_SeverityWarning	warning
DM_SeverityError	error
DM_SeverityFatal	serious error, DM cannot continue work
DM_SeverityProgErr	program error

To extract the module which has caused the error, the macro

```
DM_ExtractModule(ERRNO)
```

can be used.

The following modules are possible:

DM_ModuleIDM	the error is within DM

DM_ModuleUnix the error occurs on a function call to the operation system UNIX

DM_ModuleMpe the error occurs on a function call to the operation system MPE

DM_ModuleVms the error occurs on a function call to the operation system VMS

To extract the error, the macro

```
DM_ExtractErrno(ERRNO)
```

can be used.

The possible error codes are defined in **IDMuser.h**.

2.5 Working with Collections

- DM_ValueChange
  With this function a value reference managed by IDM may be manipulated. Either the entire value can be replaced or a single element value in a collection.
- DM_ValueCount
  Returns the number of values in a collection (without the default values). It is also possible to return the index type or the highest index value.
- DM_ValueGet
  This function allows to retrieve a single element value that belongs to a defined index from collections.
- DM_ValueIndex
  This function can be used to determine the corresponding index for a position in a collection.
- DM_ValueInit
  With this function a value reference can be converted into a local or global value reference managed by the IDM. This allows the further manipulation of the value by **DM_Value…()** functions and its transfer as parameter or return value.

2.6 String Functions

- DM_StrCreate
  With this function a text with a given character encoding can be created.
- DM_Strdup
  By using this function you can duplicate strings.
- DM_StringChange
  This function allows to modify a string that is managed by the IDM.
- DM_StringInit
  This function converts a string into a local or global respectively static string managed by the IDM.
- DM_StringReturn
  This function is used to safely return local strings from a function.

2.7 Integrating Custom Functions (Handlers)

The C interface of the IDM provides the possibility to integrate own “handlers” to individually respond to events or errors as well as to add own functionality. For these functions that have to be implemented by the user, the IDM specifies the data type and the parameters and provides functions for registering, unregistering, activating and deactivating the “handlers” with the IDM. The custom “handlers” are then called by the IDM in the respective situations (event, error...) for which they are intended.

- DM_DispatchHandler
  Setting up custom functions to process *XEvents* at the X level (IDM FOR MOTIF only).
- DM_ErrorHandler
  With this function a handler function, which is invoked with errors recognized by the rule interpreter, can be set up.
- DM_ExceptionHandler
  Setting up custom function for handling “asserts” of the IDM.
- DM_InputHandler
  Setting up custom functions to handle additional events (messages) of the window system.
- DM_InstallNlsHandler
  By using this function you can install a function for the administration of external text catalogs.
- DM_NetHandler
  Installation of a user-defined function for manipulating data that the DDM sends over a network, e.g. to encrypt the data.
- DM_PictureHandler
  Custom function (graphics handler, GFX handler) for loading images in graphic formats not supported by the IDM.
- DM_PictureReaderHandler
  Setting up graphics handlers (GFX handlers) for loading images in graphic formats not supported by the IDM.
- YiRegisterUserEventMonitor **Use not recommended!**
  Installation of event monitors that can interrupt the event loop of the IDM (IDM FOR WINDOWS only).

2.8 Handling of String Parameters

If strings are retrieved from the dialog via the interface functions of Dialog Manager out of the application, DM will copy the corresponding string in a temporary buffer from which the application can take the relevant string. It is not allowed to write in this buffer, since the application does not know in which size the memory space has been allocated for the string. On the next call to Dialog Manager which is providing again a string as result, this temporary buffer is overwritten. This is not the case, however, if this function has been called by the option DMF_DontFreeLastStrings. Only by using this option the value of the string will not be overwritten. If this option is not set, then the string which has been received before cannot be accessed any more.

Example

```
DM_GetValue (Obj, AT_text, ...);
    =>     string is copied into temporary buffer
DM_GetValue (Obj1, AT_text, ...);
    =>    string is copied, old string is not valid
        any more!

DM_GetValue(Obj, AT_text, ...);
    =>     string is copied in temporary buffer
DM_GetValue(Obj1, AT_text, ...., DMF_DontFreeLastStrings);
    =>    text of object is copied and old string
         is still valid
DM_GetValueIndex(Obj2, AT_content, ...);
    =>     string is copied, both of the strings queried
         before are no longer valid.
```

2.9 Protection in the Programming Interface

The programming interface offers functions which may be called by the application. But not every DM function may be called at any time. For example, **DM_Initialize** has to be called at the beginning, only after that **DM_LoadDialog** may be called. For these constellations there are protection measures checking whether the call of a function is allowed at a certain time. If this function may not be called, the error code will be set (*DME_WrongRunState*).

2.9.1 States of Dialog Manager

Dialog Manager passes different states during the program course. Each DM function may only be called in certain states. By using certain DM functions, Dialog Manager changes the state or accepts a different state temporarily.

U	uninitialized (initial state)
B	booted
I	initialized
M	in the main loop
W	function which is near the window system is presently active
F	calling a format function
Q	within **DM_QueueExtEvent**
S	stopped (final state)

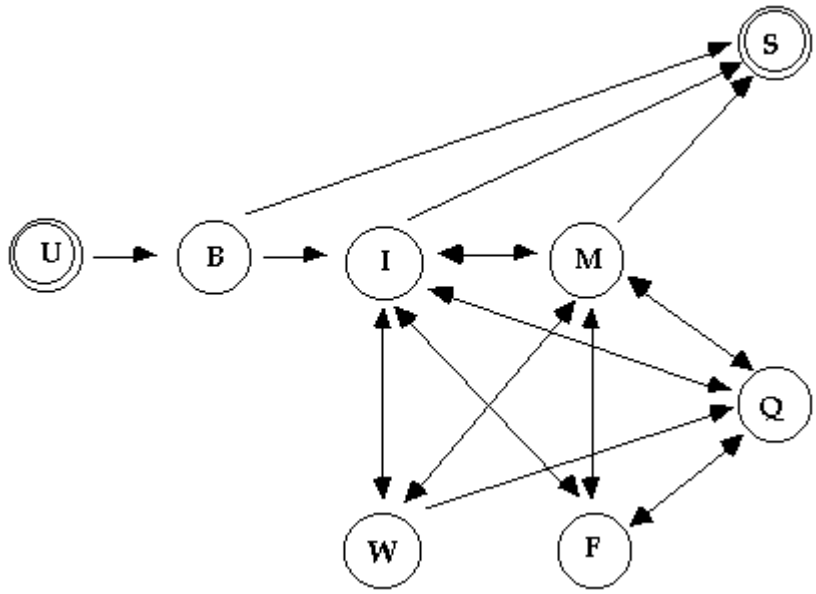

Figure 1: *State transitions in DM*

2.9.2 Transitions between States

The following functions cause a transition from one state to another state:

Function	before	while	after
DM_BootStrap	U	-	U, B
DM_EventLoop	I	M	I
DM_Initialize	B	-	B, I
DM_QueueExtEvent	I, M, W, F	Q	I, M, W, F (like before)
DM_SendEvent	I, M, W, F	Q	I, M, W, F (like before)
DM_SendMethod	I, M, W, F	Q	I, M, W, F (like before)
DM_ShutDown	B, I, M	-	S

2.9.3 Permitted States for Functions

The call of the following functions is only permitted in the marked states:

Function	U	B	I	M	W	F	Q	S
DM_ApplyFormat			X	X				
DM_BindCallBacks			X	X				

Function	U	B	I	M	W	F	Q	S
DM_BindFunctions			X	X				
DM_BootStrap	X							
DM_CallFunction			X	X				
DM_CallMethod			X	X				
DM_CallRule			X	X				
DM_Control			X	X				
DM_ControlEx			X	X				
DM_CreateObject			X	X				
DM_DataChanged			X	X	X	X		
DM_Destroy			X	X				
DM_DialogPathToID			X	X	X	X		
DM_ErrMsgText		X	X	X	X	X		
DM_EventLoop			X	X				
DM_FmtDefaultProc			X	X		X		
DM_FreeContent			X	X				
DM_FreeVectorValue			X	X				
DM_GetContent			X	X	X			
DM_GetMultiValue			X	X	X			
DM_GetToolkitData			X	X	X			
DM_GetValue			X	X	X			
DM_GetValueIndex			X	X	X			
DM_GetVectorValue			X	X	X			
DM_Initialize		X						
DM_InputHandler		X	X	X				
DM_InstallNlsHandler			X					

Function	U	B	I	M	W	F	Q	S
DM_LoadDialog			X	X				
DM_LoadProfile			X	X				
DM_OpenBox			X	X				
DM_ParsePath			X	X	X	X		
DM_PathToID			X	X	X	X		
DM_PictureReaderHandler		X	X	X				
DM_QueryBox			X	X				
DM_QueryError		X	X	X	X	X		
DM_QueueExtEvent			X	X	X	X		
DM_ResetMultiValue			X	X				
DM_ResetValue			X	X				
DM_ResetValueIndex			X	X				
DM_SaveProfile			X	X				
DM_SendEvent			X	X	X	X		
DM_SendMethod			X	X	X	X		
DM_SetContent			X	X				
DM_SetMultiValue			X	X				
DM_SetToolkitData			X	X	X			
DM_SetValue			X	X				
DM_SetValueIndex			X	X				
DM_SetVectorValue			X	X				
DM_ShutDown		X	X	X				
DM_StartDialog			X	X				
DM_StopDialog			X	X				
DM_TraceMessage		X	X	X	X	X		

There is no checking for the following functions:

- DM_Calloc
- DM_FatalAppError
- DM_Free
- DM_Malloc
- DM_ProposeInputHandlerArgs
- DM_Realloc
- DM_Strdup
- DM_WaitForInput

2.9.4 SetValue from Canvas Functions

The functions

- DM_ResetValue
- DM_ResetValueIndex
- DM_SetValue
- DM_SetValueIndex

are permitted in canvas functions only for changing variables.

3 Functions in Alphabetical Order

3.1 AppMain

This function is the main function of the application. It is called by the DM immediately after the program start and is then able to trigger the corresponding actions. It gets the same parameters as the usual "main" function of a C program.

```
int DML_c DM_CALLBACK AppMain
(
  int argc,
  char far * far *argv
)
```

Parameters

-> int argc

In this parameter the number of those command-line arguments which have been indicated at the program start are transferred.

-> char far * far * argv

In this parameter the arguments indicated on the command line are transferred.

Example

```
int DML_c DM_CALLBACK AppMain (argc, argv)
int argc;
char far *far *argv;
{
  DM_ID dialogID;

  if (DM_Initialize (&argc, argv, 0) == FALSE)
    return (1);
  if (!(dialogID = DM_LoadDialog ("dialogname", 0)))
    return (1);

  if (!DM_BindCallBacks((DM_FuncMap *) 0, 0, dialogID, 0))
    DM_TraceMessage ("Function binding incomplete", 0);

  DM_StartDialog (dialogID, 0);
  DM_EventLoop (0);
  return (0);
}
```

If an application not linked to the dialog is to be started/ended in a distributed environment, the functions

» AppInit

» AppFinish

have to be used instead of the main function.

See Also

Object Application

Chapter “Application Interface” in manual “Distributed Dialog Manager (DDM)”

3.1.1 AppInit

This function is the main function of a distributed DM application. This function starts the application and executes the steps necessary for the initialization.

```
int DML_c DM_CALLBACK AppInit
(
  DM_ID applID,
  DM_ID dialogID,
  int argc,
  char far * far *argv
)
```

Parameters

-> DM_ID applID

Identifier of the application which has been started.

-> DM_ID dialogID

Identifier of the dialog to which the application belongs.

-> int argc

Number of the transferred command-line parameters.

-> char far * far *argv

List of command-line parameters.

Return Value

0	The application has been initialized and started successfully.
!=0	The application could not be initialized and started successfully, and that the return value contains the error code.

Example

```
/*
* setup the function table of functions which should be passed
* to Dialog Manager
```

```
*/
#define ApplFuncCount
  (sizeof(ApplFuncMap)/ sizeof(ApplFuncMap[0]))

static DM_FuncMap ApplFuncMap[ ] = {
  { "FillListbox",   (DM_EntryFunc) FillListbox   },
  { "FillContinue",  (DM_EntryFunc) FillContinue  },
  { "QueryListbox",  (DM_EntryFunc) QueryListbox  },
  { "CheckFilename", (DM_EntryFunc) CheckFilename }
};

int DML_c DM_CALLBACK AppInit__4(
(DM_ID appl),
(DM_ID dialog),
(int argc)
(char far * far * argv))
{
  if (!DM_BindFunctions(ApplFuncMap, ApplFuncCount,
    appl, 0, DMF_silent))
      DM_TraceMessage ("There are some functions missing", 0);

  return 0;
}
```

3.1.2 AppFinish

The application is finished by means of this function. It is meant to carry out the necessary steps so that the applications can be finished correctly.

```
int DML_c DM_CALLBACK AppFinish
(
  DM_ID applID,
  DM_ID dialogID
)
```

Parameters

-> DM_ID applID

Identifier of application which has been finished.

-> DM_ID dialogID

Identifier of dialog to which application belongs.

Return Value

0	The application could be finished successfully.

!=0	The application could not be finished successfully and that the return value contains the error code.

Example

```
int DML_c DM_CALLBACK AppFinish (applID, dialogID)
DM_ID applID;
DM_ID dialogID;

{
  return 0;
}
```

3.2 DM_ApplyFormat

By using this function a string can be formatted so as it would be displayed with a given format. Usually, formats are available only in edittexts and tablefields, i.e. only there you can format the output text by means of defined formats or format functions. With this function you can format any texts you want.

```
DM_String DML_default DM_EXPORT DM_ApplyFormat
(
  DM_ID formatOrFunc,
  DM_String formatString,
  DM_String string,
  DM_Options options
)
```

Parameters

-> DM_ID formatOrFunc

In this parameter the ID of the format or the format function which is to format the string is indicated. If the Null-ID is indicated, a valid format string has to be indicated in the next parameter.

-> DM_String formatString

In this parameter you can indicate any valid format string with the help of which the string is to be formatted. This parameter will only be considered if you have indicated the parameter "formatOrFunc" with the ID of a format function or if you have chosen ID= 0. If in the parameter "formatOrFunc" the ID of a format is indicated, this parameter will be ignored.

-> DM_String string

In this parameter you can indicate the string to be formatted by the indicated format.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

The function returns the string which has been formatted by the chosen format; this is exactly that string which would be displayed in an input field by the chosen format.

Example

A string is to be formatted out of a C function.

```
void UseDMFormat __0()
{
  DM_String format = "NN.NN.NNNN";
  DM_String string = "12121995";

  DM_TraceMessage("Ergebnis DM_ApplyFormat: %s",DMF_Printf,
    DM_ApplyFormat((DM_ID) 0, format, string, 0));
```

```
}
```

In the trace file the formatted string appears as follows:

```
"12.12.1995"
```

See Also

Built-in function applyformat in manual “Rule Language”

3.3 DM_BindCallBacks

After having loaded the dialog a table of functions can be passed on to DM by means of this function. By using this table the DM then calls the functions indicated in the dialog.

```
DM_Boolean DML_default DM_EXPORT DM_BindCallBacks
(
  DM_FuncMap funcmap,
  DM_UInt funccount,
  DM_ID dialogID,
  DM_Options options
)
```

Parameters

-> DM_FuncMap funcmap

This parameter is the table of functions which can be called directly by DM. The structure of this table is described in the chapter on data structures.

-> DM_UInt funccount

This parameter indicates the size of the transferred function tables.

-> DM_ID dialogID

This parameter is the ID of the dialog for which this function table is to be valid. You have received this ID from the function DM_LoadDialog as return value.

-> DM_Options options

Here the following options are possible:

Option	Meaning
DMF_Silent	This option means that no error messages are to be output to the user. Otherwise a difference between superfluous functions and missing functions will be made.
DMF_Verbose	This option means that error messages are to be output to the user.

Example

A table with three functions is to be transferred to DM.

In order to initialize the function table with the addresses of these functions, these functions have to be declared at least before the table definition.

Declaration in the Header File

```
DM_boolean DML_c DM_ENTRY ReadZip __((DM_String));
char* DML_c DM_ENTRY QueryZip __((DM_String));
DM_boolean DML_c DM_ENTRY WriteFile __((DM_String, DM_String,
         DM_String));
```

Definition of the Table in the C Program

```
#define FuncCount (sizeof(FuncMap)/ sizeof(FuncMap[0]))

static DM_FuncMap far FuncMap[ ] = {
{ "ReadZip",         (DM_EntryFunc) ReadZip},
{ "QueryZip",         (DM_EntryFunc) QueryZip},
{ "WriteFile",         (DM_EntryFunc) WriteFile},
};

/* Install table of application functions */

{

  if (!DM_BindCallBacks(FuncMap, FuncCount, dialogID, 0))
    DM_TraceMessage ("There are some functions missing.", 0);

  return 0;
}
```

3.4 DM_BindFunctions

By means of this function, a table of functions can be transferred to DM, after having loaded the dialog. This function is very similar to the function DM_BindCallBacks; the difference is, however, that this function has to be used if the relevant dialog is structured in modules and if the functions have been defined in modules.

By using these functions, other functions defined in modules which have not been loaded yet can be linked to the dialog so that these functions can be called immediately after having loaded the module.

```
DM_Boolean DML_default DM_EXPORT DM_BindFunctions
(
  DM_FuncMap funcmap,
  DM_UInt funccount,
  DM_ID objID,
  DM_ID moduleID,
  DM_Options options
)
```

Parameters

-> DM_FuncMap funcmap

This parameter is the table of functions which can be directly called by DM. The structure of this table is described in the chapter on data structures. This table can be created automatically by the simulation program via the option +writefuncmap.

-> DM_UInt funccount

This parameter indicates the size of the transferred function table.

-> DM_ID objID

This parameter is the ID of the object to which this table is to be linked. This ID may be either a dialog, a module or an application.

-> DM_ID moduleID

This parameter is the ID of the module which is to supply its function immediately from this table. This ID has to be specified only if a module has been reloaded and if the functions are only then linked to the superordinate instance. Usually, the ID has to be specified by 0.

-> DM_Options options

Currently, the following values are possible:

Option	Meaning
DMF_ReplaceFunctions	This option means that a function table which probably exists has to be replaced completely by a new table at the indicated object. If this option is not specified, the new table will be attached at the end of an existing table of the object.

Example

A C file is created from a module via the **+writefuncmap** option. The dialog file has the following structure:

```
module ModFuncDate
application TimeServer
{
  !! Get the current date from the server to synchronize the
  !! clients
  !! example: CurrentDate( Year, Month, Day) returns true
  !! if successful and fills the three variables with the
  !! appropriate values
  function boolean CurrenDate( integer Year output,
    integer Month output, integer Day output);

  !! Get the current time, see function CurrentDate
  function boolean CurrentTime( integer Hour output,
    integer Minute output, integer Second output);
}
```

The generated C program looks as this:

```
#include "IDMuser.h"
#include "dateappl.h"

#define FuncCount_TimeServer (sizeof(FuncMap_TimeServer) / sizeof(FuncMap_
TimeServer[0]))

static DM_FuncMap FuncMap_TimeServer[] =
{
/*
**  Get the current date from the server to synchronize the
**  clients
**  example: CurrentDate( Year, Month, Day) returns true
**  if successful and fills the three variables with the
**  appropriate values
*/
  { "CurrenDate", (DM_EntryFunc) CurrenDate },
/*
**  Get the current time, see function CurrentDate
*/
  { "CurrentTime", (DM_EntryFunc) CurrentTime },
}

DM_Boolean DML_default BindFunctions_TimeServer __3(
    (DM_ID, dialogID),
    (DM_ID, moduleID),
```

```
    (DM_Options, options))
{
return (DM_BindFunctions (FuncMap_TimeServer,
    FuncCount_TimeServer,dialogID,moduleID,options))
}
```

3.5 DM_BootStrap

With this function, the DM is internally initialized, the arguments of the command line are saved, and the first action is carried out (-DMerrfile, -DMtracefile). In DM, this function must be the first function to be called by the application. Usually this call is made by the "Main" program which then calls the real main program of the application, **AppMain**. This is why this function may only be called if the **Main** of DM is replaced by an application-specific one.

```
int DML_default DM_EXPORT DM_BootStrap
(
  int *argcp,
  char far * far * far *argvp
)
```

Parameters

<-> int *argcp

In this parameter the address is given to the number of parameters. If the arguments are already being processed, this number can then be changed by DM_BootStrap.

<-> char far * far * far *argvp

This parameter works as a pointer to the arguments of the command line. All arguments processed by DM_BootStrap are removed from this command line.

Return Value

0	Initialization has been carried out without an error; DM can be started normally.
!=0	An error has occurred during initialization; the program must not be continued.

Example

Excerpt from the file "startup.c" by which the Dialog Manager programs are started:

```
int cdecl main __2(
(int, argc),
(char far * far *, argv))
{
  register int status;
  static char running = 0;

  if ((status = running++) == 0)
  {
    if ((status = DM_BootStrap(&argc, &argv)) == 0)
    {
      DM_InitOptions(&argc, argv, 0);
      status = AppMain (argc, argv);
    }
  }
```

```
  return (status);
}
```

3.6 DM_CallFunction

To call any functions known to Dialog Manager in other parts of the application, the function DM_CallFunction has to be used. The function calls the corresponding function in any application part.

```
DM_Boolean DML_default DM_EXPORT DM_CallFunction
(
  DM_ID funcID,
  DM_UInt argcount,
  DM_Value *argvec,
  DM_Value *result,
  DM_Options options
)
```

Parameters

-> DM_ID funcID

Identifier of the function to be called.

-> DM_UInt argcount

Specifies the number of parameters that are to be used when calling the function. The assignment of parameters can be read from the corresponding element in the argvec parameter. argcount must correspond exactly to the actual assignment of the argvec parameter. The maximum number is 8.

<-> DM_Value *argvec

This parameter is an array of DM_Value values which are to be used as parameters for the function call. The length of the vector has to correspond exactly to the value of argcount.

<- DM_Value *result

This returns the return value of the function, if the function call could be carried out.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	Function has been called successfully.
FALSE	Function could not be called.

Example

A function defined in the dialog is to be called by means of an integer parameter.

```
void DML_default DM_ENTRY CALLFUNC __((DM_ID Funktion))
{
  DM_Value argv;
  DM_Value retval;
```

```
  argv.type = DT_integer;
  argv.value.integer = 888;
  if (DM_CallFunction (Funktion, 1, &argv, &retval, 0))
    if (retval.type == DT_integer)
        printf(Erhaltener Wert: %ld", retval.value.integer);

}
```

See Also

Object Application

Manual “Distributed Dialog Manager (DDM)”

3.7 DM_CallMethod

With this function you can call object methods from the application. The parameter values depend on the method and the object at which this method is to be called.

```
DM_boolean DML_default DM_EXPORT DM_CallMethod
(
  DM_ID object,
  DM_Method method,
  DM_UInt argc,
  DM_Value *argvc,
  DM_Value *retval,
  DM_Options options
)
```

Parameters

-> DM_ID object

This parameter describes the object whose method is to be called.

-> DM_Method method

This parameter describes the method which is to be called at this object. The following parameters of this function are assigned according to this value.

-> DM_UInt argc

This parameter specifies the number of valid method parameters.

-> DM_Value *argv

Array of DM_Value structures which include the valid parameters of the called method.

The length of this parameters has to correspond absolutely to the number given at the parameter "argc". The maximal length of this array is 8.

<- DM_Value *retval

Pointer to a DM_Value structure which is used by this function as return value.

An element is set in this structure according to the called method.

-> DM_Options options

You can specify the option *DMF_DontFreeLastStrings* to keep the memory of the string parameters valid.

Return Value

TRUE	Method was carried out successfully.
FALSE	Method was not carried out successfully.

At present, the following methods exist:

Object	Method	C Constant
tablefield	clear	MT_clear
	insert	MT_insert
	delete	MT_delete
	exchange	MT_exchange

Example

Unloading a tablefield in a reloading function .

```
void DM_CALLBACK ContFunc (DM_ContentArgs* args)
{
    DM_Value methArgs[2];
    DM_Value retval;
    DM_Value first, last;
    ushort ldStart, ldCount;
    ushort visStart;

    ldStart = args->loadfirst - args->header;
    ldCount = args->loadlast - args->loadfirst + 1;

    visStart = args->visfirst - args->header;

    if (visStart > 20)
    {
        /*
        *  On loading, two parameters have to be
        *  specified for the method.The start has to
        *  take place and end index between which the
        *  deletion has to take place. Both of them
        *  have the datatype integer.
        */
        methArgs[0].type = DT_integer;
        methArgs[1].type = DT_integer;
        methArgs[0].value.integer = args->header + 1;
        methArgs[1].value.integer = visStart - 20;
        DM_CallMethod(args->object, MT_clear, 2, methArgs,
             &retval, 0);
    }
}
```

3.8 DM_Calloc

With this function you can allocate memory. DM_Calloc is carried out in dependence of the used operating system with the relevant available functions.

Memory allocated with DM_Calloc must be released with DM_Free or modified with DM_Realloc only!

```
DM_Pointer DML_default DM_EXPORT DM_Calloc
(
  DM_UInt4 nelem,
  DM_UInt4 elsize
)
```

Parameters

-> DM_UInt4 nelem

This parameter specifies the number of elements to be allocated.

-> DM_UInt4 elsize

This parameter specifies the size of an element to be allocated.

Warning

Please note that, on MS Windows, nelem * elsize is not > 64 KByte.

Return Value

Pointer is on the allocated memory. If the memory could not be allocated, the *NULL* pointer is returned.

In contrast to DM_Malloc, the memory is initialized at 0.

Example

For 10 elements memory is to be allocated for the structure DM_Content.

```
DM_Content *content;

if ((content = DM_Calloc (10, sizeof (DM_Content))
{
  ...
}
```

3.9 DM_CallRule

Specified rules and parameters can be called from the application with this function.

```
DM_Boolean DML_default DM_EXPORT DM_CallRule
(
  DM_ID ruleID,
  DM_ID objectID,
  DM_UInt argc,
  DM_Value *argv,
  DM_Value *retval,
  DM_Options options
)
```

Parameters

-> DM_ID ruleID

Identifies the rule to be executed.

-> DM_ID objectID

Corresponds to *this* in the corresponding rule, i.e. *this* in the rule gets the object you have specified here.

-> DM_UInt argc

Number of rule parameters.

-> DM_Value *argv

Parameter as an array of DM_Value structures. This parameter has to correspond exactly to the number given in argc. The maximum number of parameters is 16.

<- DM_Value *retval

Pointer to a DM_Value structure which is filled as the return value.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	Rule was executed successfully.
FALSE	Rule could not be executed.

Example

Call of a rule in the dialog. The rule is defined as follows:

```
rule integer Callrule (integer I input)
{
  I := I + atoi(Et.content);
  Et.content := itoa(I);
```

```
  print I;
  return(I);
}
```

The C program here looks as this:

```
void DML_default DM_ENTRY CALLRULE __((DM_ID Rule))
{
  DM_Value argv;
  DM_Value retval;

  argv.type = DT_integer;
  argv.value.integer = 888;
  if (DM_CallRule (Rule, Rule, 1, &argv, &retval, 0))
    if (retval.type == DT_integer)
        printf(Received Value: %ld", retval.value.integer);
}
```

3.10 DM_Control

This function can be used to change general settings in the ISA DIALOG MANAGER or to trigger actions.

```
DM_boolean DML_default DM_EXPORT DM_Control
(
  DM_ID      objectID,
  DM_UInt    action,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Object for which the specified action shall be performed.

-> DM_UInt action

Action to be performed by the IDM. For this purpose, several constants are defined in the include file **IDMuser.h**. These constants are explained in the table below.

-> DM_Options options

Contains an argument for the *action*, if required (see table below).

Return value

DM_TRUE	The action has beeen executed successfully.
DM_FALSE	The action could not be executed.

The following table shows the valid assignments of the individual parameters and explains their meaning. When nothing else is stated with the action, the *objectID* has to be *0*.

action	options	Meaning
DMF_UpdateScreen	*0*	All internal SetVal calls shall be displayed on the screen. In this case, the first parameter has to be assigned with the dialog.

action	options	Meaning
DMF_UIAutomationMode		With this action the specific UI Automation support of the IDM for its specific objects can be disabled. However, the UI Automation support provided by Microsoft for the standard controls remains active. UI Automation support is active by default. The switching must happen before calling **DM_Initialize()** and after bootstrapping.
	0	Disables the UI Automation support of the IDM.
	1	Enables the UI Automation support of the IDM.
DMF_PCREBinding	*0*	Disables the linking to the PCRE library, thus Regular Expressions are no longer possible.
	1	Linking to statically present PCRE functions in the executable (linking type E).
	2	Only dynamic linking of PCRE libraries relative to the application (linking type A).
	3	Linking to PCRE libraries relative to the application or from the system (linking sequence A – S).
	4	Linking with priority for functions in the executable (linking sequence E – A – S), this is the standard for self-built IDM applications.
	5	Linking analog to *4* but in reverse order, i.e. precedence for the PCRE library installed in the system (linking sequence S – A – E).
	See also Chapter “PCRE Library for Support of Regular Expressions” at the built-in function regex	
DMF_SignalMode	*0*	The signals are intercepted by the function **signal**.
	1	The signals are intercepted by the function **sigaction**.

action	options	Meaning
DMF_SetSearchPath	*0*	This action sets the search path for IDM files (dialog, module, interface, and binary files). The semicolon-separated directories have to be passed as string pointers in the *data* parameter. **See also** Command line option **-IDMsearchpath**
DMF_SetUsepathModifier	*0*	This action controls the converter that turns Use Paths into file paths. The control happens through a string as *data* parameter. **Value range** » *""* – empty string » *"L"* – conversion to lower case » *"F"* – conversion of the first letter in each path segment to lower case » *"U"* – conversion to upper case » *"u"* – conversion to upper case except for the file extension
DMF_SetCodePage		With this action the code page for the transfer of strings between application and IDM can be set. Usually, IDM expects and returns strings that are encoded according to the ISO 8859-1 standard. With this action a different character encoding can be defined. As of IDM version A.06.01.d, it is possible to specify an ***Application*** object in the *objectID* parameter. This changes the application-specific code page that is required for processing strings. The change of an application-specific code page within one of the functions of the corresponding application has an immediate effect. However, the call on a DDM server side does not support changing the application code page, but only affects the network application anyway.

action	options	Meaning
DMF_SetFormatCodePage		Defines the code page in which format functions interpret and return strings.
The options below apply to DMF_SetCodePage and DMF_SetFormatCodePage		
	CP_ascii	ASCII character encoding.
	CP_iso8859	Western European Latin-1 encoding according to ISO 8859-1.
	CP_cp437	English character encoding according IBM code page 437 (MS-DOS).
	CP_cp850	Western European character encoding according to IBM code page 850 (MS-DOS).
	CP_iso6937	Western European character encoding with variable length according to ISO 6937.
	CP_winansi	MICROSOFT WINDOWS character encoding.
	CP_dec169	Character encoding according to DEC code page 169.
	CP_roman8	8-bit character encoding according to HP code page Roman-8.
	CP_utf8	8-bit Unicode encoding with variable length, corresponds to ASCII encoding in the range *0 – 127*.
	CP_utf16 CP_utf16b CP_utf16l	16-bit Unicode encoding with character widths from 2 up to 4 bytes. There are two variants: » BE – big-endian, bytes with higher numerical significance first. » LE – little-endian, bytes with lower numerical significance first. UTF-16 without a specified byte order corresponds to the LE variant on Microsoft Windows and to the BE variant on Unix systems.
	CP_cp1252	Western European character encoding according to Microsoft Windows code page 1252.

action	options	Meaning
	CP_acp	Currently used ANSI code page of an application on Microsoft Windows. Can only be used on Microsoft Windows.
	CP_hp15	Western European 16-bit character encoding used by HP systems.
	CP_jap15	Japanese 16-bit character encoding used by HP systems.
	CP_roc15	Simplified Chinese 16-bit character encoding used by HP systems.
	CP_prc15	Traditional Chinese 16-bit character encoding used by HP systems.
	CP_ucp	Custom code page ("User Code Page"). Convert to any code page with **iconv()**.

Remarks

» Switching to a code page is valid until another code page is set. All strings have to be transferred in this code page and all strings that IDM transfers to the application are converted into this code page.

» The function **DM_Control** must not be called from a canvas callback function.

Example

Setting the application code page to “Roman 8”.

```
DM_TraceMessage ("This application will use roman-8 codepage",  0);
DM_Control (0, DMF_SetCodePage, CP_roman8);
```

See also

Function DM_ControlEx

3.11 DM_ControlEx

This function can be used to change general settings in the ISA DIALOG MANAGER or to trigger actions.

Compared to the function DM_Control, this function provides the additional action *DMF_SetUserCodePage*. Using *DMF_SetUserCodePage* a code page system string can be set for the **iconv** conversion which is only active if an IDM code page is set to **CP_ucp**. A setting to **CP_ucp** is done e.g. by the action *DMF_SetCodePage*. When using this code page, a code page conversion of characters/strings is performed by means of the **iconv** routines available on UNIX/LINUX. The user can freely choose the codepage. Since the iconv functionality is **not available** on MS-Windows or VMS this is not supported on these platforms (default conversion to ?-characters).
From/to which codepage is actually converted can be determined by the user. It is only necessary that iconv supports the conversion from/to **UTF-8**! By default the user codepage "8859-1" is set. By means of the action DMF_SetUserCodePage the application programmer can set the code page in which his strings are.

```
DM_boolean DML_default DM_EXPORT DM_ControlEx
(
  DM_ID      objectID,
  DM_UInt    action,
  DM_Pointer data,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Object for which the specified action shall be performed.

-> DM_UInt action

Action to be performed by the IDM. For this purpose, several constants are defined in the include file **IDMuser.h**. These constants are explained in the table below.

-> DM_Pointer data

Parameter for passing arbitrary data to the function.

For the action *DMF_SetUserCodePage*, a pointer to a string containing code page code must be given in this parameter.

-> DM_Options options

Contains an argument for the *action*, if required (see table below).

Return value

DM_TRUE	The action has beeen executed successfully.
DM_FALSE	The action could not be executed.

The following table shows the valid assignments of the individual parameters and explains their meaning. When nothing else is stated with the action, the *objectID* has to be *0*.

action	options	Meaning
DMF_UpdateScreen	*0*	All internal SetVal calls shall be displayed on the screen. In this case, the first parameter has to be assigned with the dialog.
DMF_UIAutomationMode		With this action the specific UI Automation support of the IDM for its specific objects can be disabled. However, the UI Automation support provided by Microsoft for the standard controls remains active. UI Automation support is active by default. The switching must happen before calling **DM_Initialize()** and after bootstrapping.
	0	Disables the UI Automation support of the IDM.
	1	Enables the UI Automation support of the IDM.
DMF_PCREBinding	*0*	Disables the linking to the PCRE library, thus Regular Expressions are no longer possible.
	1	Linking to statically present PCRE functions in the executable (linking type E).
	2	Only dynamic linking of PCRE libraries relative to the application (linking type A).
	3	Linking to PCRE libraries relative to the application or from the system (linking sequence A – S).
	4	Linking with priority for functions in the executable (linking sequence E – A – S), this is the standard for self-built IDM applications.
	5	Linking analog to *4* but in reverse order, i.e. precedence for the PCRE library installed in the system (linking sequence S – A – E).
	See also Chapter “PCRE Library for Support of Regular Expressions” at the built-in function regex	
DMF_SignalMode	*0*	The signals are intercepted by the function **signal**.
	1	The signals are intercepted by the function **sigaction**.

action	options	Meaning
DMF_SetSearchPath	*0*	This action sets the search path for IDM files (dialog, module, interface, and binary files). The semicolon-separated directories have to be passed as string pointers in the *data* parameter. **See also** Command line option **-IDMsearchpath**
DMF_SetUsepathModifier	*0*	This action controls the converter that turns Use Paths into file paths. The control happens through a string as *data* parameter. **Value range** » *""* – empty string » *"L"* – conversion to lower case » *"F"* – conversion of the first letter in each path segment to lower case » *"U"* – conversion to upper case » *"u"* – conversion to upper case except for the file extension
DMF_SetCodePage		With this action the code page for the transfer of strings between application and IDM can be set. Usually, IDM expects and returns strings that are encoded according to the ISO 8859-1 standard. With this action a different character encoding can be defined. As of IDM version A.06.01.d, it is possible to specify an ***Application*** object in the *objectID* parameter. This changes the application-specific code page that is required for processing strings. The change of an application-specific code page within one of the functions of the corresponding application has an immediate effect. However, the call on a DDM server side does not support changing the application code page, but only affects the network application anyway.

action	options	Meaning
DMF_SetFormatCodePage		Defines the code page in which format functions interpret and return strings.
DMF_SetUserCodePage		Sets the character code for **iconv**, and thus indirectly influences the IDM code page **CP_ucp**, which is activated by **DMF_SetCodePage**. (Only on platforms that support iconv). The code page code is passed in the *data* parameter (pointer to a string containing code page code).
The options below apply to DMF_SetCodePage and DMF_SetFormatCodePage		
	CP_ascii	ASCII character encoding.
	CP_iso8859	Western European Latin-1 encoding according to ISO 8859-1.
	CP_cp437	English character encoding according IBM code page 437 (MS-DOS).
	CP_cp850	Western European character encoding according to IBM code page 850 (MS-DOS).
	CP_iso6937	Western European character encoding with variable length according to ISO 6937.
	CP_winansi	MICROSOFT WINDOWS character encoding.
	CP_dec169	Character encoding according to DEC code page 169.
	CP_roman8	8-bit character encoding according to HP code page Roman-8.
	CP_utf8	8-bit Unicode encoding with variable length, corresponds to ASCII encoding in the range *0 – 127*.

action	options	Meaning
	CP_utf16 CP_utf16b CP_utf16l	16-bit Unicode encoding with character widths from 2 up to 4 bytes. There are two variants: » BE – big-endian, bytes with higher numerical significance first. » LE – little-endian, bytes with lower numerical significance first. UTF-16 without a specified byte order corresponds to the LE variant on Microsoft Windows and to the BE variant on Unix systems.
	CP_cp1252	Western European character encoding according to Microsoft Windows code page 1252.
	CP_acp	Currently used ANSI code page of an application on Microsoft Windows. Can only be used on Microsoft Windows.
	CP_hp15	Western European 16-bit character encoding used by HP systems.
	CP_jap15	Japanese 16-bit character encoding used by HP systems.
	CP_roc15	Simplified Chinese 16-bit character encoding used by HP systems.
	CP_prc15	Traditional Chinese 16-bit character encoding used by HP systems.
	CP_ucp	Custom code page ("User Code Page"). Convert to any code page with **iconv()**.

Remarks

» Switching to a code page is valid until another code page is set. All strings have to be transferred in this code page and all strings that IDM transfers to the application are converted into this code page.

» The function **DM_ControlEx** must not be called from a canvas callback function.

Example

Setting the application code page using CP_ucp and DMF_SetUserCodePage.

```
/* Set application code page to CP_ucp and use CP1250. */
...
DM_Control((DM_ID)0, DMF_SetCodePage, CP_ucp);
DM_ControlEx((DM_ID)0, DMF_SetUserCodePage, "CP1250", 0)
...
```

See also

Function DM_Control

3.12 DM_CreateObject

Any object or model within a dialog can be created with this function. The first parameter describes the type of the object (pushbutton, window, etc.), the second parameter describes the type of the object parent, and the last parameter describes the scope (object/model) of the new object.

```
DM_ID DML_default DM_EXPORT DM_CreateObject
(
  DM_ID classtagOrModel,
  DM_ID parentID,
  DM_Options options
)
```

Parameters

-> DM_ID classtagOrModel

This parameter describes the type of the new object. The necessary definitions are included in IDMuser.h.

The following definitions are accepted:

- DM_ClassCanvas
- DM_ClassCheck
- DM_ClassEditext
- DM_ClassGroupbox
- DM_ClassImage
- DM_ClassImport
- DM_ClassListbox
- DM_ClassMenubox
- DM_ClassMenuitem
- DM_ClassMenusep
- DM_ClassMessagebox
- DM_ClassModule
- DM_ClassNotebook
- DM_ClassNotepage
- DM_ClassPoptext
- DM_ClassPush
- DM_ClassRadio
- DM_ClassRecord
- DM_ClassRect
- DM_ClassScroll

- DM_ClassStatext
- DM_ClassTablefield
- DM_ClassTimer
- DM_ClassWindow

-> DM_ID parentID

Parent of the object to be newly generated.

-> DM_Options options

For these functions different options are possible which can be indicated in connection with an "or" (|).

Option	Meaning
DMF_CreateModel	If this option is set, a new model is to be created by means of this function.
DMF_CreateInvisible	If this option is set, the object to be newly generated is to be created independently of the definition in the specified model.
DMF_InheritFromModel	This option has to be set if a DM_ID of a model is indicated in the parameter classtagOrModel. In doing so, the function will know that a new instance has to be generated from this model. If it is a hierachical model, also its children will be created in the new object.

Return Value

ID!=0	Object was successfully created.
ID=0	Object could not be created.

Example

A new instance of a window model is to be generated out of a C function.

```
void DML_default DM_ENTRY CreateNewInstance
(
    DM_ID Modell,
    DM_ID DialogID
)
{
  DM_ID NewObject;
  DM_Value data;

  if ((NewObject = DM_CreateObject (Modell, DialogID,
    DMF_InheritFromModel | DMF_CreateInvisible) != (DM_ID) 0)
```

```
  {
    /* Setting the data in a newly generated window */

    /* Finally making windows invisible */
    data.type = DT_boolean;
    data.value.boolean = TRUE;
    DM_SetValue(NewObject, AT_visible, 0, &data, 0);
  }
}
```

See Also

Built-in function create() in manual “Rule Language”

Method :create()

3.13 DM_DataChanged

This function is used to signal that the value of the specified attribute (Model attribute) has changed on a particular Data Model (Model component). This change is put into event processing as a *datachanged* event. Not until further event processing the linked presentation objects (View components) are triggered to fetch the new data values and have them displayed.

```
DM_Boolean DML_default DM_EXPORT DM_DataChanged
(
  DM_ID        object,
  DM_Attribute attribute,
  DM_Value  *  index,
  DM_Options   options
)
```

Parameters

-> DM_ID object

This is the object ID of the Data Model object where a data value has changed.

-> DM_Attribute attribute

This parameter indicates the attribute of the Data Model that has changed.

-> DM_Value *index

This parameter defines the index of the modified attribute. If it is *NULL*, this indicates a change of all single values.

-> DM_Options options

Option	Meaning
0	There will be no tracing of this function.
DMF_Verbose	Activates tracing of this function.

Return value

DM_TRUE Modification has been successfully stored as *datachanged* event.

DM_FALSE No event could be created.

Example

Dialog File

```
dialog D
function datafunc FuncData();
function void     Reverse(integer Idx);

window Wi
{
  .title "Datafunc demo";
  .width 200;  .height 300;

  edittext Et
  {
    .datamodel FuncData;
    .dataget .text;
    .dataset .text;
    .xauto 0;
    .xright 80;
  }

  pushbutton PbAdd
  {
    .text "Add";
    .xauto -1;
    .width 80;

    on select
    {
      this.window:apply();
    }
  }

  listbox Lb
```

```
  {
    .xauto 0;  .yauto 0;
    .ytop 30;  .ybottom 30;
    .datamodel FuncData;
    .dataget .content;

    on select
    {
      PbReverse.sensitive := true;
    }
  }

  pushbutton PbReverse
  {
    .xauto 0;  .yauto -1;
    .text "Reverse element";
    .sensitive false;

    on select
    {
      Reverse(Lb.activeitem);
    }
  }

  on close { exit(); }
}
```

C File

```
#ifdef VMS
#  define EXITOK    1
#  define EXITERROR 0
#else
#  define EXITOK    0
#  define EXITERROR 1
#endif

#include <IDMuser.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "datafuncfm.h"

#define DIALOGFILE "~:datafunc.dlg"

static DM_ID     data_id = (DM_ID)0;
static DM_Value  data_vec;
```

```
static DM_String data_str;

void DML_default DM_CALLBACK FuncData __1((DM_DataArgs *, args))
{
  if (!data_id)
    data_id = args->object;
  switch (args->task)
  {
  case MT_get:
    switch(args->attribute)
    {
    case AT_text:
      args->retval.type = DT_string;
      args->retval.value.string = data_str;
      break;
    case AT_content:
      if (args->index.type == DT_void)
      {
        args->retval = data_vec;
      }
      else if (args->index.type == DT_integer)
      {
        DM_ValueGet(&data_vec, &args->index, &args->retval, 0);
      }
      break;
    default:
      break;
    }
    break;
  case MT_set:
    switch(args->attribute)
    {
    case AT_text:
      if (args->data.type == DT_string)
      {
        if (DM_ValueChange(&data_vec, NULL, &args->data, DMF_AppendValue))
          DM_DataChanged(data_id, AT_content, NULL, DMF_Verbose);
      }
      break;
    default:
      break;
    }
    break;
  default:
    break;
  }
}
```

```
void DML_default DM_ENTRY Reverse __1((DM_Integer, Idx))
{
  DM_Value  index, data;
  char     *cp,    ch;
  size_t    len,   i;

  DM_ValueInit(&data, DT_void, NULL, 0);
  index.type = DT_integer;
  index.value.integer = Idx;
  if (DM_ValueGet(&data_vec, &index, &data, 0) && data.type == DT_string)
  {
    if (DM_StringChange(&data_str, data.value.string, 0))
      DM_DataChanged(data_id, AT_text, NULL, DMF_Verbose);

    /* reverse the string */
    cp = data.value.string;
    if (cp)
    {
      len = strlen(cp);
      if (len>2)
      {
        len--;
        for (i=0; len>0 && i<len/2; i++)
        {
          ch = cp[i];
          cp[i] = cp[len-i];
          cp[len-i] = ch;
        }
      }
    }
    if (DM_ValueChange(&data_vec, &index, &data, 0) && data_id)
      DM_DataChanged(data_id, AT_content, &index, DMF_Verbose);
  }
}

int DML_c AppMain __2((int, argc), (char **,argv))
{
  DM_ID dialogID;
  DM_Value data;

  /* initialize the Dialog Manager */
  if (!DM_Initialize (&argc, argv, 0))
  {
    DM_TraceMessage("Could not initialize.", 0);
    return (1);
  }
```

```
  /* load the dialog file */
  switch(argc)
  {
  case 1:
    dialogID = DM_LoadDialog (DIALOGFILE,0);
    break;
  case 2:
    dialogID = DM_LoadDialog (argv[1],0);
    break;
  default:
    DM_TraceMessage("Too many arguments.", 0);
    return(EXITERROR);
    break;
  }
  if (!dialogID)
  {
    DM_TraceMessage("Could not load dialog.", 0);
    return(EXITERROR);
  }

  data.type = DT_type;
  data.value.type = DT_string;
  DM_ValueInit(&data_vec, DT_vector, &data, DMF_StaticValue);

  data.type = DT_string;
  data.value.string = "^ Enter a string";
  DM_ValueChange(&data_vec, NULL, &data, DMF_AppendValue);
  data.value.string = "and press 'Add'";
  DM_ValueChange(&data_vec, NULL, &data, DMF_AppendValue);

  DM_StringInit(&data_str, DMF_StaticValue);
  DM_StringChange(&data_str, "Change me!", 0);

  /* install table of application functions */
  if (!BindFunctions_D (dialogID, dialogID, 0))
    DM_TraceMessage ("There are some functions missing.", 0);

  /* start the dialog and enter event loop */
  if (DM_StartDialog (dialogID, 0))
    DM_EventLoop (0);
  else
    return (EXITERROR);

  return (EXITOK);
}
```

3.14 DM_Destroy

Any object or model and their children within the dialog can be deleted with this function. The second parameter controls what exactly is to be deleted.

```
DM_Boolean DML_default DM_EXPORT DM_Destroy
(
  DM_ID objectID,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

The ID of the object to be deleted.

-> DM_Options options

Controls the function behavior while deleting. There are the following possibilities

Option	Meaning
DMF_ForceDestroy	If you specify *DMF_ForceDestroy* as option, the object will be deleted and all rule parts using this object are changed. As a result the corresponding instructions will be removed. If the object to be deleted is a model and the parameter *DMF_ForceDestroy* is not specified here, only the repeated referencing of the model will be prohibited, but it will remain a model. However, if you indicate *DMF_ForceDestroy*, all models used by the objects will be removed, and the objects will take the values of the next higher model or default.

Return Value

TRUE	Object was successfully deleted.
FALSE	Object could not be deleted.

DM_Destroy() invokes the :clean() method of the object to be destroyed.

Example

Destroying an object out of a C function.

```
DM_Boolean DML_default DM_ENTRY DestroyObject
(
  DM_ID ObjID
)
{
  return (DM_Destroy(ObjID, DMF_ForceDestroy));
```

```
}
```

See Also

Built-in function destroy() in manual “Rule Language”

Method :destroy()

3.15 DM_DialogPathToID

Attention

The function **DM_DialogPathToID** is deprecated and is only supported for compatibility reasons. Please use DM_ParsePath instead.

By using this function you can query the identifier of an object if you have loaded more than one dialog and the desired object is not contained in the dialog loaded first.

```
DM_ID DML_default DM_EXPORT DM_DialogPathToID
(
  DM_ID dialogid,
  DM_ID rootid,
  DM_String path
)
```

Parameters

-> DM_ID dialogid

This is the identifier of the dialog in which the object is to be searched.

-> DM_ID rootid

By using this parameter you can control from which Dialog Manager object on the search for the desired object begins. There are the following possibilities:

» *rootid = 0*
The Dialog Manager searches for the specified object in the entire dialog definition.
This is the usual case. You can also query the identifiers of rules, functions, variables and resources.

» *rootid != 0*
The Dialog Manager will search only on the next lower hierarchy level from the specified object on; the DM will not search on lower hierarchy levels.
This procedure is suitable only if an object name appears more than once in a dialog.

-> DM_String path

This path describes the object being searched. The path has to describe an object unambiguously. If the object identifier exists only once in the dialog, it is enough if you specify the name to get the desired reference. If the object identifier is not unambiguous, the object has to be described by a path of object identifiers, separated by a dot.

Return Value

0	The object was not found or its identifier is not unique.
!= 0	Identifier of the searched object.

Annotation

If *"setup"* is specified as *path* and both *dialogid* and *rootid* are *0*, then the ***setup*** object is returned.

Example

```
void DML_default DM_ENTRY OkButtonCallback __1((DM_ID, dialogID))
{
  DM_ID ID1;
  DM_ID ID2;

  /* Querying an object in the dialog globally */
  ID1 = DM_DialogPathToID(dialogID, 0, "FirstObject");

  /* Querying via a path */
  ID2 = DM_DialogPathToID(dialogID, 0, "FirstObject.Child1");
}
```

See Also

Function DM_ParsePath

Built-in function parsepath in manual “Rule Language”

3.16 DM_DispatchHandler

Using the **DM_DispatchHandler** function, a user-defined function (handler) can be registered with the IDM FOR MOTIF, which is called before the **XtDispatchEvent** function. This allows *XEvents* to be processed at the X level (and not just at the X toolkit level).

```
DM_Boolean DML_default DM_EXPORT DM_DispatchHandler
(
  DM_DispatchHandlerProc  funcp,
  DM_UInt     chain_pos,
  DM_UInt     operation,
  DM_Options  options
)
```

Parameter

-> DM_DispatchHandlerProc funcp

Function pointer to the custom handler function. This function receives the *XEvent* as a parameter.

The passed function must be defined as follows:

```
DM_Boolean DML_default DM_EXPORT MyHandler
(
  XEvent * event
)
```

-> DM_UInt chain_pos

This parameter determines whether the handler function is inserted at the beginning (*DMF_InstallHead*) or at the end (*DMF_InstallTail*) of the list of handler functions.

chain_pos is only evaluated if the parameter *operation* is set to *DMF_RegisterHandler*.

-> DM_UInt operation

This parameter defines the actual operation to be performed by the function. The following constants are defined for this:

operation	Meaning
DMF_RegisterHandler	This value is used to install a handler.
DMF_WithdrawHandler	This value uninstalls a previously installed handler.
DMF_DisableHandler	This value temporarily disables a handler.
DMF_EnableHandler	With the help of this value, a disabled handler is reactivated.

-> DM_Options options

If *options* is set to the value *DMF_DontTrace*, the function call is not logged in the trace file.

Return Value

DM_TRUE	The *XEvent* has been completely processed. No further “DispatchHandler” will be called, nor will the **XtDispatchEvent** function be called.
DM_ FALSE	The next registered “DispatchHandler”, or lastly the function **XtDispatchEvent**, is called.

3.17 DM_DumpState

With this function IDM status information is written into the log or trace file.

Syntax

```
void DML_default DM_EXPORT DM_DumpState
(
  DM_Enum state,
  DM_Options options
)
```

Parameter

-> DM_Enum state

This parameter influences which sections of the status information are written out.

The dumpstate is a status information of IDM-relevant information in order to simplify error analysis within an IDM application.

The content of the dumpstate is divided into different sections that are variable and that are adapted to the error situation. In addition, the dumpstate is influenced by the errors that have previously occurred. For example, an unsuccessful memory allocation leads to information concerning the memory usage by the IDM in the next dumpstate output. If no IDM objects or identifiers can be created, then the utilization of IDM objects and identifiers is dumped.

The dumpstate information is always encased between *"*** DUMP STATE BEGIN ***"* and *"*** DUMP STATE END ***"* and can have the following sections, which are described in detail in the paragraphs below:

- PROCESS: Process and thread number, date/time.
- ERRORS: Complete content of the error codes set.
- CALLSTACK: Contains rules, DM interface functions and application functions directly called by the IDM.
- THISEVENTS and EVENT QUEUE: Currently processed thisevent objects and their values as well as events that are still in the queue.
- USAGE: The number of created objects, modules and identifiers and the size of the memory that is used by the rule interpreter and for string transfer.
- MEMORY: Memory usage as far as it can be detected by the IDM.
- SLOTS: Hints about IDM objects that have not been correctly released.
- VISIBLE OBJECTS: A list of the visible objects and their respective values.

In order to keep the output to a minimum, this is usually displayed in a shortened form. Generally, IDM strings (in "…") are always shortened to a maximum of 40 characters. Their entire length is attached in [ ]. Byte size information is given in kilo, mega or gigabytes (k/m/g).

Value Range

Value (enum)	Meaning
DM_DUMP_all	All sections are written out in an abbreviated form. This corresponds to the output in case of a FATAL ERROR.
DM_DUMP_error	The sections ERRORS, CALLSTACK and EVENTS are written out in an abbreviated form. This is the normal output in the case of EVAL ERRORS.
DM_DUMP_events	The sections THISEVENTS and EVENT QUEUE are written out in full.
DM_DUMP_full	All sections are written out in full.
DM_DUMP_locked	The section SLOTS is written out in full. In addition, for locked objects their attribute values are written out.
DM_DUMP_memory	The section MEMORY is written out in full.
DM_DUMP_none	No action (nothing is written out).
DM_DUMP_process	The section PROCESS is written out in full.
DM_DUMP_short	All sections (excluding SLOTS) are written out in an abbreviated form.
DM_DUMP_slots	The section SLOTS is written out in full.
DM_DUMP_stack	The section CALLSTACK is written out in full.
DM_DUMP_usage	The section USAGE is written out in full.
DM_DUMP_uservisible	The section VISIBLE OBJECTS is written out in full for all visible top-level objects including their children, the pre-defined and user-defined attributes.
DM_DUMP_visible	The section VISIBLE OBJECTS is completely written out.

Combinations out of multiple sections are not possible.

-> DM_Options options

This parameter is reserved for future versions. At present pass only *0*.

The output of the dumpstate also can be triggered with the built-in function dumpstate, as well as through the command line options **-IDMdumpstate** and **-IDMdumpstateseverity <string>**.

Availability

IDM versions A.05.01.g3, A.05.01.h, as well as A.05.02.e and above

See Also

Chapter “Dumpstate (Status Information)” in manual “Development Environment”

3.18 DM_ErrMsgText

This function returns the errorstring belonging to an errorcode.

```
DM_String DML_default DM_EXPORT DM_ErrMsgText
(
  DM_ErrorCode eno,
  DM_Options options
)
```

Parameters

-> DM_ErrorCode eno

Error code to which the error message is to be returned.

-> DM_Options options

Via the various options you can control which information is to be integrated in the error text. To do so, you have the following possibilities which can be specified with "or" in combination.

Option	Meaning
DMF_IncludeIdent	The name of the part which has produced the error is to be included in the error message. This can be the operating system, the window system or DM.
DMF_IncludeModule	The name of the module in which the error has occurred is to be included in the error message.
DMF_IncludeSeverity	The severity of the error (warning, error, fatal error) is to be included in the message text.
DMF_IncludeText	The actual error text is to be included in the error message.

Example

```
/* error handling function for dialog manager errors*/

static void QueryError ( )
{
        /*buffer for the errors occurred*/
    DM_ErrorCode errorbuffer[32];
    register int i;
        /*number of errors*/
    int errors;

    if ((errors = DM_QueryError(errorbuffer, 32, 0)))
        for (i = 0, i < errors; i++)
            DM_TraceMessage(DM_ErrMsgText(errorbuffer[i],
                0), 0);
```

```
}
```

3.19 DM_ErrorHandler

This function allows for handler functions to be set up that are then called when an error, which is recognized by the rule interpreter, occurs.

```
DM_Boolean DML_default DM_EXPORT DM_ErrorHandler
(
  DM_ErrorHandlerProc  funcp,
  DM_UInt     operation,
  DM_Options  options
)
```

Parameter

-> DM_ErrorHandlerProc funcp

This is a pointer to the function that shall be installed as an error handler.

-> DM_UInt operation

This parameter defines the action that shall be carried out. The following values are possible:

DMF_RegisterHandler	Handler function shall be registered
DMF_EnableHandler	Handler function shall be enabled
DMF_DisableHandler	Handler function shall be disabled
DMF_WithdrawHandler	Handler function shall be withdrawn

-> DM_Options options

Currently unused, must be *0*.

Return Value

DM_TRUE	Action was successful.
DM_FALSE	Action could not be carried out. This may happen when the action would cause the same handler to exist twice, when the function pointer is *NULL* or when the handler that the action addresses cannot be found

A non-intercepted error in the rule code of the application (i.e. faulty parameter types, dynamic access to a non-existing attribute or relative child, each of them not caught by `fail()`) is noted by the rule interpreter, as *"EVAL ERROR"* in the log file or the trace file. Messages in "[]" that begin with "W:", "E:" or "I:" such as a module or interface loading errors do not belong to the rule interpreter and are therefore not passed on to the error handler.

The error handler serves as a way of informing the IDM applications in advance of these application errors. Numerous, not identical, error handlers are possible, which are invoked in reverse order of registration. Only the activated error handlers are called and receive the same information structure (**DM_ErrorInfo)** as parameter. These calls are logged in the trace file. Recursive calls of error handlers are suppressed. When the **DM_ErrorHandler** function is called within an error handler it is not allowed to install a new or uninstall an existing handler.

Bear in mind that the error handler is synchronously called and insofar the design and coding must take place with great care and under consideration of the special constellations and restrictions. Calling the handler before starting the dialog is possible (e.g. when there are errors in **:init()** methods) as is by improper use in an unsuitable run state (i.e. forced loading of a faulty rule via **DM_CallRule** within a format function). Avoiding such constellations is recommended. Furthermore, when the IDM is used to display an error box then an independent error dialog, which is started before the application, should be used.

The error handler function to be installed must be defined as follows:

```
typedef void (DML_default DM_CALLBACK *DM_ErrorHandlerProc) __((DM_ErrorInfo
*info));
```

The transferred information structure provides information about the error:

```
define  EITK_rule_engine  1

typedef struct {
  /* user info, not changeable */
  DM_UInt1      task;     /* task/component which detects the error */
  DM_ErrorCode  errcode;  /* error code */
  DM_ID         object;   /* this object or null-ID */
  DM_ID         rule;     /* rule where error occured */
  DM_String     file;     /* module/dialgo filename or NULL */
  DM_String     message;  /* error message or NULL*/
} DM_ErrorInfo;
```

Under no circumstance this information must be changed by any error handler.

Principally, strings are encoded using the application code page.

The *task* entry reveals information about the IDM component that reports the error. Due to possible future changes it should always be checked. Currently, only errors from the rule interpreter are passed on as *EITK_rule_engine*.

The entries reveal information about the error code, about the rule in which the error occurred as well as the error text that belongs to it. When possible, the `this` object and the file name of the module or dialog, in which the faulty rule can be found, are made available.

Example

```
#include <IDMuser.h>
void DML_default DM_CALLBACK ErrorHandler __1((DM_ErrorInfo *, info))
```

```
{
  if (info->task == EITK_rule_engine)
  {
    DM_TraceMessage("ERROR: errcode=%d message=\"%s\" rule=\"$I\"",
      DMF_LogFile|DMF_Printf,
      info->errcode, info->message,
      info->rule);
  }
}
int DML_c DM_CALLBACK AppMain (int argc, char **argv)
{
  DM_ID dialogID = (DM_ID)0;
  if (DM_Initialize (&argc, argv, 0) == FALSE)
    return (1);
  if (argc>1)
  {
    if (!(dialogID = DM_LoadDialog (argv[1], 0)))
      return (1);
    DM_ErrorHandler(ErrorHandler, DMF_RegisterHandler, 0);
    DM_StartDialog (dialogID, 0);
    DM_EventLoop (0);
    return (0);
  }
  return 1;
}
```

3.20 DM_EventLoop

This function starts the processing of the dialog. It enables the user to work with the dialog. After this function call events are processed only by DM, because this function takes over the handling of the dialog. The following steps are necessary:

```
DM_Boolean DML_default DM_EXPORT DM_EventLoop
(
  DM_Options options
)
```

Parameters

-> DM_Options options

By using this parameter you can control whether the event processing is to be interrupted if no event exists any more or if the processing is to be executed until the end of the dialog.

Option	Meaning
0	This is the normal case. Dialog Manager is responsible for the processing of all events and is to exit this function only at the end of the program.
DMF_DontWait	By using this option Dialog Manager will be informed that it will not go in a passive waiting loop, if no event exists. It is to return immediately to the application so that the application can execute any actions. Then the application has to call the function DM_EventLoop again for further processing.
DMF_WaitForEvent	By using this constant DM will be informed that it can wait passively for a user event in the window system. After having processed it, DM is to return to the application so that any actions can be performed and the function DM_EventLoop can be called again.

Return Value

TRUE	Processing of event was cancelled because no further event was scheduled to be processed at the moment.
FALSE	Processing of event was cancelled because the end of the dialog has been reached.

Example

Passing over the processing to DM in the main program:

```
int DML_c DM_CALLBACK AppMain __2(
(int, argc),
```

```
(char far * far *, argv))
{
  DM_ID dialogID;
  static char * dialogfile = "format.dlg";

  DM_Initialize(&argc, argv, 0);

  dialogID = DM_LoadDialog (dialogfile, 0);

  if (!dialogID)
    return(1);

  (void) DM_BindCallBacks(funcmap, NFUNCS, dialogID, 0);

  DM_StartDialog(dialogID, 0);

  DM_EventLoop(0);

  return (0);
}
```

3.21 DM_ExceptionHandler

With this function a handler can be installed. This handler can capture possible “asserts” in Dialog Manager and output an appropriate message to the user. Furthermore databases and files can be closed and thus leave DM in a defined state.

```
DM_Boolean DML_default DM_EXPORT DM_ExceptionHandler
(
  DM_ExceptionHandlerProc funcp,
  DM_UInt operation,
  DM_Options options
)
```

Parameter

-> DM_ExceptionHandlerProc funcp

This is a pointer on the function to be installed as Exception-Handler.

-> DM_Uint operation

This parameter is used to specify the action to be carried out. The following values are valid:

DMF_RegisterHandler	handler function is to be installed
DMF_WithdrawHandler	handler function is to be quit
DMF_EnableHandler	handler function is to be recalled
DMF_DisableHandler	handler function is not to be recalled any more

-> DM_Options options

Currently not used, please specify with 0.

Return Value

TRUE	handler function has been installed successfully
FALSE	handler function has not been installed

The call is logged in the logfile. The first call will be traced. The following recursive calls won't be written into the tracefile, because they might cause endless loops. The ExceptionHandler is no longer called in case of an assert in the assfail function. The trace/logfile contains file, line and assertion. The end of the call will also be logged (with the parameter DM_ExceptionInfo).

If several functions are active, these will be called one after the other. The order will be the other way round to the order of registering. The functions work on the same **DM_ExceptionInfo**, i.e. if a function sets a *message* it can be overwritten by another function. The same applies to *showmessage*.

The function to be installed must be defined as follows:

```
typedef void (DML_default DM_CALLBACK *DM_ExceptionHandlerProc) __((DM_
ExceptionInfo *info))
```

```
typedef DM_ExceptionInfo
{
  // user info, not changeable
  DM_String   file;
  DM_Integer  line;
  DM_String   assertion;
  // output, changeable
  DM_String   message;  // default (char *)0
  DM_Boolean  showMessage;  // default TRUE
} DM_ExceptionInfo;
```

The items of the structure correspond to those of the assertion. You cannot change these values.

message is passed on with *NULL* pointer. Instead of using the message *"Contact your local…"* you may use any other message. If the *message* is *NULL* pointer, the standard message will be output.

showMessage defines whether a message is to be output.

3.22 DM_Execute

This function is used to start another program from the dialog script. Depending on the used operation system different types of programs to be started are supported.

```
DM_Boolean DML_default DM_EXPORT DM_Execute
(
  DM_String command,
  DM_String arguments,
  DM_Boolean synchronous,
  DM_Enum exetype,
  DM_Enum windowtype,
  DM_ID object,
  DM_Value * event,
  DM_Value * replydata,
  DM_Options options
)
```

The parameters correspond to the parameters for the built-in function execute (see built-in function execute in manual “Rule Language”). In the Rule Language not all the parameters have to be specified. In C, however, all parameters must have a value.

The additional parameter options is currently not used and must be specified with 0.

3.23 DM_FatalAppError

This function should be called if the application has found an error and is unable to continue. This function finishes the DM, closes all files and displays, and then returns the control, if required, to the application.

```
void DML_default DM_EXPORT DM_FatalAppError
(
  DM_String reason,
  DM_Int reaction,
  DM_Options options
)
```

Parameters

-> DM_String reason

Message which should be written into the tracefile. This is the cause why the application finishes.

-> DM_Int reaction

This is the action that should be carried out by the DM. The following values are valid:

-1	DM calls the function **abort()** to write a core dump.
0	DM closes the display and all open files, and returns to the application.
> 0	DM closes the display and all open files, and exits with these values.

Warning

When calling the function DM_FatalAppError with *reaction = 0*, you should make sure that the control will not return to Dialog Manager and that no DM function will be called any more.

-> DM_Options options

Currently not used. Please specify with 0.

Example

In the actual DM main program a DM_FatalAppError is called in the file **startup.c**, if the main program has been called for the second time within one process.

```
int cdecl main __2(
(int, argc),
(char far * far *, argv))
{
  register int status;
  static char running = 0;

  if ((status = running++) == 0)
  {
    if ((status = DM_BootStrap(&argc, &argv)) == 0)
```

```
    {
      DM_InitOptions(&argc, argv, 0);

      status = AppMain (argc, argv);
      DM_ShutDown();
    }
    else
      DM_TraceMessage ("Bootstrap failed", DMF_LogFile);
  }
    else
      DM_FatalAppError ("Unexpected restart", -1, 0);
  return (status);
}
```

3.24 DM_FmtDefaultProc

This function takes on all tasks when processing an edittext, e.g. setting a format, the input control, the navigation, etc. as soon as a format is set for the editable text.

Usually, the Dialog Manager does then automatically call this default format function without involving the actual application.

If, however, the edittext has its own format function, Dialog Manager will call this application-specific function instead of the default format function. This function then has to perform the tasks demanded by Dialog Manager. To do so, it can call the function DM_FmtDefaultProc by various tasks, if these are not to be changed application-specifically.

```
DM_boolean DML_default DM_EXPORT DM_FmtDefaultProc
(
  DM_FmtRequest *req,
  DM_FmtFormat *fmt,
  DM_FmtFormatDef **fmtDef,
  DM_FmtContent *cont,
  DM_FmtContentDef **contDef,
  DM_FmtDisplay *dpy
)
```

Parameters

-> DM_FmtRequest *req

This parameter defines the demand for the formatting routine. To do so, a structure element is assigned to the task and - according to the task - a union is filled with the necessary data (e.g. with the new contents as string when setting a new contents).

<-> DM_FmtFormat *fmt

This structure includes description data about the format string which are needed independent of the format function and the chosen kind of format.

<-> DM_FmtFormatDef **fmtDef

With this parameter, a structure is transferred which includes the format-specific data for a format string. If a foreign format function is to use the default format, it has to provide an entry for this in its private data and pass it on there.

<-> DM_FmtContent *cont

This structure includes description data about the contents string to be formatted which are needed independent of the format function and the chosen kind of format

<-> DM_FmtContentDef **contDef

With this parameter, a structure is passed on which includes the format-specific data for a contents string. If a foreign format function is to fall back to the default formats, it has to provide an entry for this in its private data and pass it there.

<-> DM_FmtDisplay *dpy

This structure includes description data about the display string to be formatted which are needed independently of the format function and the chosen kind of format.

Example

Realization of a format function which, itself, assumes only few tasks. The function makes it possible for the user to input a date.

```
/*
** The number to be inputted depends on the cursor position
** so that a valid date results.
*/
char GetMax __1(
(DM_FmtDisplay far *, dpy))
{
  char max;

  switch (dpy->curpos)
  {
    case 0:
      max = '3';
    break;
    case 1:
      if (dpy->string[0] == '3')
        max = '1';
      else
        max = '9';
    break;
    case 3:
      max = '1';
    break;
    case 4:
      if (dpy->string[3] == '1')
        max = '2';
      else
        max = '9';
    break;
    default:
    max = '9';
    break;
  }
  return (max);
}

/*
** actual format function
```

```
**
*/
DM_Boolean DML_c DM_CALLBACK My_Formatter __6(
(DM_FmtRequest far *, req),
(DM_FmtFormat far *, fmt),
(FPTR *, fmtPriv),
(DM_FmtContent far *, cont),
(FPTR *, contPriv),
(DM_FmtDisplay far *, dpy))
{
  DM_Boolean retval;

  switch (req->task)
  {
    /*
    ** The date shall be inputable only in the format
    ** dd.mm.yy. This function is realized only rudimentarily.
    ** BackSpace and DEL shall also be intercepted.
*/
    case FMTK_modify:
    {
      char max = GetMax(dpy);

      if ((!req->targs.modify.strlength
      && (dpy->curpos == dpy->length))
      || ((req->targs.modify.strlength == 1)
        && (req->targs.modify.string[0] >= '0')
        && (req->targs.modify.string[0] <= max)))
      {
        retval = DM_FmtDefaultProc(req, fmt,
            (DM_FmtFormatDef **) (FPTR) fmtPriv, cont,
             (DM_FmtContentDef **) (FPTR) contPriv, dpy);
      }
      else
        retval = FALSE;
    }
    break;

    default:
    /*
    ** Calling the default format function
    */
    retval = DM_FmtDefaultProc(req, fmt, (DM_FmtFormatDef **)
        (FPTR) fmtPriv,cont,
        (DM_FmtContentDef **) (FPTR) contPriv, dpy);
    break;
```

```
  }
  return (retval);
}
```

See Also

Chapter “Structures and Definitions for the Formatting of Input” in manual “C Interface - Basics”

Resource format

Chapter “Format Function” in manual “Rule Language”

3.25 DM_Free

With the help of this function, memory allocated with DM_Malloc or DM_Realloc can be released.

```
void DML_default DM_EXPORT DM_Free
(
  DM_Pointer ptr
)
```

Parameters

-> DM_Pointer ptr

Pointer to the memory which is to be released.

Example

```
char *string;
if ((string = (char *) DM_Malloc (5)))
{
  strcpy (string, "1234");
  ...
  DM_Free (string);
}
```

3.26 DM_FreeContent

This function releases the memory allocated by DM_GetContent. Every time DM_GetContent is called, DM_FreeContent has to follow, after the data has been processed in the application.

```
void DML_default DM_EXPORT DM_FreeContent
(
  DM_Content *content,
  DM_Options options
)
```

Parameters

-> DM_Content *content

Pointer to the contents structure of an object (e.g. listbox) whose memory is to be released. You have received this pointer from DM_GetContent.

-> DM_Options options

Currently not used. Please specify with 0.

Example

Query of the tablefield contents.

```
void DML_default DM_ENTRY GetContent  __1(
(DM_ID, tablefieldID))
{
  int i;
  DM_Integer count;
  DM_Content *vec;
  DM_GetContent(tablefieldID, (DM_Value *) 0, (DM_Value *) 0,
     &vec, &count, 0);
  for (i = 0; i < count; i++)
  {
    printf("vec[%d].sensitive = %d\n",i, vec[i].sensitive);
    printf("vec[%d].string    = %s\n",i, vec[i].string);
  }
  DM_FreeContent(vec, 0);
}
```

3.27 DM_FreeVectorValue

By using this function the memory allocated on calling DM_GetVectorValue can be released again.

```
void DML_default DM_EXPORT DM_FreeVectorValue
(
  DM_VectorValue *values,
  DM_Options options
)
```

Parameters

-> DM_VectorValue *values

In this parameter the vector contained in Dialog Manager is transferred.

-> DM_Options options

Currently not used. Please specify with 0.

Example

Querying the tablefield contents and releasing attribute vector.

```
DM_Boolean DML_default DM_ENTRY ReplaceName__3(
(DM_ID, table),
(char *, from),
(char *, to))
{
    DM_VectorValue *oldData;
    DM_Value count;
    DM_Value lastidx;
    DM_boolean retval = true;

    if (!DM_GetValue(table, AT_rowcount, 0, &count, 0)
        || (count.type != DT_integer))
        return FALSE;

    lastidx.type = DT_index;
    lastidx.value.index.first = count.value.integer;
    lastidx.value.index.second = 2;

    if DM_GetVectorValue(table, AT_content, (DM_Value *) 0,
      &lastidx, &oldData, 0))
        return FALSE;

    DM_FreeVectorValue(oldData, 0);

    return retval;
}
```

3.28 DM_GetArgv

This internal function converts the arguments of the program call into the internally used code page. It may only be called exactly once before the call of DM_BootStrap (see also the supplied startup.c or IDMuser.h).

This function is necessary for the Unicode support

```
char ** DML_default DM_EXPORT DM_GetArgv
(
  int *argc,
  char ** argv,
  DM_Options options
)
```

Parameters

-> int *argc

In this parameter a pointer to the number of command line arguments is passed.

-> char ** argv

Vector on the argument list.

-> DM_Options options

This parameter is reserved for future versions. Therefore, please enter a 0 here.

Return value

The argument list converts to the application code page used by Dialog Manager.

Remark

The function is only intended for use in **startup.c** and its call may only be made there (see also the supplied **startup.c** module).

3.29 DM_GetContent

With this function you can query the actual object contents (listbox, poptext, tablefield) from the application in one function call. In doing so, you can query the actual state (contents and selected items) by one access.

```
DM_Boolean DML_default DM_EXPORT DM_GetContent
(
  DM_ID object,
  DM_Value *firstindex,
  DM_Value *lastindex,
  DM_Content **contentvec,
  DM_UInt *count,
  DM_Options options
)
```

Parameters

-> DM_ID object

Indicates the object which is to be filled with the new contents.

-> DM_Value *firstindex

Controls which range of the contents is queried by this function. This parameter defines the starting point of the range. On querying the contents of a listbox or a poptext the type in the DM_Value structure is always set to DT_integer and the integer value in the union is assigned the starting value. In a tablefield, however, the type in the DM_Value structure is set to DT_index and the index value in the union is assigned the starting value. Here the row is specified in index.first, the column is specified in index.second.

Note

If this parameter is a *NULL* pointer, the starting point has the following defaults:

listbox	integer = 1
poptext	integer = 1
tablefield	index.first = 1, index.second = 1

-> DM_Value *lastindex

Controls which range of the contents is queried by this function. This parameter then defines the ending point of the range. On querying the contents of a listbox or a poptext the type in the DM_Value structure is always set to DT_integer and the integer value in the union is assigned the starting value. In a tablefield, however, the type in the DM_Value structure is set to DT_index and the index value in the union is assigned the starting value. Here the row is specified in index.first, the column is specified in index.second.

Note

If this parameter is a *NULL* pointer, the ending point has the following defaults:

listbox: integer = object.itemcount

poptext: integer = object.itemcount

tablefield: index.first = object.rowcount, index.second = object.colcount

<- DM_Content **contentvec

The pointer to the structure filled by the DM is returned. This structure must not be changed by the application.

<- DM_UInt *count

DM specifies how many items are indexed and therefore the size of the returned structure contentvec is returned.

-> DM_Options options

This parameter controls which information the Dialog Manager is to return. To do so, you have the following possibilities which may also be combined ("or" in bits).

Option	Meaning
DMF_OmitActive	This option specifies that the attribute *.active* in the vector is not to be provided with values, since the application is not interested in this attribute.
DMF_OmitSensitive	This option specifies that the attribute *.sensitive* in the vector is not to be provided with values.
DMF_OmitStrings	This option specifies that the attribute *.content* in the vector is not to be provided with values, since the application is not interested in this attribute. If the strings are not really needed, setting this option will bring along considerable advantages of performance.
DMF_OmitUserData	This option specifies that the attribute *.userdata* in the vector is not to be provided with values, since the application is not interested in this attribute. If the userdata are not really needed, setting this option will bring along considerable advantages of performance.

Note

The contents vector which you receive in the contentvec parameter is allocated by DM and **has** to be released by means of DM_FreeContent.

Example

Querying rows 2 to 5 in a listbox.

```
void DML_default DM_ENTRY GetContent  __1((DM_ID, lb))
{
  int i;
```

```
  DM_Integer count;
  DM_Value first, last;
  DM_Content *vec;

  first.type = DT_integer;
  first.value.integer = 2;

  last.type = DT_integer;
  last.value.integer  = 5;

  DM_GetContent(lb, &first, &last, &vec, &count, 0);

  for (i = 0; i < count; i++)
  {
    printf("vec[%d].sensitive = %d\n",i, vec[i].sensitive);
    printf("vec[%d].active    = %d\n",i, vec[i].active);
    printf("vec[%d].string    = %s\n",i, vec[i].string);
  }

  DM_FreeContent(vec, 0);
}
```

3.30 DM_GetMultiValue

With this function you can query various attributes of different DM objects in one function call. These functions should thus be used with the distributed DM, since they reduce the network traffic considerably.

```
DM_Boolean DML_default DM_EXPORT DM_GetMultiValue
(
  DM_MultiValue * values,
  DM_UInt count,
  DM_ID dialogID,
  DM_String pathname,
  DM_Options options
)
```

Parameters

<-> DM_MultiValue *values

List of attributes and objects to be queried. If the element in the structure for the object is set to 0, the object described in the parameter path name is taken. The list has to have at least the length given in the parameter "count".

-> DM_UInt count

Specifies the length of the object-attribute vector given in the parameter "values".

-> DM_ID dialogID

This parameter describes the dialog to which the given objects belong. It has to be specified only if the object identifier the attributes of which are to be queried is not known and thus the name of the object is specified in the parameter "pathname".

-> DM_String pathname

Describes the object the attributes of which are to be queried. It is only allocated if the object's internal identifier is not known yet.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	The attributes were queried successfully
FALSE	At least one attribute could not be queried

Example

The following C function is to query coordinates at different objects.

```
void DML_default DM_ENTRY Get __3((DM_ID, o1),
                  (DM_ID, o2),
```

```
                  (DM_ID, o3))
{
  DM_MultiValue val[3];

  /* Setting the relevant object ID */
  val[0].object = o1;
  val[1].object = o2;
  val[2].object = o3;
/* Setting the relevant index type */
  val[0].index.type = DT_void;
  val[1].index.type = DT_void;
  val[2].index.type = DT_void;
/* Setting the relevant attribute */
  val[0].attribute = AT_xleft;
  val[1].attribute = AT_width;
  val[2].attribute = AT_xright;

  DM_GetMultiValue(val, 3, dialogID, (char *)0, 0);

  if (val[0].data.type == DT_integer)
    printf("%d %d %d\n",
        (int) val[0].data.value.integer,
        (int) val[1].data.value.integer,
        (int) val[2].data.value.integer);
}
```

See Also

“Object Reference” for the attributes permitted for the relevant object type

3.31 DM_GetToolkitData

You can query window system-specific data by using this function.

```
FPTR DML_default DM_EXPORT DM_GetToolkitData
(
  DM_ID objectID,
  DM_Attribute attr
)
```

Parameters

-> DM_ID objectID

Identifier of the object whose window system-specific data is to be queried.

-> DM_Attribute attr

This parameter defines which window system attribute is to be queried.

The valid assignments of the parameter *attr* depend on the window system and can be taken from the following chapters.

Return value

Depending on the type of the queried value, this function returns the corresponding values converted to *FPTR*.

Remarks

The function **DM_GetToolkitData** does not exist for the server side and should not be called there!

3.31.1 Motif

Using this function you can query the data necessary for X-Windows, such as "window-id", "widget" and "color". The meanings of these datatypes are explained in the corresponding X-Windows manual.

The following values are permitted for the atttributes:

attribute	Meaning
AT_CanvasData	Returns the user-specific data of a canvas. This data have been set by a canvas callback function and contains any user-specific data (see also chapter „Strukturen für Canvas-Funktionen“ in the „C Interface - Basics“ manual).
AT_IsNull	Returns a value<>*0* if the font is a *UI_NULL_FONT*.
AT_Tile	See also AT_XTile

attribute	Meaning
AT_XColor	Returns the X Windows-specific structure for the specified color. The return value of the function is of the type “pixel”. The specified object must be a color.
AT_XColormap	Dieser Wert liefert die Colormap des Default-Screens zurück. Returns a colormap of the default screen.
AT_XCursor	Returns the X Windows-specific structure for the specified cursor. The return value of the function is of the type “Cursor”. The specified object must be a cursor.
AT_XDepth	Dieser Wert liefert die Farbtiefe des Default-Screens als *int*. Returns the color depth of the default screen as *int*.
AT_XDisplay	Returns the display for the specified dialog. The return value of the function is of the type “display *”.
AT_XFont	Returns the X Windows-specific structure for the specified font. The return value of the function is of the type “XFontStruct *”, if supported by the font defintion. Otherwise NULL.
AT_XFontSet	Dieser Wert liefert die X-Windows-spezifische Struktur für den angegebenen Zeichensatz zurück. Der Rückgabewert der Funktion ist vom Typ “XFontSet”, sofern zur Font-Definition passend. Andernfalls NULL. Returns the X Windows-specific structure for the specified font. The return value of the function is of the type “XFontSet”, if supported by the font defintion. Otherwise NULL.
AT_XmFontList	Dieser Wert liefert die X-Windows-spezifische Struktur für den angegebenen Zeichensatz zurück. Der Rückgabewert der Funktion ist vom Typ “XmFontList”, sofern zur Font-Definition passend. Andernfalls NULL. Returns the X Windows-specific structure for the specified font. The return value of the function is of the type “XFontSet”, if supported by the font defintion. Otherwise NULL.
	Returns the default font typically used by the IDM when the font attribute is not set.

attribute	Meaning
AT_XScreen	Returns the screen for the specified dialog. The return value of the function is of the type “screen”.
AT_XShell	Returns the shell widget of an object. The return value of the function is a widget.
AT_XtAp-pContext	Liefert den Application Context. Returns the Application Context.
AT_XTile	Returns the X Windows-specific structure for a tile. The return value of this function depends on how the tile was defined. If it was stored in an external ""“.gif”"-format file, ""“XImage *”"" is returned. If it was defined directly in the Dialog Manager file, “Pixmap” is returned. It ist recommended to better query AT_XTile with function **DM_GetToolkitDataEx**.
AT_XVisual	Dieser Wert liefert eine Visual Struktur für den Default-Screen. Returns a visual structure for the default screen.
AT_XWidget	Returns the widget of the specified object. The return value of the function is of the type “widget”.
AT_XWindow	Is the window belonging to the object. The return value of the function is of the type “window”.

To be noted for multiscreen dialogs

The call with AT_ XTile or AT_XColor always returns only the tile or color of the default screen (see also chapter „Multiscreen Ssupport untder Motif“ in manual „Programmiertechniken“).

Example

```
int DML_c AppMain (argc, argv)
int argc;
char far * far *argv;
{
  DM_ID dialogID;
  Widget toplevel;    /* Application shell obtained from
                         XtInitialize(). */
  static char * dialogfile = "xres.dlg";

  /* Initialize the dialog manager */
  DM_Initialize (&argc, argv, 0);
```

```
dialogID = DM_LoadDialog (dialogfile, 0);

if (!dialogID)
{
  DM_TraceMessage("%s: Could not load dialog \"%s\"",
      DMF_LogFile | DMF_InhibitTag | DMF_Printf,
      argv[0], dialogfile);
  return(1);
}


/* Querying the application shell */
if ((toplevel = (Widget) DM_GetToolkitData (dialogID,
  AT_XWidget)) != 0)
{
  DM_ID my_bgc;
  DM_ID my_font;
  DM_ID my_FontOfExit;

  /* further, own statements */ }
```

3.31.2 Microsoft Windows

Using this function, the data necessary for MICROSOFT WINDOWS such as “window-handle”, “instance handle” and “color” can be queried.

The meanings of these data types are explained in the corresponding Microsoft Windows manuals.

The following values are permitted for these attributes:

attribute	object	Return value	Meaning
AT_CanvasData	canvas	FPTR	This attribute can be used to retrieve the user-specific data of a canvas object. This data was set by DM_ SetToolkitData or a canvas callback function and contains any user-specific data (See also chapter „Structures for Canvas Functions“).

attribute	object	Return value	Meaning
AT_ClipboardText	setup or 0	DM_String	This attribute returns the string content of the Microsoft Windows clipboard. The return value is buffered and is valid until the attribute is queried again. Setting the attribute also invalidates the buffer. To release the string without changing the clipboard, invoke: `DM_SetToolkitData(0, AT_ClipboardText, (FPTR) 0, 0);`
AT_Color	color	COLORREF	See also AT_XColor
	tile	HPALETTE	See also AT_XColor
AT_DataType	tile	int	The query is only available for compatibility reasons. The attribute AT_Tile with set “data” parameter should be used with DM_GetToolkitDataEx. This attribute returns the type of the pattern. The assignment is described at AT_XTile.
AT_DPI	IDM Objects	int	See also AT_GetDPI
AT_Font	font	HFONT	See also AT_XFont
	IDM Objects	HFONT	See also AT_XFont
	setup	HFONT	See also AT_XFont
AT_GetDPI	setup or 0	int	This attribute returns the system DPI value. See the note below.
	IDM Objects	int	This attribute returns the DPI value of the object. This depends on which monitor the object is assigned to. See note below.

attribute	object	Return value	Meaning
AT_IsNull	font or color	int	Hiermit kann abgefragt werden, ob die Resource auf *NULL* definiert wurde (*UI_NULL_FONT* bzw. *UI_NULL_COLOR*). Die Resource wurde auf *NULL* definiert, wenn der Rückgabewert nicht 0 ist. Hereby it can be queried whether the resource was defined to *NULL* (*UI_NULL_FONT* or *UI_NULL_COLOR*). The resource has been defined to *NULL* if the return value is not *0*.
AT_maxsize	setup or 0	int	This attribute returns the number of WSI ID's that are still free. **Attention:** The number of WSI IDs still available has nothing to do with how many objects can actually still be made visible! It is the maximum upper limit.
AT_Raster	dialog editbox groupbox layoutbox module notebook notepage spinbox splitbox statusbar tablefield toolbar Window	DWORD	This attribute returns the size of the raster defined on the object in IDM pixels. The width and height is packed into a “DWORD”, see note below.
	font	DWORD	This attribute returns the size of the font in IDM pixels as used for raster calculation. The width and height is packed into a “DWORD”, see note below.

attribute	object	Return value	Meaning
AT_Sroll-barDimension	groupbox notepage window	DWORD	This attribute returns the width of the vertical scrollbar and the height of the horizontal scrollbar in IDM pixels. The width and height is packed into a “DWORD”, see note below.
AT_Size	dialog module	DWORD	This attribute returns the size of the primary monitor's workspace in IDM pixels. The width and height is packed into a “DWORD”, see note below.
	font	DWORD	This attribute returns the size of the font in IDM pixels. The width is calculated from the reference string if one is specified. The width and height is packed into a “DWORD”, see note below.
	Remaining IDM Objectsexcept menubox, menu-item and menuseparator	DWORD	This attribute returns the size of the object in IDM pixels. The width and height is packed into a “DWORD”, see note below.
AT_Tile	color	HBRUSH	See also AT_XTile
	tile	HANDLE	The query is only available for compatibility reasons. DM_ GetToolkitDataEx should be used with the “data” parameter set. See also AT_XTile
AT_toolhelp	setup or 0	HWND	This attribute returns the Microsoft Windows handle of the tooltip control used by Dialog Manager to display the *.toolhelp* attribute. See also “Example for AT_toolhelp at the setup object” below.
AT_value	RTF edittext	DM_String	This attribute returns the complete content, i.e. with all formatting instructions etc., of an RTF input field.

attribute	object	Return value	Meaning
AT_VSize	IDM Objekte except menubox, menuitem and menuseparator	DWORD	This attribute returns the virtual size of the object in IDM pixels. If there is no virtual size, then the real size is returned in IDM pixels. The width and height is packed into a “DWORD”, see note below.
AT_Widget	USW	HWND	See also AT_XWidget
AT_WinHandle	dialog module setup or 0	HINSTANCE	This attribute returns the Microsoft Windows handle of the application instance.
	menubox	HMENU	This attribute returns the Microsoft Windows Menu Handle.
	menuitem menuseparator	HMENU	This attribute returns the Microsoft Windows menu handle of the surrounding ***menubox*** object.
	Remaining IDM Objects	HWND	This attribute returns the Microsoft Windows handle of the object. The grouping objects (***groupbox***, ***notepage***, ***window***, ...) are usually composed of several Microsoft Windows objects, for these the handle of the “client” window (the window in which the child objects are created) is returned.
AT_wsidata	cursor	HCURSOR	See also AT_XCursor
	font	HFONT	See also AT_XFont
	tile	HANDLE	The query is only available for compatibility reasons. DM_ GetToolkitDataEx should be used with the “data” parameter set. See also AT_XTile
	Remaining IDM Objects	HWND	This attribute returns the Microsoft Windows handle of the outer Microsoft Windows object, the inner one can be queried with “AT_WinHandle”.

attribute	object	Return value	Meaning

AT_XColor	color	COLORREF	This attribute returns the Microsoft Windows-specific structure for the specified color. The color values are accessed using the appropriate Microsoft Windows macros:
			`COLORREF u1RGB = (COLORREF) (size_t) DM_GetToolkitDataEx (colorID, AT_XColor, (FPTR) 0, 0);`
			`BYTE ucRed = GetRValue (u1RGB);`
			`BYTE ucGreen = GetGValue (u1RGB);`
			`BYTE ucBlue = GetBValue (u1RGB);`

	tile	HPALETTE	This attribute returns the Microsoft Windows color palette used by the pattern.
AT_XCursor	cursor	HCURSOR	This attribute returns the Microsoft Windows cursor handle.
AT_XFont	font	HFONT	This attribute returns the Microsoft Windows font handle.
	IDM Objects	HFONT	This attribute returns the Microsoft Windows font handle of the font used on this object.
	setup	HFONT	The font handle of the default font used is returned.

attribute	object	Return value	Meaning
AT_XTile	color	HBRUSH	This attribute returns a Microsoft Windows Brush of the color. This brush can be used to fill the background.
	tile	HANDLE	The query of AT_wsidata, AT_Tile and AT_XTile without set "data" parameter is only available for compatibility reasons. The "data" parameter should be used. This attribute returns the Microsoft Windows specific structure for the pattern (tile) as in version A.06.03.a and before. GDI objects have to be created for this purpose. In order to obtain the new Microsoft Direct2D data, that the IDM uses internally, the "data" parameter must be set. The type of Microsoft Windows handle depends on AT_DataType, whereby AT_DataType may only be queried after AT_wsidata, AT_Tile or AT_XTile has been queried: » *DMF_TlkDataIsIcon*: HICON » *DMF_TlkDataIsWMF*: HMETAFILE » *DMF_TlkDataIsEMF*: HENHMETAFILE » Otherwise: HBITMAP **IMPORTANT**: The returned data should not be saved, as it is automatically released when the tile resource is no longer used by a visible IDM object. **Note**: If data was set using DM_SetToolkitData, then the set data and only the set data is returned.
AT_XWidget	USW	HWND	This attribute returns the Microsoft Windows handle of the USW object.

Note for object and attribute

The object specified in the call must generally be visible and thus created in the WSI for the returned data to make sense. Resources are generally created when they are called. If an object type is specified that is not mentioned for the attribute in question, an error message is usually written to the log file and “(FPTR) 0” is returned.

Note for access on the return value

The return value of the function is a “FPTR” or “void *”, this must be cast to the documented return value to avoid getting warnings when compiling.

Since a “void *” pointer can be cast to any other pointer, a simple cast operator is sufficient for all pointer data types. Pointer data types include, for example, all Microsoft Windows handles, such as “HWND”, “HFONT”, ... :

```
HWND hwnd = (HWND) DM_GetToolkitData(idObj, AT_wsidata);
```

For numerical values, an intermediate cast must usually be inserted, since the size of the data value must be preserved when casting from a pointer to a number in order to avoid warnings. The data type “size_t” can be used for this purpose, since it has the same size as a pointer by definition. Subsequently, it is possible to cast to a smaller number type:

```
DM_UInt2 val = (DM_UInt2) (size_t) DM_GetToolkitData(idObj, AT_wsidata);
```

Note width and height packed in “DWORD”

Under Microsoft Windows, width and height are often packed into a “DWORD”. This is also partly handled in this way by the IDM. The individual values can then be extracted with the Microsoft Windows macros “LOWORD” and “HIWORD”:

```
DWORD size = (DWORD) (size_t) DM_GetToolkitData(id, AT_Size);
WORD width = LOWORD(size);
WORD height = HIWORD(size);
```

Note IDM pixel

The IDM for WINDOWS 11 supports high resolutions. To minimize impact on existing dialog scripts, ISA DIALOG MANAGER uses virtual pixel coordinates. These are based on the size of an application that does not support high resolutions, such as IDM for WINDOWS 10.

Note for DPI values

Note that all DPI values are dynamic and can be changed by the user. For example, IDM objects can be moved to another monitor or the user can set other scale factors via the control panel.

If the application is not DPI Aware (for example IDM for WINDOWS 10) then the default DPI value of 96 is always used.

Notefor NULL values with ressources

The **DM_GetToolkitData** function returns a **NULL** value for font and color resources under MICROSOFT WINDOWS if the resource has been defined to **UI_NULL_FONT** or **UI_NULL_COLOR**,

respectively. This affects the following attributes:

» *AT_wsidata*, *AT_Font*, *AT_XFont*:

The return value for *UI_NULL_FONT* becomes *(HFONT) 0*.

» *AT_Color*, *AT_XColor*:

The return value for *UI_NULL_COLOR* becomes *(COLORREF) -1L*.

» *AT_Tile*, *AT_XTile*:

The return value for *UI_NULL_COLOR* becomes *(HBRUSH) 0*.

Note IDM pixel

Example

Querying a canvas-specific structure. If the structure is not yet defined at the canvas, the necessary memory will be allocated and the data will be transferred to the canvas.

```
void DML_default DM_ENTRY HoleCanvasDaten __3(
(DM_ID, thisID),
(DM_ID, canvasID),
(DM_ID, stvarID))
{
  MyData *d;
  /* Holen der Daten aus der Canvas und Abprüfen, ob man sie
  * wirklich bekommen hat, sonst Allokieren der Struktur
  /* Fetching data from the canvas and checking whether it has
  * actually arrived. Otherwise allocating the structure.
  */
  if ((d = (MyData *) DM_GetToolkitData (canvasID,
     AT_CanvasData)) == (MyData *) 0)
      if ((d = DM_Calloc (sizeof(MyData)))
        == (MyData *) 0)
      return;
    else
      if (!DM_SetToolkitData (canvasID, AT_CanvasData, d, 0))
        DM_TraceMessage ("Cannot set canvas toolkit data",
            0);
}
```

Example for AT_toolhelp at the setup object

Note

The functions “CloseToolhelp” and “SetToolhelpDisplayTime” used in the example must be defined accordingly within the dialog.

Temporarily closing an open tooltip control in a canvas function to redraw the canvas

```
#include <windows.h>
#include <commctrl.h>
#include IDMuser.h

void DML_default DM_ENTRY CloseToolhelp __0() {
  DM_ID idSetup =
  DM_ParsePath((DM_ID) 0, (DM_ID) 0, "setup", 0, 0);

  if (idSetuo != (DM_ID) 0) {
    HWND hwndToolhelp =
     (HWND) DM_GetToolkitData(idSetup, AT_toolhelp);

    if (hwndToolhelp != (HWND) 0) {
      SendMessage(
         hwndToolhelp, TTM_POP,
         (WPARAM) 0, (LPARAM) 0);
    }
  }
}
```

Change the display time of the tooltip

```
#include <windows.h>
#include <commctrl.h>
#include IDMuser.h

void DML_default DM_ENTRY SetToolhelpDisplayTime __1(
    (DM_Integer, iDisplayTime)) {
  DM_ID idSetup =
      DM_ParsePath((DM_ID) 0, (DM_ID) 0, "setup", 0, 0);

    if (idSetuo != (DM_ID) 0) {
      HWND hwndToolhelp =
          (HWND) DM_GetToolkitData(idSetup, AT_toolhelp);

        if (hwndToolhelp != (HWND) 0) {
          SendMessage(
              hwndToolhelp, TTM_SETDELAYTIME,
              (WPARAM) TTDT_AUTOPOP,
              (LPARAM) iDisplayTime);
       }
    }
}
```

3.31.3 Qt

The following values are permitted for these attributes:

attribute	object	Return value	Meaning
AT_Application	0	FPTR auf QApplication	This attribute can be used to query the QApplication on which the application is based.
AT_CanvasData	cancas	FPTR	Über dieses Attribut können die benutzerspezifischen Daten eines Canvas-Objekts erfragt werden. Diese Daten wurden von DM_SetToolkitData oder einer Canvas-Callback-Funktion gesetzt und beinhalten jegliche benutzerspezifischen Daten (Siehe auch Kapitel „Strukturen für Canvas-Funktionen“). This attribute can be used to retrieve the user-specific data of a canvas object. This data was set by DM_SetToolkitData or a canvas callback function and contains any user-specific data (See also chapter „Structures for Canvas Functions“).

attribute	object	Return value	Meaning
AT_Color	color	QColor / QBrush	This attribute can be used to query the color or color gradient (as a QBrush) used by the Color resource. Attention: It should always be checked first for a valid color (QColor), since a QBrush can be automatically cast to a QColor, which then however is an uninitialized but valid QColor.
AT_Font	font	QFont	This attribute can be used to query the QFont used by the font resource.
AT_FontName	font	char*	This attribute can be used to get the name of the QFont used by the font resource.
AT_Tile	tile	QPixmap	This attribute can be used to query the QPixmap of the pattern (tile).
AT_XTile	tile	QPixmap	See AT_Tile
AT_XWidget	IDM Objekte	QWidget	This attribute determines the QWidget associated with a DM_ID.

See also

Function DM_GetToolkitDataEx

3.32 DM_GetToolkitDataEx

This function is an extended form of DM_GetToolkitData and allows to pass additional values or structures via the parameters data and options.

Availability

since IDM version A.05.02.e

```
FPTR DML_default DM_EXPORT DM_GetToolkitDataEx
(
  DM_ID objectID,
  DM_Attribute attr,
  FPTR data,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

This parameter is the identifier of the object whose window system specific data should be requested.

-> DM_Attribute attr

This parameter specifies the attribute to be queried.

<-> FPTR data

Pointer to DM_ToolkitDataArgs structure. The structure is used for communication with the **DM_GetToolkitDataEx** function. It allows the transfer and return of different value types.

For compatible use to DM_GetToolkitData should be data = NULL.

-> DM_Options options

Currently unused, should be used with 0.

Return value

Depending on the type of the queried value, this function returns the corresponding values converted to *FPTR*.

Remarks

The function **DM_GetToolkitDataEx** does not exist for the server side and should not be called there!

3.32.1 Motif

Using this function you can query the data necessary for X-Windows, such as "window-id", "widget" and "color". The meanings of these datatypes are explained in the corresponding X-Windows manual.

The following values are permitted for the atttributes:

attribute	data	object	Meaning
AT_CanvasData	(FPTR) 0	canvas	Returns the user-specific data of a canvas. This data have been set by a canvas callback function and contains any user-specific data (see also chapter „Strukturen für Canvas-Funktionen“ in the „C Interface - Basics“ manual).
AT_CellRect	DM_ ToolkitDataArgs *data	tablefield	Determines the coordinates and size of a cell in IDM pixels for a tablefield (for this the “data->index” field must be set to the desired index (*argmask* = *DM_TKAM_index*)). These are stored in the *rectangle* field of the passed DM_ToolkitDataArgs structure. Cell means the rectangular area in the tablefield where shadows, focus and activation frames and text are drawn. The cell does not include the lines drawn between the cells. The coordinates supplied are relative to the upper left corner of the tablefield. If position and size of a cell could be determined, **DM_GetToolkitDataEx** returns the pointer to the specified DM_ToolkitDataArgs structure, in all other cases NULL. Position and size can only be determined if both tablefield and row and column of the cell are switched visible and the cell is fully or partially visible in the tablefield. A concrete, absolute invisibility (e.g. because the window is outside the visible screen or otherwise covered) cannot be excluded despite supplied position and size and is window system dependent.

attribute	data	object	Meaning
AT_DPI	DM_ ToolkitDataArgs *data	0	Returns - with set argmask=0 - the DPI information of the default screen (null objectID) or the DPI information of the screen on which the specified surface object is made visible. The function resets the argmask to *DM_ TKAM_dpi* | *DM_TKAM_scaledpi*. In the **dpi** field of the DM_ToolkitDataArgs structure then contains the default dpi value. In the substructure **scale.dpi** the DPI value to which the scaling from the default DPI value takes place. The element **scale.factor** element also returns the scaling factor in %. For a HiDPI-aware IDM application on a screen with 200% scaling you will typically find the values **dpi**=96 ,**scale.dpi**=192, **scale.factor**=200 are present. A conversion of pixel coordinates of the IDM into real screen pixels can then be done using with ***scale.factor**/100 or ***scale.dpi**/**dpi**.
AT_IsNull	(FPTR) 0	font	Returns a value<>*0* if the font is a *UI_NULL_ FONT*.
AT_ObjectID	DM_ ToolkitDataArgs *data	0	Determines the corresponding object ID for a widget. If successful, the data pointer is returned and in the DM_ToolkitDataArgs structure the ID is stored in the data substructure.
AT_Tile	(FPTR) 0	tile	See also AT_XTile
AT_XColor	(FPTR) 0	color	Returns the X Windows-specific structure for the specified color. The return value of the function is of the type “pixel”. The specified object must be a color.
AT_XColormap	(FPTR) 0	0	Dieser Wert liefert die Colormap des Default-Screens zurück. Returns a colormap of the default screen.

attribute	data	object	Meaning
AT_XCursor	(FPTR) 0	cursor	Returns the X Windows-specific structure for the specified cursor. The return value of the function is of the type “Cursor”. The specified object must be a cursor.
AT_XDepth	(FPTR) 0	0	Dieser Wert liefert die Farbtiefe des Default-Screens als *int*. Returns the color depth of the default screen as *int*.
AT_XDisplay	(FPTR) 0	0	Returns the display for the specified dialog. The return value of the function is of the type “display *”.
AT_XFont	(FPTR) 0	font	Returns the X Windows-specific structure for the specified font. The return value of the function is of the type “XFontStruct *”, if supported by the font defintion. Otherwise NULL.
AT_XFontSet	(FPTR) 0	font	Dieser Wert liefert die X-Windows-spezifische Struktur für den angegebenen Zeichensatz zurück. Der Rückgabewert der Funktion ist vom Typ “XFontSet”, sofern zur Font-Definition passend. Andernfalls NULL. Returns the X Windows-specific structure for the specified font. The return value of the function is of the type “XFontSet”, if supported by the font defintion. Otherwise NULL.
AT_XmFontList	(FPTR) 0	font	Dieser Wert liefert die X-Windows-spezifische Struktur für den angegebenen Zeichensatz zurück. Der Rückgabewert der Funktion ist vom Typ “XmFontList”, sofern zur Font-Definition passend. Andernfalls NULL. Returns the X Windows-specific structure for the specified font. The return value of the function is of the type “XFontSet”, if supported by the font defintion. Otherwise NULL.
		0 or visible IDM Objects	Returns the default font typically used by the IDM when the font attribute is not set.

attribute	data	object	Meaning
AT_XScreen	(FPTR) 0	0	Returns the screen for the specified dialog. The return value of the function is of the type “screen”.
AT_XShell	(FPTR) 0	0	Returns the shell widget of an object. The return value of the function is a widget.
AT_XtAp-pContext	(FPTR) 0	0	Liefert den Application Context. Returns the Application Context.
AT_XTile	(FPTR) 0	tile	Returns the X Windows-specific structure for a tile. The return value of this function depends on how the tile was defined. If it was stored in an external ""“.gif”"-format file, ""“XImage *”" is returned. If it was defined directly in the Dialog Manager file, “Pixmap” is returned.
	DM_ ToolkitDataArgs *data	tile	This value returns the X-Windows specific structure of a pattern tile (with set argmask=0 or armask=DM_TKAM_scaledpi). If image information is present, the following information is returned in the DM_ToolkitDataArgs structure: » DM_TKAM_tile: In the tile substructure the image type (DM_GFX_PIXMAP or DM_GFX_XIMAGE)in gfxtype. For a PIXMAP the image information in the pixmap member and the transparency clipmask in the trans_mask member. For an XIMAGE the full image information is supplied in the ximage member. » DM_TKAM_rectangle: width and height of the image in the rectangle substructure. » DM_TKAM_dpi: The DPI information analog to the AT_DPI call. » DM_TKAM_scaledpi: The DPI information analogous to the AT_DPI call.

attribute	data	object	Meaning
AT_XVisual	(FPTR) 0	0	Dieser Wert liefert eine Visual Struktur für den Default-Screen. Returns a visual structure for the default screen.
AT_XWidget	(FPTR) 0	IDM Objects	Returns the widget of the specified object. The return value of the function is of the type “widget”.
AT_XWindow	(FPTR) 0	IDM Objects	Is the window belonging to the object. The return value of the function is of the type “window”.

To be noted for multiscreen dialogs

The call with AT_ XTile or AT_XColor always returns only the tile or color of the default screen (see also chapter „Multiscreen Ssupport untder Motif“ in manual „Programmiertechniken“).

3.32.2 Microsoft Windows

Using this function, the data necessary for MICROSOFT WINDOWS such as “window-handle”, “instance handle” and “color” can be queried.

The meanings of these data types are explained in the corresponding Microsoft Windows manuals.

The following values are permitted for these attributes:

attribute	data	object	Return value	Meaning
AT_CanvasData	(FPTR) 0	canvas	FPTR	This attribute can be used to retrieve the user-specific data of a canvas object. This data was set by DM_ SetToolkitData or a canvas callback function and contains any user-specific data (See also chapter „Structures for Canvas Functions“).

attribute	data	object	Return value	Meaning
AT_CellRect	DM_ ToolkitDataArgs *data	tablefield	data	This attribute determines the coordinates of a tablefield cell in IDM pixels. For this the “data->index” field must be set to the desired index (do not forget the bit “DM_ TKAM_index” in “data->argmask”). In data->argmask the bit “DM_TKAM_rectangle” is set and the corresponding fields are filled in (see description DM_ ToolkitDataArgs). By cell is meant the rectangular area in the tablefield where shadows, focus and activation frames and text are drawn. The cell does not include the lines drawn between the cells. The coordinates supplied are relative to the upper left corner of the tablefield. If position and size of a cell could be determined, DM_ GetToolkitDataEx returns the pointer to the specified DM_ ToolkitDataArgs structure, in all other cases (FPTR) 0. Position and size can only be determined if both

attribute	data	object	Return value	Meaning
				tablefield and row and column of the cell are switched visible and the cell is completely or partially visible in the tablefield. A concrete, absolute invisibility (e.g. because the window is outside the visible screen or otherwise covered) cannot be excluded despite supplied position and size and is window system dependent.
AT_ClipboardText	(FPTR) 0	setup or 0	DM_String	This attribute returns the string content of the Microsoft Windows clipboard. The return value is buffered and is valid until the attribute is queried again. Setting the attribute also invalidates the buffer. To release the string without changing the clipboard, invoke: `DM_SetToolkitData (0, AT_ClipboardText, (FPTR) 0, 0);`
AT_Color	(FPTR) 0	color	COLORREF	See also AT_XColor
		tile	HPALETTE	See also AT_XColor

attribute	data	object	Return value	Meaning
AT_DataType	(FPTR) 0	tile	int	The query is only available for compatibility reasons. The attribute AT_Tile with set “data” parameter should be used with DM_ GetToolkitDataEx. This attribute returns the type of the pattern. The assignment is described at AT_ XTile.

attribute	data	object	Return value	Meaning
AT_DPI	(FPTR) 0	IDM Objects	int	See also AT_GetDPI
	DM_ ToolkitDataArgs *data	setup oder 0	int	This attribute returns the system DPI value as in "AT_GetDPI". If "DM_TKAM_handle" is set, the DPI value of the Microsoft Windows control whose window handle (HWND) is specified in "data->handle" is determined instead. In addition, the bits "DM_TKAM_dpi" and "DM_TKAM_scaleddpi" are set in "data->argmask" and the corresponding fields are filled out (see description DM_ ToolkitDataArgs).
		Remaining IDM Objects	int	This attribute returns the DPI value of the object as in AT_ GetDPI. In addition, in data->argmask the bits DM_TKAM_dpi and DM_TKAM_ scaleddpi are set and the corresponding fields are filled in (see description DM_ ToolkitDataArgs).
AT_Font	(FPTR) 0	font	HFONT	See also AT_XFont
		IDM Objects	HFONT	See also AT_XFont
		setup	HFONT	See also AT_XFont

attribute	data	object	Return value	Meaning
AT_GetDPI	(FPTR) 0	setup or 0	int	This attribute returns the system DPI value. See the note below.
		IDM Objects	int	This attribute returns the DPI value of the object. This depends on which monitor the object is assigned to. See note below.
	HWND data	setup or 0	int	This attribute returns the DPI value of the Microsoft Windows object whose handle (HWND) was passed in “data”. This depends on which monitor the object is assigned to. See note below.
AT_IsNull		font or color	int	Hiermit kann abgefragt werden, ob die Resource auf *NULL* definiert wurde (*UI_NULL_FONT* bzw. *UI_NULL_COLOR*). Die Resource wurde auf *NULL* definiert, wenn der Rückgabewert nicht 0 ist. Hereby it can be queried whether the resource was defined to *NULL* (*UI_NULL_FONT* or *UI_NULL_COLOR*). The resource has been defined to *NULL* if the return value is not *0*.

attribute	data	object	Return value	Meaning
AT_maxsize	(FPTR) 0	setup or 0	int	This attribute returns the number of WSI ID's that are still free. **Attention:** The number of WSI IDs still available has nothing to do with how many objects can actually still be made visible! It is the maximum upper limit.
AT_ObjectID	DM_ ToolkitDataArgs *data	setup oder 0	data	This attribute returns the DM_ID of a Microsoft Windows object. For this the “data->handle” field must be set to the Microsoft Windows window handle (HWND) (do not forget the bit “DM_ TKAM_handle” in “data->argmask”). If a Dialog Manager ID can be determined, the return value is set to “data”, the “data->argmask” bit “DM_ TKAM_data” is set and the “data->data” field is filled with the DM_ID.

attribute	data	object	Return value	Meaning
AT_Raster	(FPTR) 0	dialog editbox groupbox layoutbox module notebook notepage spinbox splitbox statusbar tablefield toolbar Window	DWORD	This attribute returns the size of the raster defined on the object in IDM pixels. The width and height is packed into a “DWORD”, see note below.
		font	DWORD	This attribute returns the size of the font in IDM pixels as used for raster calculation. The width and height is packed into a “DWORD”, see note below.
AT_Sroll-barDimension	(FPTR) 0	groupbox notepage window	DWORD	This attribute returns the width of the ver-tical scrollbar and the height of the hori-zontal scrollbar in IDM pixels. The width and height is packed into a “DWORD”, see note below.

attribute	data	object	Return value	Meaning
AT_Size	(FPTR) 0	dialog module	DWORD	This attribute returns the size of the primary monitor's workspace in IDM pixels. The width and height is packed into a “DWORD”, see note below.
		font	DWORD	This attribute returns the size of the font in IDM pixels. The width is calculated from the reference string if one is specified. The width and height is packed into a “DWORD”, see note below.
		Remaining IDM Objectsexcept menubox, menuitem and menuseparator	DWORD	This attribute returns the size of the object in IDM pixels. The width and height is packed into a “DWORD”, see note below.
AT_Tile	(FPTR) 0	color	HBRUSH	See also AT_XTile
		tile	HANDLE	The query is only available for com-patibility reasons. DM_ GetToolkitDataEx should be used with the “data” parameter set. See also AT_XTile

attribute	data	object	Return value	Meaning
AT_toolhelp	(FPTR) 0	setup or 0	HWND	This attribute returns the Microsoft Windows handle of the tooltip control used by Dialog Manager to display the *.toolhelp* attribute. See also “DM_GetToolkitDataEx” below.
AT_value	(FPTR) 0	RTF edittext	DM_String	This attribute returns the complete content, i.e. with all formatting instructions etc., of an RTF input field.
AT_VSize	(FPTR) 0	IDM Objekte except menubox, menuitem and menuseparator	DWORD	This attribute returns the virtual size of the object in IDM pixels. If there is no virtual size, then the real size is returned in IDM pixels. The width and height is packed into a “DWORD”, see note below.
AT_Widget	(FPTR) 0	USW	HWND	See also AT_XWidget

attribute	data	object	Return value	Meaning
AT_WinHandle	(FPTR) 0	dialog module setup or 0	HINSTANCE	This attribute returns the Microsoft Windows handle of the application instance.
		menubox	HMENU	This attribute returns the Microsoft Windows Menu Handle.
		menuitem menuseparator	HMENU	This attribute returns the Microsoft Windows menu handle of the surrounding ***menubox*** object.
		Remaining IDM Objects	HWND	This attribute returns the Microsoft Windows handle of the object. The grouping objects (***groupbox***, ***notepage***, ***window***, ...) are usually composed of several Microsoft Windows objects, for these the handle of the “client” window (the window in which the child objects are created) is returned.

attribute	data	object	Return value	Meaning
AT_wsidata	(FPTR) 0	cursor	HCURSOR	See also AT_XCursor
		font	HFONT	See also AT_XFont
		tile	HANDLE	The query is only available for compatibility reasons. DM_ GetToolkitDataEx should be used with the “data” parameter set. See also AT_XTile
		Remaining IDM Objects	HWND	This attribute returns the Microsoft Windows handle of the outer Microsoft Windows object, the inner one can be queried with “AT_WinHandle”.

attribute	data	object	Return value	Meaning

AT_XColor	(FPTR) 0	color	COLORREF	This attribute returns the Microsoft Windows-specific structure for the specified color. The color values are accessed using the appropriate Microsoft Windows macros:
				`COLORREF u1RGB = (COLORREF) (size_t) DM_GetToolkitDataEx (colorID, AT_XColor, (FPTR) 0, 0);`
				`BYTE ucRed = GetRValue (u1RGB);`
				`BYTE ucGreen = GetGValue (u1RGB);`
				`BYTE ucBlue = GetBValue (u1RGB);`

		tile	HPALETTE	This attribute returns the Microsoft Windows color palette used by the pattern.
AT_XCursor	(FPTR) 0	cursor	HCURSOR	This attribute returns the Microsoft Windows cursor handle.

attribute	data	object	Return value	Meaning
AT_XFont	(FPTR) 0	font	HFONT	This attribute returns the Microsoft Windows font handle.
		IDM Objects	HFONT	This attribute returns the Microsoft Windows font handle of the font used on this object.
		setup	HFONT	The font handle of the default font used is returned.

attribute	data	object	Return value	Meaning
AT_XTile	(FPTR) 0	color	HBRUSH	This attribute returns a Microsoft Windows Brush of the color. This brush can be used to fill the back-ground.

attribute	data	object	Return value	Meaning
	(FPTR) 0	tile	HANDLE	The query of AT_wsidata, AT_Tile and AT_XTile without set "data" parameter is only available for compatibility reasons. The "data" parameter should be used. This attribute returns the Microsoft Windows specific structure for the pattern (tile) as in version A.06.03.a and before. GDI objects have to be created for this purpose. In order to obtain the new Microsoft Direct2D data, that the IDM uses internally, the "data" parameter must be set. The type of Microsoft Windows handle depends on AT_DataType, whereby AT_DataType may only be queried after AT_wsidata, AT_Tile or AT_XTile has been queried: » *DMF_TlkDataIsIcon*: HICON » *DMF_TlkDataIsWMF*: HMETAFILE » *DMF_TlkDataIsEMF*: HENHMETAFILE

attribute	data	object	Return value	Meaning
			» *DMF_TlkDataIsIcon*: HICON » *DMF_TlkDataIsWMF*: HMETAFILE » *DMF_TlkDataIsEMF*: HENHMETAFILE » Otherwise: HBITMAP **IMPORTANT**: The returned data should not be saved, as it is automatically released when the tile resource is no longer used by a visible IDM object. **Note**: If data was set using DM_SetToolkitData, then the set data and only the set data is returned.	

attribute	data	object	Return value	Meaning
DM_ ToolkitDataArgs *data	tile	HANDLE / LPUNKNOWN	This attribute returns the Microsoft Windows specific structure for the pattern (tile) as in version A.06.03.a and before. GDI objects have to be created for this purpose. In order to receive the new Microsoft Direct 2D data that IDM uses internally, the “DM_ TKAM_tile_req” bit must be set in “data->argmask”. The desired data types are specified in “data->tile_req”, the following values are possible: - DM_GFX_ BMP: GDI Bitmap Handle (HBITMAP) - DM_GFX_ WMF: GDI Metafile Handle (HMETAFILE) - DM_GFX_ EMF: GDI	

attribute	data	object	Return value	Meaning
			Enhanced Metafile Handle (HENHMETAFILE) - DM_GFX_ICO: GDI Icon Handle (HICON) - DM_GFX_D2D1BMP: Direct2D Bitmap (ID2D1Bitmap *) - DM_GFX_D2D1SVG: Direct2D SVG Documnet (ID2D1SvgDocument *) - DM_GFX_D2D1EMF: Direct2D Metafile (ID2D1GdiMetafile *) These values can be linked with “bitwise or”. One of them is then selected, the data type DM_GFX_BMP is always used as a fallback (even if not explicitly specified). If necessary and possible, it is converted to one of the	

attribute	data	object	Return value	Meaning
			desired data types, which may consume additional resources. This attribute returns the Microsoft Windows specific structure for the pattern (tile). The pattern is determined for a requested DPI value. This is determined in the specified order from the following data: » If in “data->argmask” the “DM_TKAM_handle” bit is set, then the DPI value of the Microsoft Windows control is determined whose Windows handle (HWND) is in the “data->handle” field. » Otherwise,	

attribute	data	object	Return value	Meaning
			if the “DM_TKAM_scaledpi” bit is set in “data->argmask”, then the system DPI value is determined (corresponding to the scaling factor of the primary monitor). » If none of the previous conditions was met, then the original image size, i.e. the DPI value for which the images were designed, is used. Also, in “data->argmask” the “DM_TKAM_tile”, “DM_TKAM_rectangle”, “DM_TKAM_dpi” and “DM_TKAM_scaleddpi” bits are set and the	

attribute	data	object	Return value	Meaning
			corresponding fields are filled in (see description DM_ ToolkitDataArgs), where the DPI values are set to the requested DPI value and the rectangle is set to the size matching this requested DPI value. Note: This does not mean that the image data will be the appropriate size, it may need to be scaled. The return value corresponds to either “data->tile.data” or “data->tile.iunk”, see DM_ ToolkitDataArgs. **Note**: The data type of the return value is either in the “entry data->tile.gfxtype” or “data->tile.datatype”. The	

attribute	data	object	Return value	Meaning
			query of AT_ DataType is obsolete and may no longer be used when using the “data” parameter. **IMPORTANT**: The returned data should not be saved, as it is automatically released when the tile resource is no longer used by a visible IDM object. **Note**: If data was set using DM_ SetToolkitData, then the set data and only the set data is returned.	
AT_XWidget	(FPTR) 0	USW	HWND	This attribute returns the Microsoft Windows handle of the USW object.

Note for object and attribute

The object specified in the call must generally be visible and thus created in the WSI for the returned data to make sense. Resources are generally created when they are called. If an object type is specified that is not mentioned for the attribute in question, an error message is usually written to the log file and “(FPTR) 0” is returned.

Note for access on the return value

The return value of the function is a “FPTR” or “void *”, this must be cast to the documented return value to avoid getting warnings when compiling.

Since a “void *” pointer can be cast to any other pointer, a simple cast operator is sufficient for all pointer data types. Pointer data types include, for example, all Microsoft Windows handles, such as “HWND”, “HFONT”, ... :

```
HWND hwnd = (HWND) DM_GetToolkitDataEx(idObj, AT_wsidata, (FPTR) 0, 0);
```

For numerical values, an intermediate cast must usually be inserted, since the size of the data value must be preserved when casting from a pointer to a number in order to avoid warnings. The data type “size_t” can be used for this purpose, since it has the same size as a pointer by definition. Subsequently, it is possible to cast to a smaller number type:

```
DM_UInt2 val = (DM_UInt2) (size_t) DM_GetToolkitDataEx(idObj, AT_wsidata,
(FPTR) 0, 0);
```

Note width and height packed in “DWORD”

Under Microsoft Windows, width and height are often packed into a “DWORD”. This is also partly handled in this way by the IDM. The individual values can then be extracted with the Microsoft Windows macros “LOWORD” and “HIWORD”:

```
DWORD size = (DWORD) (size_t) DM_GetToolkitDataEx(id, AT_Size, (FPTR) 0, 0);
WORD width = LOWORD(size);
WORD height = HIWORD(size);
```

Note IDM pixel

The IDM for WINDOWS 11 supports high resolutions. To minimize impact on existing dialog scripts, ISA DIALOG MANAGER uses virtual pixel coordinates. These are based on the size of an application that does not support high resolutions, such as IDM for WINDOWS 10.

Note for DPI values

Note that all DPI values are dynamic and can be changed by the user. For example, IDM objects can be moved to another monitor or the user can set other scale factors via the control panel.

If the application is not DPI Aware (for example IDM for WINDOWS 10) then the default DPI value of 96 is always used.

Notefor NULL values with ressources

The **DM_GetToolkitDataEx** function returns a **NULL** value for font and color resources under MICROSOFT WINDOWS if the resource has been defined to **UI_NULL_FONT** or **UI_NULL_COLOR**, respectively. This affects the following attributes:

- *AT_wsidata*, *AT_Font*, *AT_XFont*:

  The return value for *UI_NULL_FONT* becomes *(HFONT) 0*.
- *AT_Color*, *AT_XColor*:

  The return value for *UI_NULL_COLOR* becomes *(COLORREF) -1L*.

» *AT_Tile*, *AT_XTile*:

The return value for *UI_NULL_COLOR* becomes *(HBRUSH) 0*.

Note IDM pixel

3.32.3 Qt

The following values are permitted for these attributes:

attribute	data	object	Return value	Meaning
AT_Application	(FPTR) 0	0	FPTR auf QApplication	This attribute can be used to query the QApplication on which the application is based.
AT_CanvasData	(FPTR) 0	cancas	FPTR	Über dieses Attribut können die benutzerspezifischen Daten eines Canvas-Objekts erfragt werden. Diese Daten wurden von DM_SetToolkitData oder einer Canvas-Callback-Funktion gesetzt und beinhalten jegliche benutzerspezifischen Daten (Siehe auch Kapitel „Strukturen für Canvas-Funktionen“). This attribute can be used to retrieve the user-specific data of a canvas object. This data was set by DM_SetToolkitData or a canvas callback function and contains any user-specific data (See also chapter „Structures for Canvas Functions“).

attribute	data	object	Return value	Meaning
AT_Color	(FPTR) 0	color	QColor / QBrush	This attribute can be used to query the color or color gradient (as a QBrush) used by the Color resource. Attention: It should always be checked first for a valid color (QColor), since a QBrush can be automatically cast to a QColor, which then however is an uninitialized but valid QColor.
AT_DPI	(DM_ ToolkitDataArgs *data	0	int	This attribute returns the system DPI value. In addition, the DM_ TKAM_dpi and DM_ TKAM_scaleddpi bits are set in data->argmask and the corresponding fields are filled in (see description DM_ToolkitDataArgs).
AT_Font	(FPTR) 0	font	QFont	This attribute can be used to query the QFont used by the font resource.
AT_FontName	(FPTR) 0	font	char*	This attribute can be used to get the name of the QFont used by the font resource.

attribute	data	object	Return value	Meaning
AT_ObjectID	DM_ ToolkitDataArgs *data	0	data	This attribute returns the DM_ID of a Qt widget. This requires that the pointer to DM_ ToolkitDataArgs structure has the “data->widget” field set to the QWidget and argmask = DM_TKAM_widget. If a Dialog Manager ID can be determined, the return value is set to data, the data->argmask bit “DM_ TKAM_data” is set and the “data->data” field is filled with the DM_ID (see description DM_ ToolkitDataArgs).
AT_Tile	(FPTR) 0	tile	QPixmap	This attribute can be used to query the QPixmap of the pattern (tile).

attribute	data	object	Return value	Meaning
AT_XTile	(FPTR) 0	tile	QPixmap	See AT_Tile
	DM_ ToolkitDataArgs *data	tile	QPixmap	This attribute returns the Qt specific structure for the pattern (tile). Also, in “data->argmask” the “DM_ TKAM_tile”, “DM_ TKAM_rectangle”, “DM_TKAM_dpi” and “DM_TKAM_ scaleddpi” bits are set and the corresponding fields are filled in (see description DM_ ToolkitDataArgs). The return value corresponds to “data->tile.pixmap”.
AT_XWidget	(FPTR) 0	IDM Objekte	QWidget	This attribute determines the QWidget associated with a DM_ID.

See also

Function DM_GetToolkitDataEx

3.33 DM_GetValue

Using this function you can query attributes of DM objects. For the attributes which are allowed for the relevant object type please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_GetValue
(
  DM_ID objectID,
  DM_Attribute attr,
  DM_UInt index,
  DM_Value *data,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

This parameter describes the object whose attribute you want to query.

-> DM_Attribute attr

This parameter describes the object attribute you want to query. All attributes permitted are defined in the file **IDMuser.h**.

-> DM_UInt index

This parameter is analyzed only in vector attributes of objects and describes the index of the desired object (e.g. text in listbox).

-> DM_Value *data

In this parameter you can query the attribute value. You should make sure that you read out the correct element out of this union. For the data type of each attribute please refer to the “Object Reference”.

-> DM_Options options

With this parameter you can control which form of texts are returned by DM, if the corresponding attribute is of the text-type. The following assignment is possible for this parameter:

Option	Meaning
DMF_GetMasterString	This object means that the string of textual attributes is to be returned in the development language, independently of the language the user is working with.
DMF_GetLocalString	This option means that the string of textual attributes is returned in the selected language.

Option	Meaning
DMF_GetTextID	This option means that the string of textual attributes is returned as textID. This is especially useful if the text is to be assigned to another object.
DMF_ DontFreeLastStrings	Usually, strings are transferred in a temporary buffer (which remains until the next call to DM) to the application. If strings are to remain valid for a longer time in the application, the option *DMF_ DontFreeLastStrings* has to be set. The memory will only be released, if a DM function is called without this option and if then a string is returned by the DM to the application.

Return Value

TRUE	The object could be queried successfully.
FALSE	The attribute is not permitted for this object.

Example

Querying the contents of an editable text in an object-callback function.

```
DM_Boolean DML_default DM_CALLBACK CheckFilename __1(
(DM_CallBackArgs *, data))
{
    DM_Value value;    /*structure for DM_GetValue*/

    /* get current contents */
    if (DM_GetValue(data->object, AT_content, 0, &value,
                    DMF_GetLocalString))
        /* check the datatype */
        if(value.type == DT_string)
```

See Also

Built-in function getvalue in manual “Rule Language”

3.34 DM_GetValueIndex

This function can be used to query attributes with two indexes.

The attributes which are allowed for the relevant object type are listed in the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_GetValueIndex
(
  DM_ID objectID,
  DM_Attribute attr,
  DM_Value *index,
  DM_Value *data,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Describes the object whose attribute you query.

-> DM_Attribute attr

Describes the attribute you query. Valid attributes are defined in **IDMuser.h**.

-> DM_Value *index

Specifies the data type of the index (enum, index) and its value.

-> DM_Value *data

In this parameter you receive the value of the desired attribute. You have to make sure that the right element has to be read out of this union. The attributes which are allowed for the relevant object type are listed in the “Object Reference”.

-> DM_Options options

Using this parameter you can control in which form the texts are returned from the Dialog Manager, if the corresponding attribute is of the text-type. The following assignments are possible with this parameter:

Option	Meaning
DMF_GetMasterString	This object means that the string of textual attributes is to be returned in the development language, independently of the language the user is working with.
DMF_GetLocalString	This option means that the string of textual attributes is returned in the selected language.
DMF_GetTextID	This option means that the string of textual attributes is returned as textID. This is especially useful if the text is to be assigned to another object.

Option	Meaning
DMF_ DontFreeLastStrings	Usually, strings are transferred in a temporary buffer (which remains until the next call to DM) to the application. If strings are to remain valid for a longer time in the application, the option *DMF_ DontFreeLastStrings* has to be set. The memory will only be released, if a DM function is called without this option and if then a string is returned by the DM to the application.

Return Value

TRUE	The object could be queried successfully.
FALSE	The attribute is not permitted for this object.

Example

Querying a vectorial user-defined attribute of a groupbox or a window.

```
void DML_default DM_ENTRY GetInfo  __1((DM_ID, obj))
{
  DM_Value  attr;
  DM_Value data, index;
  DM_ID groupbox;

  DM_GetValue(obj, AT_parent, 0, &data, 0);
  groupbox = data.value.id;
  DM_TraceMessage("\nin GetInfo\n",DMF_Printf);

  /*
  ** Getting the number of user-defined attributes
  */
  if (DM_GetValue(obj, AT_membercount, 0, &data, 0))
    DM_TraceMessage("groupbox.membercount = %ld\n",
        DMF_Printf, data.value.integer);

  /*
  ** Choosing the name of the user-defined attribute
  */
   index.type = DT_string;
   index.value.string = ".StringVec";
   if (DM_GetValueIndex(groupbox, AT_label, &index, &attr, 0))
   {
    DM_Integer n, i;
    DM_GetValueIndex(groupbox, AT_count, &attr, &data, 0);
    n = data.value.integer;
    for (i=1 ; i<=n; i++)
```

```
    {
      DM_GetValue(groupbox, attr.value.attribute, i, &data,
        0);
      DM_TraceMessage ("first attribute%ld.string %s\n",
        DMF_Printf, i, data.value.string);
    }
  }
}
```

3.35 DM_GetVectorValue

Using this function, attributes occurring several times in an object (vector attributes) can be queried (see end of this function description for details). Moreover, you can query user-defined attributes by means of this function.

In contrast to **DM_GetValue**, this function causes the DM to process structures and allocate memory. Afterward, the memory allocated must be released by DM_FreeVector.

```
DM_Boolean DML_default DM_EXPORT DM_GetVectorValue
(
  DM_ID objectID,
  DM_Attribute attr,
  DM_Value *firstindex,
  DM_Value *lastindex,
  DM_VectorValue **values,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Describes the object whose attribute you query.

-> DM_Attribute attr

This parameter describes the object attribute you want to query.

-> DM_Value *firstindex

Using this parameter you can control which part of the contents is to be queried by this function. In this parameter the starting point of the part is defined. For a one-dimensional attribute this means that the type in the DM_Value structure is set to DT_integer and the integer values in the union are assigned the starting value. For a two-dimensional attribute this means that the type in the DM_Value structure is set to DT_index and that the index value in the union is assigned the starting value. In index.first the row has to be specified, in index.second the column has to be indicated.

Note

If this parameter is a *NULL* pointer, the starting point has the following default values:

Listbox.content:	integer = 1
Tablefield.content:	index.first = 1, index.second = 1

-> DM_Value *lastindex

Using this parameter you can control which part of the contents is to be queried by the function. In this parameter the starting point of the part is defined. For a one-dimensional attribute this means that the type in the DM_Value structure is set to DT_integer and the integer values in the union are assigned the starting value. For a two-dimensional attribute this means that the type in the DM_

Value structure is set to DT_index and that the index value in the union is assigned the starting value. In index.first the row has to be specified, in index.second the column has to be indicated.

Note

If this parameter is a *NULL* pointer, the ending point has the following default values:

Listbox.content:	integer = object.itemcount
Tablefield.content:	index.first = object.rowcount, index.second = object.colcount

<- DM_VectorValue **values

This parameter is a pointer to the values which are to be queried. In the DM_VectorValue structure you can control via the field *type* the data type of the individual values. You can control in the DM_VectorValue structure via the field *count* how many values the vector contains. The fields *type* and *count* are executed by the function call.

-> DM_Options options

Using this parameter you can control the form of the texts to be returned from DM, if the corresponding attribute is of the text-type.

Option	Meaning
DMF_GetMasterString	This object means that the string of textual attributes is to be returned in the development language, independently of the language the user is working with.
DMF_GetLocalString	This option means that the string of textual attributes is returned in the selected language.
DMF_GetTextID	This option means that the string of textual attributes is returned as textID. This is especially useful if the text is to be assigned to another object.
DMF_DontFreeLastStrings	Usually, strings are transferred in a temporary buffer (which remains until the next call to DM) to the application. If strings are to remain valid for a longer time in the application, the option *DMF_DontFreeLastStrings* has to be set. The memory will only be released, if a DM function is called without this option and if then a string is returned by the DM to the application.

Return Value

TRUE	The object could be queried successfully.
FALSE	The attribute is not permitted for this object.

Example

Querying row-wise the contents of a tablefield with 5 columns.

```
/*
*write the content of a tablefield to a file
*the file format is described above
*/
DM_Boolean DML_default DM_ENTRY SaveTable_ _2(
(DM_ID, Table),
(char *, filename))
{
    DM_boolean retval = FALSE;
    DM_VectorValue *vector;

    if (DM_GetVectorValue (table, AT_field, (DM_Value *) 0,
        (DM_Value *) 0, &vector, 0))
    {
        FILE *f;

        if ((f = fopen(filename, "w")))
        {
          int vpos = 0;
          int i;

          retval = TRUE;

          while ((vpos + 5) < vector->count)
          {
              DM_boolean ok = TRUE;
              for (i=0; i<5, i++)
                  if (!vector->vector.stringPtr[vpos+i]
                && !*vector->vector.stringPtr[vposi])
                      ok = FALSE;
            if (ok)
                for (1=0, i<5; i++)
                {
                fputs(vector->vector.stringPtr[vpos+i], f);
                    putc((i<4) ? ' ' : '\n', f);
                vpos += 5;
                }
          }
          fclose(f);

          DM_FreeVectorValue(vector,0);
    }
    return retval;
```

```
}
```

3.36 DM_IndexReturn

This function is used to safely return local index values (**DM_Index**) from a function. When local variables and structures are used in a C function, they are invalid after they have been returned. This function can safely and easily return a local index.

For this purpose, a temporary copy is created.

```
DM_Index * DML_default DM_EXPORT DM_IndexReturn
(
  DM_Index   *pindex,
  DM_Options  options
)
```

Parameters

-> DM_Index * pindex

This parameter refers to the index that shall be returned.

-> DM_Options options

Should be set to *0* since no options are available.

Return value

Zurückgegeben wird ein für die Funktionsrückgabe gültiger Zeiger auf eine **DM_Index**-Struktur oder *NULL* im Fehlerfall. Ein Fehler kann z.B. vorliegen, wenn die Funktion im falschen “runstate” aufgerufen wird oder das Kopieren nicht ausgeführt werden konnte.

A pointer to a valid **DM_Index** structure is returned or *NULL* in case of an error. An error may occur, for instance, if the function is called in the wrong “runstate” or copying failed.

Example

Dialog File

```
dialog YourDialog
function index SwapIndex(index Idx);

on dialog start
{
  print SwapIndex([1,3]);
  exit();
}
```

C Part

```
...

DM_Index* DML_default DM_ENTRY StringOf(DM_Index* Idx)
```

```
{
  DM_Index newIdx;
  newIdx.first = Idx->second;
  newIdx.second = Idx->first;

  /* wrong: return &newIdx; => newIdx is local! */
  return DM_IndexReturn(&newIdx, 0);
}
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_StringReturn, DM_ValueReturn

3.37 DM_Initialize

Using this function DM-internal data structures are initialized. This function must be called exactly **once** before the application is started.

```
DM_Boolean DML_default DM_EXPORT DM_Initialize
(
  int far *argc,
  char far * far *argv,
  DM_Options options
)
```

Parameters

<-> int far *argc

In this parameter a pointer is transferred to the number of command-line arguments. DM changes this parameter. It removes all arguments it can process in the following argument vector and reduces the number of arguments accordingly.

<-> char far * far *argv

Vector to the argument list. DM removes from this list all arguments it can process directly.

-> DM_Options options

Option	Meaning
0	To set no option
DMF_FatalNetErrors	Sets a compatible behavior to the IDM versions before A.05.01.d for the DISTRIBUTED DIALOG MANAGERS (DDM), enforcing an immediate termination on client and server side when a network, protocol or version error occurs. This means that except for local applications no more *start* and *finish* events will be triggered and **AppFinish** will not be called. Thus the option *DMF_FatalNetErrors* is applicable within the function **AppMain** of the client application as well as on the server side by recompiling **startup.c** (with the additional compiler define XXX_FATALNETERRORS) and relinking the server application This option is intended primarily for use cases where the use of the command line option **-IDMfataneterrors** is impossible. *DMF_FatalNetErrors* takes precedence over **-IDMfatalneterrors**. **See Also** Command line option **-IDMfatalneterrors** in chapter “Command Line Options” of manual “Development Environment”

Return Value

TRUE	DM has successfully executed the initialization.
FALSE	DM was unable to initialize its internal structures correctly. In this case the application should be canceled, because it cannot continue operating correctly.

When calling this function you should keep in mind that all arguments which the application cannot process in the function have to be transferred. DM then removes the arguments it needs from this argument vector and passes the remaining vector on to the window system. The window system is also initialized with this function. This behavior enables the user to start the application with certain options designed for the relevant window system.

Example

Start of a standard main program:

```
int DML_c AppMain (argc, argv)
int argc;
char far * far *argv;
{
  DM_ID dialogID;

  /* Initializtion
  if (!DM_Initialize (&argc, argv, 0))
    return (1);
```

3.38 DM_InitMSW

This function has to be called in the Microsoft Windows version of DM, if the "startup.obj" has been replaced by an individual starting program. This function assumes the task of parsing the command line and saves all important information with which a Windows program can be supplied as parameter on starting.

```
char far * far*  DML_default DM_EXPORT DM_InitMSW
(
  HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpCmdLine,
  int *argc
)
```

Parameters

-> HANDLE hInstance

In this parameter a pointer to the current application instance is transferred. Each Windows program receives this value as parameter.

-> HANDLE hPrevInstance

In this parameter a pointer is transferred to the previous application instance. Every Windows program receives this value.

<-> LPSTR lpCmdLine

This is the actual command line the user has specified on starting the program. This function separates this line into individual parameters and returns the line as return value.

-> int *argc

In this parameter the number of parameters the function has formed out of the command line is returned.

Return Value

This function provides an array with strings as result. The individual strings represent the separated command line.

Example

Standard-startup file of DM for MS Windows:

```
int PASCAL WinMain __4(
(HANDLE, hInstance),
(HANDLE, hPrevInstance),
(LPSTR, lpCmdLine),
(int, nCmdShow))
{
  int argc;
```

```
  char far * far *argv;

  argv = DM_InitMSW(hInstance, hPrevInstance, lpCmdLine,
    &argc);

  if (argv)
    return (main (argc, argv));

  return (-1);
}
```

3.39 DM_InputHandler

By using this function, additional messages can be received and processed in the application. The kind of messages as well as the type of message reception depend on the window system. The function itself is window-system-dependent and is introduced in the following chapters.

3.39.1 Microsoft Windows

In these window systems the input handler helps to react to any messages sent to objects. These messages are defined by the relevant window system.

```
HWND DML_default DM_EXPORT DM_InputHandler
(
  DM_InputHandlerProc funcp,
  FPTR funcarg,
  DM_UInt msg,
  DM_UInt iomode,
  DM_UInt operation,
  DM_Options options
)
```

Parameters

-> DM_InputHandlerProc funcp

This parameter specifies a pointer to the function to be called as soon as the indicated message arrives.

-> FPTR funcarg

This parameter transfers a pointer to a structure defined by the application. This structure is then passed on to the application on calling the installed input-handler function. DM stores this parameter only, it does not interpret it.

-> DM_UInt msg

This parameter specifies the message on whose arrival the indicated function is to be called. Here all messages defined in the window system can be specified.

-> DM_UInt iomode

This parameter informs Dialog Manager on how the installed input handler is to be interpreted. The following constants are defined:

iomode	Meaning
DMF_ModeAny	This option is not permitted if an input handler is to be installed by means of the DM_InputHandler function.

iomode	Meaning
DMF_ModeMsgNotify	This option means that the installed input handler will only be informed if the specified message has arrived. Dialog Manager takes on the actual handling of the message.
DMF_ModeMsgManage	This option means that the installed input handler completely takes on the processing of the specified message. Dialog Manager only passes on this message to the specified function, but does not process them.

-> DM_UInt operation

This parameter informs the function about the operation to be actually executed. The relevant constants are defined here:

operation	Meaning
DMF_RegisterHandler	Using this value, you can install an input handler in DM.
DMF_WithdrawHandler	Using this value, you can de-install an input handler previously installed.
DMF_DisableHandler	Using this value, an input handler is deactivated temporarily.
DMF_EnableHandler	Using this value, a deactivated input handler is reactivated.

-> DM_Options options

Usually, you have to indicate 0 for this parameter. If exactly one handler is to be de-installed, you can control via one additional option that all function arguments are compared and that the handler is de-installed.

Option	Meaning
DMF_Checkfuncarg	This option means that all function arguments are to be used for searching the handler and that exactly the same handler is to be installed. This can be useful if several handlers have been installed and if one of these is to be de-installed.

Return Value

If the return value is not equal HWND 0, the HWND of the object to which the input handler has been linked is returned in this parameter. A HWND 0 means that the input handler could not be installed.

Example

The following example for the PC platforms shows how an asynchronous function

```
gethostbyname
```

can be implemented by means of this function.

```
/* Definition of static variables */
static HWND TcpWinHwnd = (HWND) 0;
    /* Definition of the desired message number */
static UINT TcpWinMsgGetXByY = 0x6FE1;
/*
** The following function enables you to calculate
** a free message number. This number takes on
** DM_ProposeInputHandlerArgs. The result will be returned.
*/
static boolean TcpWin_CheckAvail __1(
(TranspDescr *, tpdesc))
{
  DM_InputHandlerArgs InpArgs;

  /* provides WinHandle of the invisible window to which the
  ** input handler is attached and returns free message.
*/
  InpArgs.hwnd = (HWND) 0;
  InpArgs.message = TcpWinMsgGetXByY;
  InpArgs.wParam = (WPARAM) 0;
  InpArgs.lParam = (LPARAM) 0;
  InpArgs.mresult = (LRESULT) 0;
  InpArgs.userdata = (FPTR) 0;

  DM_ProposeInputHandlerArgs (&InpArgs, DMF_DontTrace);
  TcpWinHwnd = InpArgs.hwnd;
  TcpWinMsgGetXByY = InpArgs.message;

}

/*
** The following function is the actual handler function.
** It takes the desired data from the corresponding
** structures.
*/
static DM_Boolean DML_default DM_CALLBACK TcpWinGetXByYHandler __3(DM_
InputHandlerArgs far *, pInpArgs),
  (DM_UInt, msg),
  (DM_UInt, iomode))
  {
    if (msg == TcpWinMsgGetXByY)
    {
      if (   (WSAGETASYNCERROR (pInpArgs->lParam) == 0)
      || (WSAGETASYNCERROR (pInpArgs->lParam) == WSABASEERR))
      {
        TcpWinHostent = (struct hostent FAR *) (FPTR)
```

```
          TcpWinBuffer;
       }
       return (FALSE);
      }
      return (TRUE);
  }

/*
** This function has the control. It enables you to calculate
** the free message, it installs the input handler and then
** calls the asynchronous function gethostbyname.
*/

static struct hostent FAR * TcpWin_gethostbyname __1(
(const char FAR *, name))
{
  HANDLE h = WSAAsyncGetHostByName (TcpWinHwnd,
    TcpWinMsgGetXByY,name,TcpWinBuffer,MAXGETHOSTSTRUCT);
  TcpWinHostent = (struct hostent FAR *) 0;

  if ((h != (HANDLE) 0)
  && (DM_InputHandler (TcpWinGetXByYHandler, (FPTR) 0,
        TcpWinMsgGetXByY, DMF_ModeMsgNotify,
        DMF_RegisterHandler, DMF_DontTrace)
    != (HWND) 0)
    && DM_WaitForInput (TcpWinMsgGetXByY, 0,
    DMF_IgnoreExtEvent | DMF_DontTrace))
    {
      DM_InputHandler (TcpWinGetXByYHandler, (FPTR) 0,
        TcpWinMsgGetXByY, DMF_ModeMsgNotify,
        DMF_WithdrawHandler,
        DMF_DontTrace | DMF_CheckFuncarg);
    }

    return (TcpWinHostent);
}
```

3.39.2 Motif

In this window system the input handler is used to be able to wait for events on any file descriptors. Thus, for example messages coming from other processes can be processed.

```
DM_Boolean DML_default DM_EXPORT DM_InputHandler
(
  DM_InputHandlerProc funcp,
  FPTR funcarg,
  int FileDescriptor,
  DM_UInt iomode,
```

```
  DM_UInt operation,
  DM_Options options
)
```

Parameters

-> DM_InputHandlerProc funcp

This parameter passes on a pointer to the function which is to be called on arrival of a message.

-> FPTR funcarg

In this parameter a pointer to one of the structures defined by the application is passed on. This structure is then transferred to the application on calling the installed input-handler function. Dialog Manager stores this parameter only, without interpreting it itself.

-> int FileDescriptor

In this parameter a file descriptor on which arriving messages are waited for is passed on. These messages are then transferred to the installed input handler.

-> DM_UInt iomode

This parameter informs DM on how to interpret the installed input handler. To do so, the following constants are defined

iomode	Meaning
DMF_ModeAny	This option is not permitted if an input handler is to be installed by means of the DM_InputHandler function.
DMF_ModeRead	This option informs DM that the specified input handler is to be called if something has been read on the specified file descriptor.
DMF_ModeWrite	This option informs DM that the specified input handler is to be called if it can be read from the specified file descriptor.

-> DM_UInt operation

This parameter informs the function about the operations actually to be executed. To do so, the following constants are defined:

Operation	Meaning
DMF_RegisterHandler	Using this value, you can install an input handler in DM.
DMF_WithdrawHandler	Using this value, you can de-install an input handler previously installed.
DMF_DisableHandler	Using this value, an input handler is deactivated temporarily.
DMF_EnableHandler	Using this value, a deactivated input handler is reactivated.

-> DM_Options options

This parameter is reserved for future versions. Please specify with 0.

Return Value

TRUE	The input handler has been installed successfully.
FALSE	The input handler could not be installed.

3.40 DM_InstallNlsHandler

This function installs another function which provides texts from message catalogs.

```
void DML_default DM_EXPORT DM_InstallNlsHandler
(
  DM_NlsFunc func
)
```

Parameters

-> DM_NlsFunc func

Address of the function that provides texts. It has the following format:

```
DM_String (DML_default DM_CALLBACK *DM_NlsFunc)
          (DM_Int4 msgno, int *codepage);
```

The first parameter is the number of the text desired in the row (1 - 65535); the second parameter is a pointer to the desired code page (e.g. CP_ascii, CP_iso8859, etc.).

The function returns the text assigned to the number. If no appropriate text is available, the function may return a *NULL* pointer.

If the function returns the text in a different code page, the function has to save the used code page in **codepage*.

See Also

Resource text

3.41 DM_InstallWSINetHandler

This function of the DM interface registers the user-defined functions in which the encryption software is called.

This function must be called in **AppMain** before **DM_Initialize**.

Result value and parameters of the user-defined functions are predefined.

```
DM_Boolean DML_default DM_EXPORT DM_InstallWSINetHandler
(
  DM_WSINetFunctions *wsinetfunctions,
  DM_Uint Operation,
  DM_Options Options
)
```

parameter

-> DM_WSINetFunctions *wsinetfunctions

Structure containing the function pointers to the user-defined functions. It has the following form:

```
DM_WSINetFunctions
{
  DMAcceptProc,
  DM_SessionProc,
  DM_ShutDownProc,
  DM_OpenProc,
  DM_CloseProc,
  DM_SendProc,
  DM_ExistsMessageProc,
  DM_RecvProc,
  DM_FreeWarningProc
}
```

The individual function types have the following form:

```
int <name of DM_AcceptProc function>
    (int serverfd, void *cliaddr, void *addrlen, char *message)
void * <name of DM_SessionProc function>
    (int clientfd, void *support, char *message)
void <name of DM_ShutDownProc function>()
int <name of DM_OpenProc function
    (int *port, void **supportptr, char *message, char **warning)
int <name of DM_CloseProc function>
    (void *connptr, int *clientfd, char *message)
int <name of DM_SendProc function>
    (void *connptr, char *buffer, int length, char message)
int <name of DM_ExistsMessageProc function>
    (void *connptr, char *message)
int <name of DM_Recv function>
    (void *connptr, char *buffer, int count, char *message)
```

```
void <name of DM_FreeWarningProc function>
    (char *warning)
```

-> DM_Uint Operation

One of the two predefined constants:

1. *DMF_RegisterHandler* to register the user defined functions.
2. *DMF_WithdrawHandler* to deregister the user-defined functions.

-> DM_Options Options

Is not used and is to be preassigned with 0.

Return value

TRUE	For *DMF_RegisterHandler*: functions could be registered. For *DMF_WithdrawHandler*: return value always immer = TRUE.
FALSE	For *DMF_RegisterHandler*: functions could not be registered.

3.41.1 User defined functions

Description of the individual functions:

DM_AcceptProc

Accepts a connection to a client.

DM_SessionProc

Returns the descriptor of the client as return value void *.

Sets parameters for the client session.

DM_ShutDownProc

Performs closing actions when closing the connection.

DM_OpenProc

Configures the socket and opens it.

DM_CloseProc

Closes the connection.

DM_SendProc

Sends a message to the client.

DM_ExistsMessage

Checks if the client has sent a message.

DM_RecvProc

Reads a message sent by the client.

DM_FreeWarningProc

Frees memory allocated in **DM_OpenProc** for the *warning* parameter.

Order of the call is usually

1. DM_OpenProc function
2. DM_FreeWarningProc function
3. DM_AcceptProc function
4. DM_SessionProc function
5. DM_SendProc function, DM_Recv function, DM_ExistsMessageProc function alternating
6. DM_CloseProc function
7. DM_ShutDownProc function

If custom encryption is to be implemented using the user-defined functions, the order of calls specified by the IDM must be taken into account when using the parameters as input and output.

3.42 DM_LoadDialog

Using this function dialogs can be loaded into the application.

```
DM_ID DML_default DM_EXPORT DM_LoadDialog
(
  DM_String path,
  DM_Options options
)
```

Parameters

-> DM_String path

Specifies the file to be loaded by means of a path. As is usual with all file accesses, the indicated name has to have the following structure:

```
environment variable:name of dialog file
```

The preceding environment variable serves as path on which the dialog file is to be searched.

-> DM_Options options

This parameter is reserved for future versions. Please specify with 0.

Return Value

0	The indicated file is not an error-free DM file, or the file was not found.
!= 0	Identifier of the dialog loaded by DM.

The DM has the ability to load the dialog data from an ASCII file (dialog script) as well as from a binary file. The main advantage of a binary file is that it can be loaded in a significantly shorter time, because no internal checks have to be carried out. A binary file is generated with this command:

```
idm +writebin <name of binary file> <name of ASCII file>
```

Example

Excerpt from an AppMain function:

```
  /*
  ** Loading the dialog. Standard name is given in the
  ** program, and can be overwritten via the command line.
*/
  switch(argc)
  {
    case 1:
        dialogID = DM_LoadDialog (dialogfile, 0);
    break;
    case 2:
        dialogfile = argv[1];
        dialogID = DM_LoadDialog (dialogfile, 0);
```

```
  break;
  default:
      DM_TraceMessage("Zuviele Argumente ",
          DMF_LogFile | DMF_InhibitTag);
  return(0);
  break;
}

if (!dialogID)
{
  DM_TraceMessage("%s: Could not load dialog \"%s\"",
          DMF_LogFile | DMF_InhibitTag | DMF_Printf,
          argv[0], dialogfile);
  return(1);
}
```

See Also

Built-in function load in manual “Rule Language”

3.43 DM_LoadProfile

Using this function, you can read variables which can be changed by the end user and processed by the DM. This facility enables end users to influence the dialog behavior when the dialog source or the DM are not available.

```
DM_Boolean DML_default DM_EXPORT DM_LoadProfile
(
  DM_ID dialog,
  DM_String filename,
  DM_Options options
)
```

Parameters

-> DM_ID dialog

This is the identifier of the dialog for which the specified profile is to be read.

-> DM_String filename

This name denotes the profile file.

As is usual with all file accesses, the specified name may have the following structure:

```
environment variable:name of dialog file
```

The preceding environment variable serves as path on which the dialog file is to be searched.

The dialog description looks, for example, as follows:

```
config variable integer HUGO;
```

This variable now is to be set by means of the profile. The file then looks like this:

```
HUGO := 5;
```

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	File could be read.
FALSE	File could not be read.

Example

Dialog File

```
dialog YourDialog
{
    .xraster 10;
    .yraster 16;
}
```

```
config variable string WindowText := "Sorry no profile";
config variable integer WindowXPos := 5;
config variable integer WindowYPos := 5;

window W1
{
    .xleft 4;
    .ytop 6;
    .width 25;
    .title "Testwindow";
    .visible false;
    .xraster 10;
    .yraster 16;
    .posraster true;
    .sizeraster true;
    child pushbutton End

    {
    .xleft 7;
    .width 9;
    .ytop 6;
    .height 2;
    .text "End";
    .visible true;
    .sizeraster true;
    }
}
on End select
{
    exit ();
}

on dialog start
{
    W1.xleft :=WindowXPos;
    W1.ytop :=WindowYPos;
    W1.title :=WindowText;
    W1.visible :=true;
}
```

Profile

```
WindowXPos:=10;
WindowYPos:=5;
WindowText:="Out Of Profile";
```

Note

This profile can only be loaded in the application via DM_LoadProfile or it can be loaded in the simulator via **-profile <filename>**.

Furthermore, you have to note that DM_LoadProfile has to be called before DM_StartDialog and DM_EventLoop.

See also

C function DM_SaveProfile

Built-in function loadprofile

3.44 DM_Malloc

Using this function, you can allocate memory. The allocation is carried out according to the used operating system with the relevant available functions.

Memory allocated with DM_Malloc may be released with DM_Free or modified with DM_Realloc only!

```
DM_Pointer DML_default DM_EXPORT DM_Malloc
(
  DM_UInt4 size
)
```

Parameters

-> DM_UInt4 size

This parameter specifies the size of the memory to be newly allocated.

Warning

This memory capacity must not be **> 64 KByte**, if the application is to run with Microsoft Windows!

Return Value

Pointer to the memory allocated. If the memory could not be allocated, the *NULL* pointer is returned.

Example

Memory is to be provided for a string.

```
char * string;

if ((string = DM_Malloc(20)))
    strcpy (string, "1234567");
```

3.45 DM_NetHandler

With the help of this function a NetHandler can be registered. A NetHandler is a user-defined function which is called by the IDM with the contents of network packets immediately before sending and immediately after receiving these packets.

The Distributed Dialog Manager sends all data as plain text over the network, i.e. the data can be read by anyone. In order to protect confidential data against unauthorized persons the IDM user needs a possibility to encrypt the data being sent over the network. This possibility is provided by the registration of NetHandlers, where the IDM user can carry out encryption.

Apart from encryption, other applications of NetHandlers are also possible.

```
void DML_default DM_EXPORT DM_NetHandler
(
  DM_NetHandlerProc NetHandler,
  DM_UInt           Operation,
  DM_Options        Options
)
```

Parameter

-> DM_NetHanderProc NetHandler

Function pointer to the user-defined handler. The handler must have the following format:

```
void DML_default DM_Callback <ProcName> (reason)
DM_NetInfo far *reason;
{
  /* User code */
}
```

-> DM_UInt Operation

This parameter can be assigned the following values:

- *DMF_RegisterHandler*
  Register the handler
- *DMF_WithdrawHandler*
  Deregister the handler
- *DMF_EnableHandler*
  Activate the handler (not yet implemented)
- *DMF_DisableHandler*
  Deactivate the handler (not yet implemented)

NetHandlers are automatically activated after their registration. It is possible to install several NetHandlers, yet each handler can only be installed once.

-> DM_Options Options

Is not used, has to be initialized with *0*.

Please note that such NetHandlers, which modify data packets, absolutely have to be registered with the sending and receiving processes synchronously. If, for example, one process is sending encrypted data and the other process is expecting unencrypted data this may cause the process or even the entire system to crash.

The registered handlers are called in the inverse order they had been registered. There is only one exception that will be explained later on.

When calling a NetHandler, it is passed a **DM_NetInfo** structure that contains all necessary information and the data of the network packet:

```
struct {
  char        *data;   /* Data packet */
  DM_Integer  length;  /* Data size in bytes */
  DM_Integer  action;  /* Communication process */
  DM_Integer  error;   /* Error number */
  int         socket;  /* Communication interface */
  DM_ID       applID;  /* ObjectId of the application */
} DM_NetInfo;
```

In the structure element *action*, one of these five constants can be passed to the handler:

- *DM_NET_SEND*
- *DM_NET_RECEIVE*
- *DM_NET_CONNECT*
- *DM_NET_MESSAGE*
- *DM_NET_ERROR*
- *DM_NET_MESSAGE*
- *DM_NET_CONNECTMESSAGE*

With *DM_NET_SEND* the transferred data packet is to be sent and with *DM_NET_RECEIVE* it has been received. The content of the data packet can be manipulated in these operations, e.g. it can be encrypted or decrypted.

The constant *DM_NET_CONNECT* indicates that a connection is being established. Yet, the content of the data packet must not be changed, otherwise the establishment of the connection will fail.

DM_NET_CONNECTMESSAGE indicates the STDOUT output for establishing a connection to the application page started by .exec. This must be forwarded to IDM unchanged.

DM_NET_MESSAGE indicates the STDOUT output of the application page started by .exec. The output is passed line by line to the NetHandler (exception when the socket is closed). Attention: First a 0 byte is sent (rsh protocol ?) and the IDM sends a newline before the connection is established. **Attention:** 0 bytes remain as 0 bytes. The length of the string should be strictly observed.

If the constant *DM_NET_RECEIVE* is set, the registered NetHandlers are invoked in the same order as they had been registered. This case is the exception mentioned above. This is necessary because the manipulations of the data by several handlers before sending must be undone in reverse order

when receiving the data. If, for instance, first a logging and then an encrypting handler is called, then the data must be decrypted by the encrypting handler first before it can be recorded by the logging handler.

Data manipulations carried out with *DM_NET_SEND* and *DM_NET_RECEIVE* have to be inverse to each other. This means that data that has been passed through the handler by the receiver with *DM_NET_RECEIVE*, must be identical with the data that the sender previously passed through the handler with *DM_NET_SEND*.

Here, the data packets have to be regarded as a stream, because what the sender transmits to the handler as one packet does not necessarily reach the recipient as one packet.The TCP/IP implementation or other network components may have interposed buffers in-between. Consequently, manipulations of the packet contents must be independent of the position within the packet.

The *data* pointer to the contents of the network packet, must not be changed by the handler. Re-allocation of *data* at the same address is also not allowed. Thus data packets cannot be enlarged which is why decompressing compressed data is not possible.

Example

```
/* NetHandler to encrypt data
   through bit inversion */
void DML_default DM_CALLBACK MyNetHandler
  __1((DM_NetInfo *, info)) {
  switch (info->action) {
    case DM_NET_SEND:
    case DM_NET_RECEIVE: {
      int ZaVa;
      for(ZaVa=0;ZaVa<info->length;ZaVa++)
        (info->data)[ZaVa]=~(info->data)[ZaVa];
        break;
    }
  }
}
```

For example, one can register the handler with client and server before the attachment of C functions to the IDM.

```
if (!DM_NetHandler(MyNetHandler,DMF_RegisterHandler,0))
  DM_TraceMessage("NetHandler registering error", 0);
```

3.46 DM_OpenBox

With this function, a specified ***messagebox*** or **dialogbox** (***window*** with attribute *.dialogbox* = *true*) can be opened. The program waits until the user has closed this messagebox or dialogbox.

Note

When using functions that have ***records*** as parameters, please refer to the notes in chapter “Handling of String Parameters” and the chapter “Note for Using DM Functions” in manual “C Interface - Basics”.

```
DM_Boolean DM_OpenBox
(
  DM_ID       objectID,
  DM_ID       parentID,
  DM_Value   *retval,
  DM_Options  options
)
```

Parameters

-> DM_ID objectID

This parameter specifies the ***messagebox*** or **dialogbox** to be opened.

-> DM_ID parentID

This parameter specifies the ***window*** in which the ***messagebox*** should appear. The parameter may be ignored. If this parameter is specified, a ***window*** or *NULL* must be specified.

If the window system supports it, the ***messagebox*** is displayed centered in the parent window. Otherwise, the position is determined by the window system itself (e.g. screen center).

-> DM_Value *retval

Contains the return value of the respective object after closing the ***messagebox*** or **dialogbox**:

- For ***messageboxes*** the number of the pressed button. There are the following definitions for this:
  - *MB_abort*
  - *MB_cancel*
  - *MB_ignore*
  - *MB_no*
  - *MB_ok*
  - *MB_retry*
  - *MB_yes*
- For **dialogboxes**, the value defined in the closequery function.

-> DM_Options options

This parameter is currently not used and must therefore be set to *0*.

Return Value

The return value indicates whether the ***messagebox*** or **dialogbox** could be opened.

See Also

C function DM_QueryBox

Object Messagebox

Built-in function querybox in manual “Rule Language”

3.47 DM_ParsePath

With this function, the identifier of an object can be queried if more than one dialog is loaded and the searched object is not in the first loaded dialog.

```
DM_ID DML_default DM_EXPORT DM_ParsePath
(
  DM_ID      dialogid,
  DM_ID      rootid,
  DM_String  path,
  DM_UInt    idx,
  DM_Options options
)
```

Parameters

-> DM_ID dialogid

This is the identifier of the dialog in which to search for the object.

-> DM_ID rootid

This parameter controls from which object the IDM starts the search for the object. There are the following options:

» *rootid = 0*
The IDM searches for the specified object in the entire dialog definition.
This is the usual case. In this way the identifiers of rules, functions, variables and resources can be queried too.

» *rootid != 0*
From the specified object, the IDM searches only on the next lower hierarchy level; lower hierarchy levels are not searched.
This procedure is only appropriate if an object name occurs more than once in a dialog.

-> DM_String path

This path defines the searched object. It must unambiguously describe an object.

If the object name exists only once within the dialog, then the indication of the name suffices to obtain the desired reference. If the object name is not unique, the object must be described by a point separated path of object names.

-> DM_UInt idx

This parameter specifies which occurrence of the object with the specified name shall be searched.

The counter starts at *0*. To find the first occurrence of an object, enter *0* here.

-> DM_Options options

This parameter is currently not used.

Return Value

0 The searched object was not found or the name is not unique.
That is, there are none or more objects with the given name.

!= 0 Identifier of the searched object.

Annotation

If *"setup"* is specified as *path* and both *dialogid* and *rootid* are *0*, then the ***setup*** object is returned.

Example

```
void DML_default DM_ENTRY OkButtonCallback __1((DM_ID, dialogID))
{
  DM_ID ID1;
  DM_ID ID2;

  /* Query an object in the dialog globally */
  ID1 = DM_ParsePath(dialogID, 0, "FirstObject", 0, 0);

  /* Query using a path */
  ID2 = DM_ParsePath(dialogID, 0, "FirstObject.Child1", 0, 0);
}
```

See Also

Built-in function parsepath in manual “Rule Language”

3.48 DM_PathToID

Attention

The function **DM_PathToID** is deprecated and is only supported for compatibility reasons. Please use DM_ParsePath instead.

Using this function, the known external identifier of an object is changed into an internal identifier. This internal identifier is not changed during program execution, so you do not have to query the ID of a frequently used object on every access.

```
DM_ID DML_default DM_EXPORT DM_PathToID
(
  DM_ID rootid,
  DM_String path
)
```

Parameters

-> DM_ID rootid

This parameter controls from which object the DM is to begin searching the object you desire. The following options are available:

- *rootid = 0*
  DM searches the entire dialog definition for the object specified. This is the usual option. The internal names or identifiers of rules, functions, variables and resources can also be queried in this way.
- *rootid != 0*
  DM is to search the object desired only on the next subordinate hierarchy level from the object specified.
  This method is appropriate only if an object identifier occurs more than once in a dialog. Rules, functions, variables and resources can **not** be queried.

-> DM_String path

Using this path, you can describe the searched object. This path has to describe an object unambiguously. If the object name exists only once within a dialog, the specification of the name is enough to receive the desired reference. If the object name is unambiguous, the object has to be described by means of a path of object names, separated by a dot.

Return Value

0	The searched object could not be found or the name is not unique.
!= 0	Identifier of the searched object.

By using the object identifier you have received in the above-mentioned way you can now access its attributes.

Example

```
DM_Value value;
DM_ID id;

/* The file could be opened, enable the other objects */
value.type = DT_boolean;
value.value.boolean = TRUE;

/* Get the id of the edittext "Actives" */
if ((id=DM_PathToID(0, "Actives")))
  /* Change the object to sensitive */
  DM_SetValue(id, AT_sensitive, 0, &value, DMF_ShipEvent);
```

See Also

Function DM_ParsePath

Built-in function parsepath in manual “Rule Language”

3.49 DM_PictureHandler

With a custom graphics handler (GFX handler), images can be loaded and displayed in graphic formats not supported by the IDM. In this case whenever an image has to be loaded (e.g. because it has been attached to the application via a tile resource), the IDM first calls the registered graphics handlers. If one of these handlers can load the image, it passes the image data to the IDM and no further graphics handler is called. If no graphics handler could load the image, the IDM itself attempts to parse the image. Then the behavior is like no graphics handlers were used.

Graphics handlers are registered and unregistered with the IDM by means of the DM_PictureReaderHandler function.

A graphics handler has to be defined like this:

```
DM_Boolean DML_default DM_CALLBACK <ProcName>
(
  DM_PicInfo * pic
)
{
  /* custom code */
}
```

When called, a pointer to the structure **DM_PicInfo** is passed to the graphics handler. This structure contains all the information about an image that is to be loaded or whose memory is to be freed.

```
typedef struct {
  DM_Integer  struct_size;  // size of the structure to verify the version
  DM_UInt1    task;         // task for the GFX handler
  DM_String   fname;        // file path of the image
  DM_String   name;         // image name as indicated at tile or image
#if defined(WSIWIN)
  DM_UInt2    width;        // width of the image
  DM_UInt2    height;       // height of the image
  DM_UInt2    type;         // image type
  HANDLE      palette;      // handle for the color palette
  HANDLE      image;        // handle for the image
  HANDLE      trans_mask;   // handle for the tansparency mask
#endif
#if defined(MOTIF) || defined(QT)
  DM_UInt2    type;         // image type
  DM_Pointer  image;        // pointer to the image data
  DM_Pointer  trans_mask;   // pointer to the transparency mask
#endif
  DM_Integer  trans_color;  // index of transparent color
} DM_PicInfo;
```

Meaning of elements

DM_Integer struct_size

Here the size of the structure is passed. This can be used in the graphics handler to check with **sizeof()** whether the size of the passed structure matches the structure definition used.

DM_UInt1 task

Defines what the graphics handler should do. There are two possible tasks:

- *DM_GFX_TASK_LOAD*
  Load an image.
  The graphics handler needs to load an image file with the file name *fname* and set the following items of the **DM_PicInfo** structure.
  - *width* (required on MICROSOFT WINDOWS; otherwise contained in the image data)
  - *height* (required on MICROSOFT WINDOWS; otherwise contained in the image data)
  - *type* (required)
  - *palette* (optional; MICROSOFT WINDOWS only)
  - *image* (required)
  - *trans_mask* (optional; supported since IDM version A.05.02.e)
  - *trans_color* (optional; currently not evaluated)
- *DM_GFX_TASK_UNLOAD*
  Free the memory of the image data.
  In this case, the following structure items are set so that the graphics handler can free the memory used by these objects:
  - *palette* (if existent; MICROSOFT WINDOWS only)
  - *image*
  - *trans_mask* (if existent; supported since IDM version A.05.02.e)

DM_String fname

Only valid for *task* == *DM_GFX_TASK_LOAD*.

This item specifies the path and file name of the image to be loaded.

The IDM checks whether the image can be loaded. If so, *fname* will contain the resolved file name of the image.

If the image cannot be loaded, i.e. the file name cannot be resolved, then *name* == *fname*.

DM_String name

Only valid for *task* == *DM_GFX_TASK_LOAD*.

The item contains the unresolved file name of the image as specified at the ***tile*** resource or ***image*** object.

The structure item exists since IDM version A.05.02.e.

DM_Integer trans_color

Here the color to be displayed transparently can be passed.

The item is currently not evaluated.

Assignment of the Structure Items on Microsoft Windows

DM_UInt2 width
DM_UInt2 height

Only valid for *task == DM_GFX_TASK_LOAD*.

In these items the graphics handler must return the width and height of the loaded image.

DM_UInt2 type

In this item the graphics handler must return the type of the handle that is passed in *image*.

Value range

- *DM_GFX_BMP* Windows Bitmap
- *DM_GFX_WMF* Windows Meta File
- *DM_GFX_EMF* Enhanced Meta File
- *DM_GFX_ICO* Windows Icon

HANDLE palette

For *task == DM_GFX_TASK_LOAD*

The graphics handler can pass a color palette (Windows Handle Type *HPALETTE*) to the IDM along with the image. If this item is *NULL*, the system color palette is used.

In general, the use of a separate color palette for each image should be avoided, as color distortions may occur with focus changes on systems with low color depth. Instead, the graphics handler should adjust the colors of an image to the system color palette and thus ensure the color fastness of the image.

For *task == DM_GFX_TASK_UNLOAD*

If a color palette was used, in this element its handle is passed from the IDM to the graphics handler so that the handler can free the memory used for the color palette.

HANDLE image

For *task == DM_GFX_TASK_LOAD*

In this item, the graphics handler has to return the handle of the image it loaded. The type of the handle must match the one given in *type*.

For *task == DM_GFX_TASK_UNLOAD*

In this item the handle of the image, whose memory has to be freed, is passed from the IDM to the graphics handler.

HANDLE trans_mask

This structure item is only evaluated since IDM version A.05.02.e.

For *task* == *DM_GFX_TASK_LOAD*

In this element a transparency mask can be passed. The transparency mask must be a monochrome Windows Bitmap (data type *HBITMAP*) with the same size as the image passed in *image*.

A transparency mask is only supported for the image type *DM_GFX_BMP*, for other image types it is ignored.

For *task* == *DM_GFX_TASK_UNLOAD*

If a transparency mask was used, in this element its handle is passed from the IDM to the graphics handler so that the handler can free the memory used for the transparency mask.

Assignment of the Structure Items on Motif and Qt

DM_UInt2 type

In this item the graphics handler must return the type of the image that is passed in *image*.

Value Range with IDM FOR MOTIF

- *DM_GFX_XIMAGE* XImage image
- *DM_GFX_PIXMAP* PixMap image (supported since IDM version A.05.02.e)

Value Range with IDM FOR QT

- *DM_GFX_QPIXMAP*
- *DM_GFX_QIMAGE*
- *DM_GFX_QICON*

DM_Pointer image

For *task* == *DM_GFX_TASK_LOAD*

In this element, the graphics handler has to pass a pointer to a PixMap image or an XImage structure with the image data corresponding to the *type* element.

The XImage structure can be generated on X Windows with the Xlib function **XCreateImage()**. This requires information about current screen settings (e.g. Display, Visual, Screen, Depth). This data can be queried with the interface function DM_GetToolkitData.

For *task* == *DM_GFX_TASK_UNLOAD*

In this item a pointer to the image data, whose memory has to be freed, is passed from the IDM sto the graphics handler.

DM_Pointer trans_mask

This structure item is only evaluated since IDM version A.05.02.e.

For *task == DM_GFX_TASK_LOAD*

In this element a pointer to a transparency mask can be passed. On MOTIF the transparency mask must be a PixMap image, on QT its data type must correspond to the image type given in *type*. The transparency mask must have the same size as the image passed in *image*.

For *task == DM_GFX_TASK_UNLOAD*

If a transparency mask was used, in this element a pointer to it is passed from the IDM to the graphics handler so that the handler can free the memory used for the transparency mask.

Return value

DM_TRUE	The image could be loaded.
DM_FALSE	In case of an error.

Remarks

Graphic handlers are system-dependent. As the **DM_PicInfo** structure shows, the data formats for returning image data differ depending on the system.

Since a graphics handler must allocate memory to load an image (e.g. for *image*, *palette* and *trans_mask*), it must also be charged with freeing this memory. Therefore the IDM calls the graphics handlers with *task == DM_GFX_TASK_LOAD* when an image is needed. If the image is no longer needed, the graphics handlers are called with *task == DM_GFX_TASK_UNLOAD* so that they can free the allocated memory.

The first graphics handler that frees the memory returns *DM_TRUE*, so no further handlers are called afterward. This does not necessarily need to be the same handler that loaded the image.

The IDM does not link an image to a graphics handler responsible for loading and releasing the image. If, for instance, all graphics handlers allocate the memory in the same way, each handler can also free the memory of any other handler. This means that the first handler called immediately frees the memory.

Example

A typical GFX handler basically has the following structure:

```
DM_Boolean DML_default DM_CALLBACK MyGfxHandler __1((DM_PicInfo *, pic))
{
  /* first determine what to do */
  if (pic->task == DM_GFX_TASK_LOAD) {
    /* the actual loading is done in LoadPicture_... */

#if defined(WIN32)
    /* load image (Microsoft Windows) */
    pic->image = LoadPicture_Win32 (pic->fname);
#endif
#if defined(MOTIF) || defined(QT)
```

```
    /* load image (XWindows) */
    pic->image = LoadPicture_X (pic->fname);
#endif

    if (pic->image) {
      /* write return values to pic */

#if defined(WIN32)
      pic->type    = DM_GFX_...;
      pic->palette = handle to pallet, if wanted;
      pic->width   = width of the image;
      pic->height  = height of the image;
#endif
#if defined(MOTIF) || defined(QT)
      pic->type = DM_GFX_...;
#endif

      return DM_TRUE;  /* success */
    } else {
      /* image could not be loaded, maybe another GFX handler
         or the IDM graphics parser will be more successful */
      return DM_FALSE;
    }
  } else {
    /* free memory */

#if defined(WIN32)
    if (pic->image)       /* loaded here */
      DeleteObject(pic->image);
    if (pic->palette)     /*  created here */
      DeleteObject(pic->palette);
#endif
#if defined(MOTIF) || defined(QT)
    if (pic->image)       /* loaded here */
      XDestroyImage((XImage *)pic->image);
    if (pic->trans_mask)  /*  created here */
      free(pic->trans_mask);
#endif

    return DM_TRUE;
  }
}
```

The GFX handler should be registered before loading image files and thus usually before loading the dialog:

```
if (!DM_PictureReaderHandler (MyGfxHandler, DMF_RegisterHandler, 0))
  DM_TraceMessage("GFX-Handler registering error", 0);
```

```
...
/* load dialog */
```

See also

Function DM_PictureReaderHandler

3.50 DM_PictureReaderHandler

This function is used to register and manage custom graphics handlers (GFX handlers) with the IDM. These handlers are invoked by the IDM to load images specified at ***tile*** resources or ***image*** objects.

Custom graphics handlers are described in DM_PictureHandler.

```
DM_Boolean DML_default DM_EXPORT DM_PictureReaderHandler
(
  DM_PictureReaderProc  funcp,
  DM_UInt  operation,
  DM_UInt  options
)
```

Parameters

-> DM_PictureReaderProc funcp

Pointer to a graphics handler for which the action defined in *operation* shall be performed.

The handler must be of this format:

```
DM_Boolean DML_default DM_CALLBACK <ProcName>
(
  DM_PicInfo * pic
)
{
  /* custom code */
}
```

Accordingly, the data type *DM_PictureReaderProc* is defined like this:

```
typedef DM_Boolean (DML_default DM_CALLBACK * DM_PictureReaderProc) __
((DM_PicInfo * pic));
```

-> DM_UInt operation

This parameter defines which action shall be performed for the graphics handler.

Value range

- *DMF_RegisterHandler*
  Registers the handler with the IDM.
- *DMF_WithdrawHandler*
  Unregisters the handler with the IDM.
- *DMF_EnableHandler*
  Activates the handler.
- *DMF_DisableHandler*
  Deactivates the handler.

After a handler is registered, it is activated automatically. More than one graphics handler may be registered, but each handler only once.

-> DM_UInt options

Currently unused; has to be *0*.

Return value

DM_TRUE	The action was completed successfully.
DM_FALSE	In case of an error.

Remarks

The registered graphics handlers are invoked in reverse order as they were registered.

Because graphics handlers allocate memory when loading images, a handler must not be disabled or unregistered until the memory it has allocated is freed by itself or another graphics handler. If all graphics handlers allocate their memory in the same way, a graphics handler can free the memory that has been allocated by other handlers. In that case, the first invoked graphics handler frees the memory, regardless of which handler previously allocated it. Then it is sufficient for one graphics handler to remain registered and active in order to free the memory allocated by graphics handlers.

It is not necessary to unregister graphics handlers when exiting an application.

See also

Function DM_PictureHandler

3.51 DM_ProposeInputHandlerArgs

Using this function, DM can query a message which has not been assigned. This is necessary so that newly assigned message numbers do not overlap with already assigned message numbers.

This function is only available onMICROSOFT WINDOWS.

```
DM_Boolean DML_default DM_EXPORT DM_ProposeInputHandlerArgs
(
  DM_InputHandlerArgs pInputArgs,
  DM_Options options
)
```

Parameters

<-> DM_InputHandlerArgs pInputArgs

On calling this function the elements "msg" or "message" in this parameter are given the message number. DM interprets this value if desired. On returning this element receives the message number which has actually been assigned. It has to be the message defined by the user; it ranges from WM_USER to 0x7FFF.

On returning this function the element hwnd is assigned additionally. In this element the object is returned to which the function has been linked.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	Function was carried out successfully.
FALSE	Function was not carried out successfully, since the specified message was in the wrong area or a *NULL* pointer has been specified as first parameter.

Example

Search for a free message:

```
static boolean TcpWin_CheckAvail __1(
(TranspDescr *, tpdesc))
{
  DM_InputHandlerArgs InpArgs;

  /* provides WinHandle of the invisible window to which
  ** Input-Handler is attached and returns free message.
  */
  InpArgs.hwnd = (HWND) 0;
  InpArgs.message = TcpWinMsgGetXByY;
  InpArgs.wParam = (WPARAM) 0;
  InpArgs.lParam = (LPARAM) 0;
```

```
  InpArgs.mresult = (LRESULT) 0;
  InpArgs.userdata = (FPTR) 0;

  DM_ProposeInputHandlerArgs (&InpArgs, DMF_DontTrace);
  TcpWinHwnd = InpArgs.hwnd;
  TcpWinMsgGetXByY = InpArgs.message;

}
```

3.52 DM_QueryBox

Using this function you can open a specified messagebox. The program waits until the user has closed this box.

```
DM_Enum DML_default DM_EXPORT DM_QueryBox
(
  DM_ID objectID,
  DM_ID parentID,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

This parameter specifies the messagebox to be opened.

-> DM_ID parentID

This parameter specifies the window in which the messagebox is to appear. The parameter may be ignored. If this parameter is indicated, a window or NULL has to be specified.

If the window system allows it, the messagebox is represented centered in the parent window. Otherwise, the position is defined by the window system (e.g. screen center).

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

Number of pressed button. There are the following definitions for it:

- *MB_abort*
- *MB_cancel*
- *MB_ignore*
- *MB_no*
- *MB_ok*
- *MB_retry*
- *MB_yes*

Example

A messagebox is to be opened out of a C function. This C function looks as follows:

```
DM_Boolean DML_default DM_ENTRY MESSBOX __1((DM_ID id))
{
    DM_Value data;
    /* Setting the text to be displayed */
    data.type = DT_string;
    data.value.string = "HELLO TEST";
```

```
    DM_SetValue(id , AT_text , 0 , &data , DMF_ShipEvent );

    /* Setting the title of the messagebox. */
    data.type = DT_string;
    data.value.string = "test output";
    DM_SetValue(id , AT_title , 0 , &data , DMF_ShipEvent );
    /*
    ** Opening messagebox and returning TRUE
    ** on pressing OK
    */
    return ((DM_QueryBox(id,0,0)== MB_ok) ? TRUE : FALSE);
}
```

See Also

C function DM_OpenBox

Object Messagebox

Built-in function querybox in manual “Rule Language”

3.53 DM_QueryError

Using this function the application can query the reason for the failure of the last DM call. The DM returns the number of errors and the errors themselves.

```
DM_UInt DML_default DM_EXPORT DM_QueryError
(
  DM_ErrorCode buffp[ ],
  DM_UInt bufflen,
  DM_Options options
)
```

Parameters

<-> DM_ErrorCode buffp[]

This is an array of error codes. It is filled by the DM, but must be allocated inside the application. If the array is not large enough, the last errors are truncated. To be sure that all error codes stored in DM can be passed to the application, this array should have a size of 32.

-> DM_UInt bufflen

Length of the error code array passed on to the DM.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

The number of valid errors in the array.

Example

Basic routine for error handling:

```
static void QueryError()
{
  DM_ErrorCode errorbuffer[32];
  /* buffer for the errors that occurred */
  register int i;
  int errors;  /* number of errors */

    if ((errors = DM_QueryError(errorbuffer, 32, 0)))
      for (i = 0; i < errors; i++)
        DM_TraceMessage(DM_ErrMsgText(errorbuffer[i], 0), 0);
}
```

3.54 DM_QueueExtEvent

The execution of a rule attached to an external event is registered by the application with the function **DM_QueueExtEvent**.“Registered” means that the rule is not executed immediately, but that the external event is queued and then processed according to the dialog event mechanisms.

The event is processed by the IDM with the usual event processing algorithm which results in calling a rule with the scheme on `<object> extevent <no.>`.

```
DM_Boolean DML_default DM_EXPORT DM_QueueExtEvent
(
  DM_ID       objectID,
  DM_Int4     event_no,
  DM_UInt     argc,
  DM_Value   *argv,
  DM_Options  options
)
```

Parameters

-> DM_ID objectID

This is the identifier of the object to which this external event is to be sent.

-> DM_Int4 event_no

This parameter is the number of the external event to be triggered.

-> DM_Int argc

This parameter transfers the number of parameters (up to 16).

-> DM_Value *argv

This parameter indicates the arguments (up to 16) which the IDM passes to the rule on invocation. This vector must have the length that is specified in the *argc* parameter.

-> DM_Options options

The following values can be specified as options:

Option	Meaning
DMF_DontTrace	This option implies that the function call shall not be traced, if the application is started with the trace option.
DMF_Synchronous	This option can be set, if it is ensured that the function **DM_QueueExtEvent** is called synchronously to the process. In this case, the function internally can work more efficiently. A synchronous call is not given, when the function is called from a signal handler, for instance.

Option	Meaning
DMF_NoCriticalSection	This option prevents the function from using a “critical section” on MICROSOFT WINDOWS.

Return Value

DM_TRUE External event could be put in queue.

DM_FALSE External event could not be put in queue.

Note for Microsoft Windows

Since IDM version A.05.01.a, the function **DM_QueueExtEvent** uses a “critical section” to ensure, that it has been completed before this function or the function**DM_SendEvent** is invoked once more. If one of these functions is called in a situation where a “critical section” is not permitted, the use of the “critical section” can be prevented through the option *DMF_NoCriticalSection*.

Attention

A thread, that carries out one of the two functions must not be canceled.

Example

In a dialog, there shall be a response to a signal of the operating system. The dialog looks as follows:

```
on dialog extevent 4711 (integer ErrCode)
{
  variable string S;
  S := "Dialog has received extevent " + itoa(ErrCode);
  print S;
}
```

The corresponding C program looks like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <IDMuser.h>

DM_ID dialogID;
/* This function has been installed as signal handler */
void handler __1((int, sig))
{
  DM_Value data;

  data.type = DT_integer;
  data.value.integer = sig;
  DM_QueueExtEvent(dialogID, 4711, 1, &data, 0);
```

```
}
```

See Also

C functions **DM_SendEvent**, **DM_SendMethod**

Chapter “External Events” and built-in function sendevent() in manual “Rule Language”

3.55 DM_Realloc

A memory already allocated can be resized with the help of this function. This memory to be modified has to be allocated with DM_Malloc.

```
DM_Pointer DML_default DM_EXPORT DM_Realloc
(
  DM_Pointer ptr,
  DM_UInt4 size
)
```

Parameters

-> DM_Pointer ptr

Pointer to the memory already allocated.

-> DM_UInt4 size

New size of the memory.

Warning

On MS-Windows, this size must not be > 64 KByte!

Return Value

Pointer to the memory allocated. If the memory could not be allocated, the *NULL* pointer is returned.

Example

A string is to be copied in a memory already allocated.

```
char * string;
if ((string = DM_Realloc(string, strlen("12345")+1))
    strcpy(string, "12345");
```

3.56 DM_ResetMultiValue

Using this function you are able to reset various attributes of different DM objects to their model value in one function call. Therefore these functions should be especially used with the distributed DM, since they obviously reduce the network load.

For the permitted attributes of the respective object type please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_ResetMultiValue
(
  DM_MultiValue *values,
  DM_UInt count,
  DM_ID dialogID,
  DM_String pathname,
  DM_Options options
)
```

Parameters

<-> DM_MultiValue *values

List of attributes and objects to be reset. If the element in the structure for the object is set to 0, the object described in the parameter *pathname* is chosen. The list has to have at least the length specified in the parameter *count*.

-> DM_UInt count

Specifies the length of the object-attribute vector indicated in the parameter *values*.

-> DM_ID dialogID

This parameter describes the dialog to which the given objects belong. It has to be specified only if the object identifier the attributes of which are to be reset is not known and thus the name of the object is given in the parameter *pathname*.

-> DM_String pathname

Describes the object the attributes of which are to be reset. It is allocated only if the object's internal identifier is not known yet.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	The attributes were reset successfully
FALSE	At least one attribute could not be reset

Example

Resetting coordinates for several objects:

```
void DML_default DM_ENTRY Reset  __3((DM_ID, o1),
```

```
                    (DM_ID, o2),
                    (DM_ID, o3))
{
    DM_MultiValue val[3];

/* Setting the relevant object ID */
    val[0].object = o1;
    val[1].object = o2;
    val[2].object = o3;

/* Setting the relevant index type */
    val[0].index.type = DT_void;
    val[1].index.type = DT_void;
    val[2].index.type = DT_void;

/* Setting the relevant attributes */
    val[0].attribute = AT_xleft;
    val[1].attribute = AT_width;
    val[2].attribute = AT_xright;

    DM_ResetMultiValue(val, 3, dialogID, (char *)0, 0);
}
```

3.57 DM_ResetValue

This function can be used to reset object attributes to the values of the relevant model or default. For information on the attributes valid for the relevant object type, please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_ResetValue
(
  DM_ID objectID;
  DM_Attribute attr;
  DM_UInt index;
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Describes the object the attribute of which you want to reset.

-> DM_Attribute attr

Describes the attribute to be reset. All attributes permitted are defined in **IDMuser.h**.

-> DM_UInt index

Analyzed only in vector attributes. Describes the index of the searched sub-object (e.g. text in list-box).

-> DM_Options options

Controls whether DM is to trigger rule processing after an attribute has been set successfully.

Option	Meaning
DMF_Inhibit	This option means that the function call is not to trigger any internal events. If this flag is set, you can achieve a better performance in case that a lot of attribute changes are carried out by the application.
DMF_ShipEvent	This option means that the function call is to trigger internal events. Then rules are triggered which have been defined for this object and which react to the changing of the specified attribute.

Return Value

TRUE	The attribute was reset successfully.
FALSE	The attribute could not be reset.

3.58 DM_ResetValueIndex

Using this function, attributes with two indexes can be reset to the value of the corresponding default attribute (with index 0).

For the attributes valid for the relevant object type, please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_ResetValueIndex
(
  DM_ID objectID,
  DM_Attribute attr,
  DM_Value *index,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Describes the object the attribute of which you want to change.

-> DM_Attribute attr

Describes the attribute to be changed. All valid attributes are defined in **IDMuser.h**.

-> DM_Value *index

Specifies the data type of the index (enum, index) and its value.

-> DM_Options options

Controls whether DM is to trigger rule processing after an attribute has been set successfully.

Option	Meaning
DMF_Inhibit	This option means that the function call is not to trigger any internal events. If this flag is set, you can achieve a better performance in case that a lot of attribute changes have been made by the application.
DMF_ShipEvent	This option means that the function call is to trigger internal events. Then rules are triggered which have been defined for this object and which react to the changing of the specified attribute.

Return Value

TRUE	The attribute was reset successfully.
FALSE	The attribute could not be reset.

3.59 DM_SaveProfile

This function writes the current values of all configurable ***record*** instances (*.configurable = true*) and **global variables** (declared with **config**) of a dialog or module into a configuration file (profile), from which they can be reloaded using the function **DM_LoadProfile()**.

For ***records***, only values that are not inherited are written into the file by default. In order to also write the inherited values into the file, the parameter *options* needs to be set to *DMF_SaveAll*.

Only values from the indicated dialog or module are saved. ***Records*** and variables imported from other modules are omitted.

```
DM_Boolean DML_default DM_EXPORT DM_SaveProfile
(
  DM_String  filename,
  DM_ID      dialog,
  DM_String  comment,
  DM_Options options
)
```

Parameters

-> DM_String filename

This parameter defines the file name of the configuration file. A file path can be specified which may also contain an environment variable.

-> DM_ID dialog

This parameter contains the identifier of the dialog or module whose ***record*** and variable values are to be written into the file.

-> DM_String comment

In this parameter a text can be specified, which is written as a comment into the configuration file.

-> DM_Options options

These are the options available:

Option	Meaning
DMF_SaveAll	Writes the inherited values into the configuration file too.

Return value

DM_TRUE	Saving the values in the configuration file has been successful.
DM_FALSE	The values could not be saved. This may be due to errors accessing the file or an invalid module ID.

Availability

Since IDM version A.06.02.g

See also

C function DM_LoadProfile

Built-in function saveprofile

3.60 DM_SendEvent

The execution of a rule attached to an external event is registered by the application with the function **DM_SendEvent**.“Registered” means that the rule is not executed immediately, but that the external event is queued and then processed according to the dialog event mechanisms.

The event is processed by the IDM with the usual event processing algorithm which results in calling a rule with the scheme on `<object> extevent <no.>`.

```
DM_Boolean DML_default DM_EXPORT DM_SendEvent
(
  DM_ID       objectID,
  DM_Value   *eventData,
  DM_UInt     argc,
  DM_Value   *argv,
  DM_Options  options
)
```

Parameters

-> DM_ID objectID

This is the identifier of the object to which this external event is to be sent.

-> DM_Value *eventData

This parameter defines the external event to be triggered.

-> DM_Int argc

This parameter transfers the number of parameters (up to 16).

-> DM_Value *argv

This parameter indicates the arguments (up to 16) which the IDM passes to the rule on invocation. This vector must have the length that is specified in the *argc* parameter.

-> DM_Options options

The following values can be specified as options:

Option	Meaning
DMF_DontTrace	This option implies that the function call shall not be traced, if the application is started with the trace option.
DMF_Synchronous	This option can be set, if it is ensured that the function **DM_SendEvent** is called synchronously to the process. In this case, the function internally can work more efficiently. A synchronous call is not given, when the function is called from a signal handler, for instance.

Option	Meaning
DMF_NoCriticalSection	This option prevents the function from using a “critical section” on MICROSOFT WINDOWS.

Return Value

DM_TRUE External event could be put in queue.

DM_FALSE External event could not be put in queue.

Note for Microsoft Windows

Since IDM version A.05.01.a, the function **DM_SendEvent** uses a “critical section” to ensure, that it has been completed before this function or the function**DM_QueueExtEvent** is invoked once more. If one of these functions is called in a situation where a “critical section” is not permitted, the use of the “critical section” can be prevented through the option *DMF_NoCriticalSection*.

Attention

A thread, that carries out one of the two functions must not be canceled.

See Also

C functions **DM_QueueExtEvent**, **DM_SendMethod**

Chapter “External Events” and built-in function sendevent() in manual “Rule Language”

Resource message

3.61 DM_SendMethod

This function puts a method call into the event queue to be executed asynchronously from the event loop (DM_EventLoop). It is therefore a convenience function for sending an external event with **DM_SendEvent()** and calling the method in the event rule for that external event.

DM_SendMethod() supports a maximum of *14* arguments for the method call and cannot be used for methods with `output` parameters.

Return values from methods cannot be processed.

Definition

```
DM_Boolean DML_default DM_EXPORT DM_SendMethod
(
  DM_ID      object,
  DM_Method  method,
  DM_UInt    argc,
  DM_Value  *argv,
  DM_Options options
)
```

Parameters

-> DM_ID *object*

Object whose method shall be invoked asynchronously.

-> DM_Method *Method*

Identifier of the method to invoke.

-> DM_Int argc

This parameter transfers the number of arguments for the method call (up to *14*).

-> DM_Value *argv

This parameter indicates the arguments (up to *14*) which the IDM passes to the method on invocation. This vector must have the length that is specified in the *argc* parameter.

-> DM_Options options

This parameter is reserved for future versions. At present pass only *0*.

Return value

DM_TRUE	Method call has been put into the event queue.
DM_FALSE	Method call could not be put into the event queue.

Availability

Since IDM version A.06.02.g

See also

C function DM_SendEvent

Built-in function sendmethod

3.62 DM_SetContent

Using this function the complete contents of an object can be set by the application in one function call. This function is significantly faster than setting the contents with DM_SetValue and the attribute AT_content. The function can be used in tablefields, poptexts, and listboxes.

```
DM_Boolean DML_default DM_EXPORT DM_SetContent
(
  DM_ID objectID,
  DM_Value *firstindex,
  DM_Value *lastindex,
  DM_Content *contentvec,
  DM_UInt count,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Specifies the object to be filled with the new contents.

-> DM_Value *firstindex

Controls which range of the contents is modified by this function. This parameter then defines the starting point of the range.

For a listbox or a poptext the type in the DM_Value structure has to be set to DT_index and the index value in the union has to be assigned the starting value. For tablefield you have to set the type in the DM_Value structure to DT_index and the index value in the union has to be assigned the starting value. For index.first you have to specify the row, for index.second you have to specify the column.

Note

If this parameter is a *NULL* pointer, the starting point has the following defaults, e.g.

listbox	integer = 1
poptext	integer = 1
tablefield	index.first = 1, index.second = 1

-> DM_Value *lastindex

Controls which range of the contents is to be modified by this function. This parameter defines the ending point of the range. For a listbox or a poptext the type in the DM_Value structure has to be set to DT_index and the index value in the union has to be assigned the ending value. For tablefield you have to set the type in the DM_Value structure to DT_index and the index value in the union has to be assigned the ending value. For index.first you have to specify the row, for index.second you have to specify the column.

Note

If the parameter is a *NULL* pointer, the ending point is defined by the size of the new contents. The object contents is cut after the last modified entry.

listbox	.itemcount is modified
poptext	.itemcount is modified
tablefield	if direction = 1, then .rowcount will be modified if direction = 2, then .colcount will be modified

-> DM_Content *contentvec

Passes on the new contents of the object. The information in this array can be deleted after DM_SetContent has been called successfully in the application, since the DM copies the information.

For the object tablefield, the contents is specified as a list and is read according to the index. The attribute .direction determines whether the rectangular area which is defined by firstindex and lastindex, is filled over rows or columns:

.direction = 1	rectangular area is filled row-wise
.direction = 2	rectangular area is filled column-wise

-> DM_UInt count

Specifies the number of elements to be set by the call.

-> DM_Options options

Controls whether DM is to trigger rule processing after an attribute has been set successfully.

Option	Meaning
DMF_Inhibit	This option means that the function call is not to trigger any internal events. If this flag is set, you can achieve a better performance in case that a lot of attribute changes have been made by the application.
DMF_ShipEvent	This option means that the function call is to trigger internal events. Then rules are triggered which have been defined for this object and which react to the changing of the specified attribute.
DMF_UseUserData	This option means that on analyzing the attribute vector, the attribute *.userdata* has to be considered. If the option *DMF_UseUserData* is set, DM copies the userdata for each object entry. If the option is not set, the userdata will be ignored.

Option	Meaning
DMF_OmitActive	This option means that on analyzing the attribute vector, the attribute *.active* is not to be considered. If the option *DMF_OmitActive* is set, DM will ignore the activating state for each object entry. If the object is not set, the activating state of the entries will be normally accepted.
DMF_OmitStrings	This option means that on analyzing the attribute vector, the attribute *.content* is not to be considered. If the option *DMF_OmitStrings* is set, DM will ignore the strings for each object entry. If the option is not set, the contents of the entries will normally be accepted.
DMF_OmitSensitive	This option means that on analyzing the attribute vector, the attribute *.sensitive* is not to be considered. If the option *DMF_OmitSensitive* is set, DM will ignore the selectivity of each object entry. If the option is not set, the selectivity of the entries will normally be accepted.

Return Value

TRUE	Object was filled successfully.
FALSE	Object could not be filled.

Example

Filling a tablefield:

```
static DM_Content *content;
static ushort ColCount;

void DML_default DM_ENTRY ContInit__2(
(DM_ID, table)
(long, fillRows))
{
    DM_Value data;
    ushort rowcount;
    ushort rowheader;
    ushort count;
    DM_Value first, last;
    ushort i;

    DM_GetValue(table, AT_rowcount, 0, &data, 0);
    rowcount = data.value.integer;

    DM_GetValue(table, AT_colcount, 0, &data, 0);
```

```
    colcount = data.value.integer;

    DM_GetValue(table, AT_rowheader, 0, &data, 0);
    rowheader = data.value.integer;

    count = (rowcount-rowheader) * ColCount;

    content=(DM_Content*)DM_Malloc(count*sizeof(DM_Content));

    for (i=0; i<count; i++)
    {
        char buf[10];

        sprintf(buf, "<%8d>", i);

        content[i].string = DM_Strdup(buf);
        content[i].active = FALSE;
        content[i].sensitive = TRUE;
    }

    if (fillRows > (rowcount-rowheader))
         fillRows = rowcount-rowheader;

    if (fillRows)
    {
        first.type = DT_index;
        first.value.index.first = rowheader + 1;
        first.value.index.second =  1;

        last.type = DT_index;
        last.value.index.first = fillRows + rowheader;
        first.value.index.second =  ColCount;

        DM_SetContent(table, &first, &last, content,
                    fillRows*ColCount,0);
    }
}
```

3.63 DM_SetMultiValue

Using this function you are able to set several attributes of different DM objects in one function call. This function should therefore be implemented especially together with the DISTRIBUTED DIALOG MANAGER (DDM), since the network capacity is considerably reduced.

For information on the permitted attributes for the corresponding object type please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_SetMultiValue
(
  DM_MultiValue *values,
  DM_UInt count,
  DM_ID dialogID,
  DM_String pathname,
  DM_Options options
)
```

Parameters

<-> DM_MultiValue *values

List of attributes and objects to be reset. If the element in the structure for the object is set to 0, the object described in the parameter *pathname* is chosen. The list has to have at least the length specified in the parameter *count*.

-> DM_UInt count

Specifies the length of the object-attribute vector indicated in the parameter *values*.

-> DM_ID dialogID

This parameter describes the dialog to which the given objects belong. It has to be specified only if the object identifier whose attributes are to be set is not known and thus the name of the object is specified in the parameter *pathname*.

-> DM_String pathname

Describes the object the attributes of which are to be set. It will only be allocated if the object's internal identifier is not yet known.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	The attributes have been set successfully.
FALSE	At least one attribute could not be set.

Example

Setting coordinates for several objects:

```
void DML_default DM_ENTRY Set  __3((DM_ID, o1),
                   (DM_ID, o2),
                   (DM_ID, o3))
{
  DM_MultiValue val[3];
/* Setting the relevant object */
  val[0].object = o1;
  val[1].object = o2;
  val[2].object = o3;

/* Setting the relevant index type */
  val[0].index.type = DT_void;
  val[1].index.type = DT_void;
  val[2].index.type = DT_void;

/* Setting the relevant attribute */
  val[0].attribute = AT_xleft;
  val[1].attribute = AT_width;
  val[2].attribute = AT_xright;

/* Setting the attribute datatype */
  val[0].data.type = DT_integer;
  val[1].data.type = DT_integer;
  val[2].data.type = DT_integer;

  val[0].data.value.integer = 10;
  val[1].data.value.integer = 50;
  val[2].data.value.integer = 10;

  DM_SetMultiValue(val, 3, dialogID, (char *)0, 0);
}
```

3.64 DM_SetToolkitData

With this function a direct access to the window system is possible. This means that with this function the application is able to change attributes that are not supported by the ISA Dialog Manager but exist in the toolkit.

```
DM_Boolean DML_default DM_EXPORT DM_SetToolkitData
(
  DM_ID objectID,
  DM_Attribute attr
  FPTR value,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

This parameter is the identifier of the object whose window system specific data is to be changed.

-> DM_Attribute attr

With the help of this parameter you can define which window system attribute should be changed.

-> FPTR value

This parameter can be used to set new, window system specific data of the object.

-> DM_Options options

This parameter is currently not used and must therefore be set to 0.

Return value

TRUE	The attribute was set successfully.
FALSE	The attribute could not be set successfully.

The attributes and the associated return values are window system dependent and will be explained in the following chapters.

3.64.1 Motif

These functions can be used to change the data necessary for X-Windows, such as “window-id”, “widget” and “color”. The meaning of these data types is explained in the corresponding X-Windows manuals.

The following values are allowed for the attributes:

attribute	Meaning
AT_CanvasData	This value stores the user-specific data of a canvas. This data is remembered by the specified canvas and contains user-specific data. **See also** Chapter „Strukturen für Canvas-Funktionen“ in manual „C-Schnittstelle - Grundlagen“
AT_XAppClass	This attribute can be used to set the Xt application class.
AT_XColor	This value sets the X-Windows specific structure for the specified color. The value of the value parameter of the function should be of type “Pixel”.
AT_XCursor	This value sets the X-Windows specific structure for the specified cursor. The value of the value parameter of the function should be of type “Cursor”.
AT_XFont	This value sets the X-Windows specific structure for the specified font. The value of the value parameter of the function should be of type XFontStruct*. The availability of this attribute depends on the Motif version used.
AT_XFontSet	This value sets the X-Windows specific structure for the specified font. The value of the value parameter of the function should be of type XFontSet. The availability of this attribute depends on the Motif version used.
AT_XmFontList	This value sets the X-Windows specific structure for the specified font. The value of the value parameter of the function should be of type XmFontList. The availability of this attribute depends on the Motif version used.
AT_XtAddEvents	This attribute can be used to request additional X events for a canvas, e.g. mouse movements. When using this attribute, the event mask ("event_mask") must be passed in the value parameter and the non-selected events in options (non-maskable). If no more additional events are to be sent to the canvas, value and options must be set to 0.

Note for multiscreen dialogs

When called with AT_XTile or AT_XColor, only the tile or color of the default screen can be set.

See also

Chapter„Multiscreen Ssupport untder Motif“ in manual „Programmiertechniken“

3.64.2 Microsoft Windows

With the help of these functions the data necessary for Microsoft Windows can be changed. The meaning of these data types is explained in the corresponding Microsoft Windows manuals.

The following values are allowed for the attributes:

attribute	Meaning
AT_CanvasData	This value stores the user-specific data of a canvas. This data is remembered by the specified canvas and contains user-specific data. **See also** Chapter „Strukturen für Canvas-Funktionen“ in manual „C-Schnittstelle - Grundlagen“

AT_ClipboardText	Permitted only at the ***setup*** object.
	The AT_ClipboardText attribute can be used to set the content of the MS Windows clipboard:
	`DM_SetToolkitData(<setup>, AT_ClipboardText, str, 0);`
	The string obtained remains valid until DM_GetToolkitData or DM_SetToolkitData is called again.
	To release the string without changing the clipboard use the call
	`DM_SetToolkitData(<setup>, AT_ClipboardText, (FPTR) 0, 0);`

AT_WinDisableAll	This value makes all toplevel windows of the same application insensitive - except for the window whose DM-ID is specified as *value* parameter.
AT_WinEnableAll	This value makes all toplevel windows of the same application sensitive - except for the window whose DM-ID is specified as *value* parameter.
AT_XColor	This is used to set a Microsoft Windows RGB value for a color resource. The value *0* resets to the original resource value.

attribute	Meaning
AT_XTile	This is used to set a Microsoft Windows Brush for a color resource. The value must be a valid Microsoft Windows Brush handle. The value should absolutely correspond to the value of *AT_XColor*, since it cannot be predicted when the Dialog Manager will use the *AT_XColor* value or the *AT_XTile* value. The value *0* resets to the original resource value. **Attention** If the **value is invalid**, the ISA Dialog Manager may **crash**.
AT_wsidata	This is used to set a Microsoft Windows cursor for a cursor resource. The value must be a valid Microsoft Windows cursor handle. Der Wert *0* setzt wieder auf den originalen Ressourcenwert zurück-.The value *0* resets to the original resource value. It releases the original resource if it is no longer needed by Dialog Manager. **Attention** If the **value is invalid**, the ISA Dialog Manager may **crash**.
AT_wsidata	Allows you to set a Microsoft Windows font for a font resource. The value must be a valid Microsoft Windows font handle. The value *0* resets to the original resource value. **Attention** If the **value is invalid**, the ISA Dialog Manager may **crash**.

attribute	Meaning
AT_Tile / AT_XTile / AT_wsidata	This allows the setting of an own image for a tile resource. Depending on the option set, the following data types must be specified: - DMF_TlkDataIsIcon: Microsoft Windows Icon Handle - DMF_TlkDataIsWMF: Microsoft Windows Metafile Handle - DMF_TlkDataIsEMF: Microsoft Windows Enhanced Metafile Handle - DMF_TlkDataIsD2D1Bmp: Microsoft Direct 2D Bitmap (ID2D1Bitmap *) - DMF_TlkDataIsD2D1SVG: Microsoft Direct 2D SVG Documnet (ID2D1SvgDocument *) - DMF_TlkDataIsD2D1EMF: Microsoft Direct 2D Metafile (ID2D1GdiMetafile *) - other: Microsoft-Windows Bitmap Handle The value *0* resets to the original resource value. **Attention** If the **value is invalid**, the ISA Dialog Manager may **crash**. Note: If a HANDLE is set, it is usually converted internally to a Microsoft Direct2D object. A query returns the set HANDLE and not the converted value.
AT_XColor	This is used to set a Microsoft Windows color palette for a tile resource, which is used to draw a bitmap (not an icon). The value must be a valid Microsoft Windows palette handle. The value *0* resets to the original resource value. **Attention** If the **value is invalid**, the ISA Dialog Manager may **crash**.

3.65 DM_SetValue

This function enables you to change attributes of DM objects. For information on the attributes permitted for the relevant object type, please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_SetValue
(
  DM_ID object,
  DM_Attribute attr,
  DM_UInt index,
  DM_Value *data,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Describes the object the attribute of which you want to change.

-> DM_Attribute attr

Describes the attribute to be changed. All valid attributes are defined in **IDMuser.h**.

-> DM_UInt index

This parameter is only analyzed for vector attributes of objects and describes the index of the searched sub-object (e.g. text in listbox).

-> DM_Value*data

In this parameter the value to be accepted by the attribute is to be transferred. You have to take care, however, to assign the correct element in this union. For information on the data type of the individual attributes, please refer to the “Attribute Reference”.

-> DM_Options options

Using this parameter you can control whether DM is to trigger the rule processing by setting the attribute successfully.

Option	Meaning
DMF_Inhibit	This option means that the function call is not to trigger any internal events. If this flag is set, you can achieve a better performance in case that a lot of attribute changes have been made by the application.
DMF_ShipEvent	This option means that the function call is to trigger internal events. Then rules are triggered which have been defined for this object and which react to the changing of the specified attribute.

Option	Meaning
DMF_AcceptChild	This option is valid for the attribute *AT_options* (object canvas) on Motif. It describes a canvas which can have child objects.
DMF_NoFocusFrame	Valid on Motif with attribute *AT_options*. It describes a canvas which is not to have a focus border.
DMF_XlateString	The specified string is to be translated in the currently used language. This is of course only possible if the text already exists internally and if also a translation exists for it.

Return Value

TRUE	The attribute has been set successfully.
FALSE	The attribute could not be set.

Example

Application-callback function checking whether an input string corresponds to an existing file.

```
DM_Boolean DML_default DM_CALLBACK CheckFilename __1(
(DM_CallBackArgs *, data))
{
  DM_Value value;        /* structure for DM_SetValue */
  FILE *fptr;            /* file pointer */
  DM_ID id;            /* Identifier of object */


  /* get the current content */
  if (DM_GetValue(data->object, AT_content, 0, &value,
        DMF_GetLocalString))
    /* check the datatype */
    if(value.type == DT_string)
    {
      /* try to open the file */
      if(!((fptr = fopen(value.value.string, "r"))))
      {
        /*
         * the file cannot be opened for reading.
         * activate the edittext again
         */
        value.type = DT_boolean;
        value.value.boolean = TRUE;
        DM_SetValue(data->object, AT_active, 0, &value,
            DMF_Inhibit);
```

```
        /*
         * The file cannot be read. So don't continue
         * processing with the rules
         */
        return (FALSE);
      }
      else
        fclose(fptr);

      /*
      * the file could be opened. enable the other objects
      */
      value.type = DT_boolean;
      value.value.boolean = TRUE;

      /* Get the id of the edittext "Actives" */
      if ((id = DM_PathToID(0, "Actives")))
        /* Change the object to sensitive */
        DM_SetValue(id, AT_sensitive, 0, &value,
            DMF_ShipEvent);
      /*
       * the last parameter must be ShipEvent,
       * because a rule should be triggered
       */

      /*
       * Everything is ok. So let the rule process normally
       */
      return(TRUE);
    }

  /* Too many errors don't continue the rule processing */
  return (FALSE);
}
```

See Also

Built-in function setvalue in manual “Rule Language”

3.66 DM_SetValueIndex

With this function attributes of the tablefield object can be changed. The function can work with two indexes.

For the attributes valid for the relevant object type, please refer to the “Object Reference”.

```
DM_Boolean DML_default DM_EXPORT DM_SetValueIndex
(
  DM_ID objectID,
  DM_Attribute attr,
  DM_Value *index,
  DM_Value *data,
  DM_Options options;
)
```

Parameters

-> DM_ID objectID

Describes the object the attribute of which you want to change.

-> DM_Attribute attr

Describes the attribute to be changed. All valid attributes are defined in **IDMuser.h**

-> DM_Value *index

Specifies the data type of the index (enum, index) and its value.

-> DM_Value *data

In this parameter the value to be accepted by the attribute is to be transferred. You have to take care, however, to assign the correct element in this union. For information on the data type of the individual attributes, please refer to the “Attribute Reference”.

-> DM_Options options

Using this parameter you can control whether DM is to trigger the rule processing by setting the attribute successfully.

Option	Meaning
DMF_Inhibit	This option means that the function call is not to trigger any internal events. If this flag is set, you can achieve a better performance in case a lot of attribute changes have been made by the application.
DMF_ShipEvent	This option means that the function call is to trigger internal events. Then rules are triggered which have been defined for this object and which react to the changing of the specified attribute.

Option	Meaning
DMF_XlateString	The specified string is to be translated in the currently used language. This is of course only possible if the text already exists internally and if also a translation exists for it.

Return Value

TRUE	The attribute was set successfully.
FALSE	The attribute could not be set.

Example

A tablefield is to be filled row-wise by means of the function DM_SetValueIndex.

```
void DML_default DM_ENTRY SetTable __3(
  (DM_ID, tableID),
  (DM_Integer, Rows),
  (DM_Integer, Cols))
{
  DM_Value index,data;
  int row, column;

  index.type = DT_index;
  data.type = DT_string;
  data.value.string = "new string";
  for (row = 1; row <= (int) Rows; row++)
  {
    index.value.index.first = row;
    for (column = 1; column <= (int) Cols; column++)
    {
        index.value.index.second=column;
        DM_SetValueIndex(tableID, AT_content, &index, &data,
          DMF_Inhibit);
    }
  }
}
```

3.67 DM_SetVectorValue

Using this function you can set attributes which occur several times in an object (so-called "vector attributes").

```
DM_Boolean DML_default DM_EXPORT DM_SetVectorValue
(
  DM_ID objectID
  DM_Attribute attr,
  DM_Value *firstindex,
  DM_Value *lastindex,
  DM_VectorValue *values,
  DM_Options options
)
```

Parameters

-> DM_ID objectID

Describes the object the attribute of which you want to change.

-> DM_Attribute attr

Describes the attribute to be changed.

-> DM_Value *firstindex

Controls which range of the contents is modified by this function. This parameter then defines the starting point of the range.

For a listbox or a poptext the type in the DM_Value structure has to be set to DT_index and the index value in the union has to be assigned the starting value. For tablefield you have to set the type in the DM_Value structure to DT_index and the index value in the union has to be assigned the starting value. For index.first you have to specify the row, for index.second you have to specify the column.

Note

If this parameter is a *NULL* pointer, the starting point has the following defaults, e.g.

listbox	integer = 1
tablefield	index.first = 1, index.second = 1

-> DM_Value *lastindex

Controls which range of the contents is to be modified by this function. This parameter defines the last point of the range. For a listbox or a poptext the type in the DM_Value structure has to be set to DT_index and the index value in the union has to be assigned the ending value. For tablefield you have to set the type in the DM_Value structure to DT_index and the index value in the union has to be assigned the ending value. For index.first you have to specify the line, for index.second you have to specify the column.

Note

If the parameter is a *NULL* pointer, the ending point is defined by the size of the new contents. The object contents is cut after the last modified entry.

listbox	.itemcount is modified
tablefield	if direction = 1, then .rowcount will be modified if direction = 2, then .colcount will be modified

-> DM_VectorValue *values

Pointer to the values to be set. By the field *type* in the DM_VectorValue structure you can control which data type the individual values have.

By the field *count* in the DM_VectorValue structure you can control how many values a vector is to have.

-> DM_Options options

Controls whether the DM is to trigger rule processing after an attribute has been set successfully.

Option	Meaning
DMF_Inhibit	This option means that the function call is not to trigger any internal events. If this flag is set, you can achieve a better performance in case a lot of attribute changes have been made by the application.
DMF_ShipEvent	This option means that the function call is to trigger internal events. Then rules are triggered which have been defined for this object and which react to the changing of the specified attribute.
DMF_XlateString	The specified string is to be translated into the currently used language. This is of course only possible if the text already exists internally and if also a translation exists for it.

Return Value

TRUE	The attribute was set successfully.
FALSE	The attribute could not be set.

Example

A new content is to be set for a listbox.

```
void DML_default DM_ENTRY SetVector  __1((DM_ID, lb)){  DM_Value first, last;
  DM_VectorValue vec;
  char *list[9]

  list[0] = "V_1";
```

```
  list[1] = "V_2";
  list[2] = "V_3";
  list[3] = "V_4";
  list[4] = "V_5";
  list[5] = "V_6";
  list[6] = "V_7";
  list[7] = "V_8";
  list[8] = "V_9";

/*
** Setting the starting and ending line
** Content of line 1 to 9 is replaced by
** the new content. The rest remains unchanged.
*/
  first.type = DT_integer;
  first.value.integer = 1;
  last.type = DT_integer;
  last.value.integer  = 9;

/* Specifying the number of lines to be set */
  vec.type = DT_string;
  vec.count = 9;
  vec.vector.stringPtr = list;
  DM_SetVectorValue(lb, AT_content, &first, &last, &vec,
    DMF_ShipEvent);

}
```

3.68 DM_ShutDown

This function shuts down the DM. If this function is called all global initialization procedures will be recalled. It is usually called by the same function that calls "AppMain" in the application. Therefore, **DM_ShutDown** may only be called if the “Main” program in the DM is replaced by an application-specific one.

```
void DML_default DM_EXPORT DM_ShutDown
(
  void
)
```

Parameters

None.

Return value

None.

Example

Start program of Dialog Manager which is usually linked by means of the files **startup.o** or **startup.obj**.

```
int cdecl main __2(
(int, argc),
(char far * far *, argv))
{
  register int status;
  static char running = 0;

  if ((status = running++) == 0)
  {
    if ((status = DM_BootStrap(&argc, &argv)) == 0)
    {
      DM_InitOptions(&argc, argv, 0);

      DM_TraceMessage ("[AC] Transfer to AppMain(...)",
        DMF_Printf | DMF_InhibitTag);
      status = AppMain (argc, argv);
      DM_TraceMessage ("[AR] AppMain() = %d", DMF_Printf |
        DMF_InhibitTag, status);

      DM_ShutDown();
    }
    else DM_TraceMessage ("Bootstrap failed", DMF_LogFile);
  }
  else
    DM_FatalAppError ("Unexpected restart", -1, 0);
```

```
  return (status);
}
```

3.69 DM_StartDialog

This function starts the actual dialog application. The DM creates all necessary resources (colors, cursor, fonts etc.) in the window system, puts all top level objects defined as visible in the dialog on the screen, and executes the start rule.

```
DM_Boolean DML_default DM_EXPORT DM_StartDialog
(
  DM_ID dialogID,
  DM_Options options
)
```

Parameters

-> DM_ID dialogID

Identifier of the dialog to be started. This identifier was received as the return value from DM_ LoadDialog.

-> DM_Options options

Currently not used. Please specify with 0.

Return Value

TRUE	The dialog was successfully started.
FALSE	The dialog could not be started, e.g. because another dialog is already running or the given parameter is no dialog.

Example

Typical main program for DM applications:

```
int DML_c DM_CALLBACK AppMain __2(
(int, argc),
(char far * far *, argv))
{
  DM_ID dialogID;

  /*
   * Initialize Dialog Manager
  */
  if (!DM_Initialize (&argc, argv, 0))
  {
    DM_TraceMessage("could not initialize", DMF_LogFile);
    return (1);
  }

  /*
   * Load the dialog file
```

```
  */

  dialogID = DM_LoadDialog ("tabdemo.dlg",0);
  if (!dialogID)
  {
    DM_TraceMessage("could not load dialog", DMF_LogFile);
    return(1);
  }

  /*
   * Start the dialog and enter event loop
  */

  if (DM_StartDialog (dialogID, 0))
    DM_EventLoop (0);
  else
    return (1);

  return (0);
}
```

See Also

Built-in function run in manual “Rule Language”

3.70 DM_StopDialog

This function cancels a dialog. If that dialog was the last running dialog the event loop is left. If no dialog is specified the current dialog is stopped.

```
DM_Boolean DML_default DM_EXPORT DM_StopDialog
(
  DM_ID dialogID,
  DM_Options options
)
```

Parameters

-> DM_ID dialogID

Identifier of the dialog to be stopped. This identifier was received as the return value from DM_LoadDialog

-> DM_Options options

In this parameter the following values are possible:

Option	Meaning
DMF_Destroy	This option means that the stopped dialog is to be deleted. If this option is not set, the stopped dialog will remain in the memory and be restarted.

Return Value

TRUE	The dialog could be stopped successfully.
FALSE	The dialog could not be stopped.

See Also

Built-in function stop in manual “Rule Language”

3.71 DM_StrCreate

With the function **DM_StrCreate** a text with a given character encoding can be created.

```
DM_String DML_default DM_EXPORT DM_StrCreate
(
  DM_String str,
  DM_UInt1 strCP,
  DM_UInt1 toCP,
  DM_Options options
)
```

Parameters

-> DM_String str

Source text to initialize the newly created text.

-> DM_UInt1 strCP

Character encoding (code page) of the source text.

-> DM_UInt1 toCP

Character encoding (code page) of the newly created text.

-> DM_Options options

Currently unused, should be set to 0.

Parameter Values

Besides the already known code page constants (CP_ascii, CP_iso8859,...) the following code page constants can be used for the parameters *strCP* and *toCP*:

Constant	Meaning
CP_appl	currently set application code page
CP_format	currently set format code page
CP_input	currently set input code page
CP_output	currently set output code page
CP_display	currently set display code page
CP_system	currently set system code page

These constants may **only** be used with DM_StrCreate.

Return Value

The newly created text.

The text has to be freed with DM_Free when it is not needed anymore.

Availability

DM_StrCreate is available as of IDM release A.05.02.k.

3.72 DM_Strdup

Using this function you can duplicate any strings. For the copies of the strings you have to use the Dialog Manager functions for the memory administration DM_Malloc. This is why these strings can be released only via the function DM_Free.

```
DM_String DML_default DM_EXPORT DM_Strdup
(
  DM_string string
)
```

Parameters

-> DM_string string

This parameter is the string to be duplicated.

Return Value

This function returns the pointer to the duplicate of the string or a *NULL* pointer if the string could not be duplicated.

Example

A string transferred from Dialog Manager is to be saved for the application.

```
  DM_String new_string;

void DML_default DM_ENTRY StoreString __1((DM_String string))
{

  if (new_string = DM_Strdup(string) == (DM_String) 0)
    DM_TraceMessage("String could not be created", 0);
}
```

If this string is not needed any more, it has to be freed via the function DM_Free.

```
void DML_default DM_ENTRY FreeString __0((void))
{

  DM_Free(new_string);

}
```

3.73 DM_StringChange

This function can be used to modify or manipulate a managed string.

Besides the mere replacement of the string, it is possible to concatenate strings via the *DMF_AppendValue* option.

If the *pstring* parameter is a yet unmanaged string, it is copied and from then on managed by the IDM. If the option *DMF_StaticValue* has been set, the string will be treated as a static or global string and not be released as usual when the function returns.

Generally, only string arguments and local or global strings can be managed.

```
DM_Boolean DML_default DM_EXPORT DM_StringChange
(
  DM_String * pstring,
  DM_String   newstring,
  DM_Options  options
)
```

Parameters

<-> DM_String * pstring

This parameter refers to the string reference that shall be manipulated. It may already be a managed string, however this is not mandatory.

-> DM_String newstring

This parameter refers to the string which shall be assigned or appended to the string reference (*pstring* parameter).

-> DM_Options options

These are the options available:

Option	Meaning
DMF_StaticValue	When the *pstring* parameter is automatically converted into a managed string reference, it is treated as a static respectively global string reference.
DMF_AppendValue	The string in the *newstring* parameter is appended to the string in the *pstring* parameter.

Return value

DM_TRUE	The string has been manipulated successfully.
DM_FALSE	An error occurred. This may be due to a wrong runstate on call or an invalid string reference.

Example

Dialog File

```
dialog YourDialog
function anyvalue StringOf(integer I);

on dialog start
{
  print StringOf(123);
  print StringOf(-42);
  exit();
}
```

C Part

```
...

DM_String DML_default DM_ENTRY StringOf(DM_Integer I)
{
  char      buf[10];
  DM_String data;
  DM_String negative;

  if (I>=0)
  {
    /* return an unmanaged local string */
    sprintf(buf, "%d", (int)I);
    data = buf;
    /* wrong: return data; => buf is a local char array! */
    return DM_StringReturn(&data, 0);
  }
  else
  {
    /* return a managed string */
    DM_StringInit(&negative, 0);  /* can be omitted */
    DM_StringChange(&negative, "!!negative", &data, 0);
    return DM_StringReturn(&negative);
    /* return &negative; => also possible for managed strings! */
  }
}
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_StringInit, DM_StringReturn

3.74 DM_StringInit

This function converts a string into a local or global respectively static string managed by the IDM. This allows further manipulation of the string with the **DM_StringChange()** function, as well as simplified handling as parameter or return value. The IDM then manages the memory for the string.

For managed strings, only read access of the string characters or the string pointer is allowed. Releasing via **DM_Free()** is not allowed!

This function initializes the string with *NULL*.

By specifying the *DMF_StaticValue* option, the string is treated as static and is not released after the function returns, as it is usually the case for local strings.

Generally, only string arguments and local or global strings can be managed.

```
DM_Boolean DML_default DM_EXPORT DM_StringInit
(
  DM_String * pstring,
  DM_Options  options
)
```

Parameters

-> DM_String * pstring

This parameter refers to the string pointer that shall be initialized and managed.

-> DM_Options options

These are the options available:

Option	Meaning
DMF_StaticValue	The *pstring* parameter will be initialized as a static or global string reference.

Return value

DM_TRUE	The string has been initialized successfully and is now managed.
DM_FALSE	The string could not be managed or initialized.

Example

Dialog File

```
dialog YourDialog
function string FormatString(integer Op, string String);

on dialog start
{
```

```
  print FormatString (-1, "**");          // set decoration
  print FormatString (0, "hello world");  // print with decoration
  exit();
}
```

C Part

```
...

DM_String DML_default DM_ENTRY FormatString (DM_Integer Op, DM_String String)
{
  static DM_String decoration = NULL;
  DM_String newstring;

  if (!decoration)
  {
    /* static initialization */
    DM_StringInit(&decoration, DMF_StaticValue);
    DM_StringChange(&decoration, "-", 0);
    /* above can be done simpler via:
     * DM_StringChange(&decoraton, "-", DMF_StaticValue);
     */
  }

  switch(Op)
  {
  case -1: /* change decoration */
    DM_StringChange(&decoration, String);
    return &decoration;
  case 1:  /* decor only at the beginning */
    DM_StringChange(&newString, decoration, 0);
    DM_StringChange(&newString, String, DMF_AppendValue);
    break;
  case 2:  /* decor only at the end */
    DM_StringChange(&newString, String, 0);
    DM_StringChange(&newString, decoration, DMF_AppendValue);
    break;
  default:
    DM_StringChange(&newString, decoration, 0);
    DM_StringChange(&newString, String, DMF_AppendValue);
    DM_StringChange(&newString, decoration, DMF_AppendValue);
    break;
  }
  return newstring;
  /* also possible: return DM_StringReturn(&newstring, 0); */
}
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_StringChange, DM_StringReturn, DM_ValueInit

3.75 DM_StringReturn

This function is used to safely return local strings (*DM_String* values) from a function. When local variables and structures are used in a C function, they are invalid after they have been returned. This function can safely and easily return a local string.

If necessary, a temporary copy created (e.g. the string in it is copied) for this purpose. There is no copying for managed value references. In this case the returned *DM_String* pointer should be passed to the caller with `return`.

```
DM_String DML_default DM_EXPORT DM_StringReturn
(
  DM_String * pstring,
  DM_Options  options
)
```

Parameters

-> DM_String * pstring

This parameter refers to the string that will be returned. It may be a managed string, however this is not mandatory.

-> DM_Options options

Should be set to *0* since no options are available.

Return value

A valid string to return by a function is returned or *NULL* in case of an error. An error may occur, for instance, if the function is called in the wrong “runstate”, the managed string is invalid or copying failed.

Please Note

The functions DM_StringInit and DM_StringChange can be used to return *output* string parameters.

Example

Dialog File

```
dialog YourDialog
function anyvalue StringOf(integer I);

on dialog start
{
  print StringOf(123);
  print StringOf(-42);
  exit();
}
```

C Part

```
...

DM_String DML_default DM_ENTRY StringOf(DM_Integer I)
{
  char      buf[10];
  DM_String data;
  DM_String negative;

  if (I>=0)
  {
    /* return an unmanaged local string */
    sprintf(buf, "%d", (int)I);
    data = buf;
    /* wrong: return data; => buf is a local char array! */
    return DM_StringReturn(&data, 0);
  }
  else
  {
    /* return a managed string */
    DM_StringInit(&negative, 0);  /* can be omitted */
    DM_StringChange(&negative, "!!negative", &data, 0);
    return DM_StringReturn(&negative);
    /* return &negative; => also possible for managed strings! */
  }
}
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_IndexReturn, DM_StringChange, DM_StringInit, DM_ValueReturn

3.76 DM_TraceMessage

Using this function you can write trace messages from the application in the tracefile of Dialog Manager.

```
void DML_c DM_EXPORT DM_TraceMessage
(
  DM_string string,
  DM_Options options,
  ...
)
```

Parameters

-> DM_string string

This parameter is the string to be written in the tracefile.

-> DM_Options options

For this function the following options are permitted:

Option	Meaning
DMF_InhibitTag	Using this option, the application can influence whether Dialog Manager writes the header “[UM]” at the beginning of a line or not. If the parameter is set to *DMF_InhibitTag*, the beginning of a line will not be printed. If this option is not set, the message in the tracefile will be as follows: *[UM]: specified string
DMF_Printf	If this option is set, the parameter *string* will be interpreted in the same way as in the C function **printf**. The corresponding parameters have to be specified after the parameter *options*.
DMF_LogFile	If this option is set, the output appears in the tracefile, not in the logfile. **Note** If the command line option **-IDMtracefile** is set, everything will always be written in the tracefile. This is also valid if the option *DMF_LogFile* is set. This option does have the above described effect - output in logfile and not in tracefile - only, if **-IDMtracefile** is not set!

Note

The traces will only be printed if the DM has been started with the trace option.

Example

Output of a message in the main program, if the function DM_Initialize returns FALSE as a result.

```
int DML_c DM_CALLBACK AppMain __2(
  (int, argc),
```

```
  (char **, argv))
{
  DM_ID dialogID;

  /* Initialization of Dialog Manager */
  if (!DM_Initialize (&argc, argv, 0))
  {
    DM_TraceMessage("could not initialize",
      DMF_LogFile);
    return (1);
  }
```

3.77 DM_ValueChange

With this function a value reference managed by IDM may be manipulated. Either the entire value can be replaced or a single element value in a collection.

There is an automatic conversion of unmanaged value references into managed value references. If the option *DMF_StaticValue* is set in this case, a static respectively global managed value reference is created. For better control the explicit use of DM_ValueInit is recommended.

If the *value* parameter is a collection, e.g. of type *DT_vector*, *DT_list*, *DT_hash*, *DT_matrix* or *DT_refvec*, an element value can be substituted by specifying the *index* parameter. Similar to predefined attributes, a collection can be extended by incrementing the index with *+1*. For associative arrays, simply a not yet assigned index key may be used.

If a collection in the *data* parameter is assigned to the target as a whole (i.e. with *NULL* as *index* parameter), the entire value with all value elements is copied. The conversion of an argument into a locally managed value also requires a complete copying to allow further manipulation.

```
DM_Boolean DML_default DM_EXPORT DM_ValueChange
(
  DM_Value   *value,
  DM_Value   *index,
  DM_Value   *data,
  DM_Options  options
)
```

Parameters

-> DM_Value * value

This parameter refers to the value reference to be changed. If it is not yet managed by the IDM, it is converted to a managed value reference.

-> DM_Value * index

This parameter can be used to change element values in collections and specifies the index. Otherwise, it should be set to *NULL*. This parameter does not need to be a managed value reference.

-> DM_Value * data

This parameter defines the value to be set. It may be a managed or an unmanaged value reference. If this value is *NULL*, the value or element is set to *DT_undefined*.

-> DM_Options options

These are the options available:

Option	Meaning
DMF_StaticValue	When the *value* parameter is automatically converted into a managed value reference, it is treated as a static respectively global value reference.
DMF_AppendValue	For collections, the data value from *data* is appended at the end. The index must be *NULL* for this.
DMF_SortBinary	In collections, the newly set value is finally sorted. May be used in combination with *DMF_SortReverse*.
DMF_SortLinguistic	In collections, the newly set value is finally sorted, for strings according to linguistic rules (see the function **sort()**). May be used in combination with *DMF_SortReverse*.
DMF_SortReverse	In collections, the newly set value is finally sorted in reverse order.

Return value

DM_TRUE	The function has been completed successfully, setting the value succeeded or the value had already been set.
DM_FALSE	Value could not be set. This may be due to an faulty call, an unmanaged or invalid value reference, or an incorrect indexing.

Example

Dialog File

```
dialog YourDialog
function anyvalue FindData(hash DataHash, string Pattern,
                           anyvalue FirstIndex output);

on dialog start
{
  variable hash Stations := ["1"=>"ABC","2"=>"CBS","9"=>"HBO"];
  variable anyvalue Idx;

  print "Found(D)=" + FindData(Stations,"D",Idx);
  print " at " + Idx;
  exit();
}
```

C Part

```
...
```

```
static DM_Value InvalidIndex;
static DM_Value InvalidValue;

DM_Value * DML_default DM_ENTRY FindData(DM_Value *DataHash, DM_String
Pattern,
                                        DM_Value *FirstIndex)
{
  DM_Value index;
  DM_Value value;

  /* initialize the managed values */
  DM_ValueInit(&index, DT_void, NULL, 0);
  DM_ValueInit(&value, DT_void, NULL, 0);
  DM_StringInit(&retString, 0);

  count = DM_ValueCount(DataHash, NULL, 0);
  while(count>0)
  {
    /* loop through the hash */
    if (DM_ValueIndex(DataHash, count--, &index, 0)
        && DM_ValueGet(DataHash, &index, &value, DMF_GetLocalString)
        && value.type == DT_string)
    {
      if (strstr(value.value.string, Pattern))
      {
        /* return the first found index & value */
        DM_ValueChange(FirstIndex, NULL, &index, 0);
        return DM_ValueReturn(&value, 0);
      }
    }
  }

  /* return the invalid index & value */
  DM_ValueChange(FirstIndex, NULL, &InvalidIndex, 0);
  return &InvalidValue;
}

...

int DML_c AppMain __2((int, argc), (char **,argv))
{
  DM_Value data;

  ...

  data.type = DT_string;
  data.value.string = "NO-VALUE";
```

```
DM_ValueChange(&InvalidValue, NULL, &data, DMF_StaticValue);
data.type = DT_string;
data.value.string = "INVALID-INDEX";
DM_ValueChange(&InvalidIndex, NULL, &data, DMF_StaticValue);

...

DM_StartDialog(...)
```

Availability

Since IDM version A.06.01.a

3.78 DM_ValueCount

Returns the number of values in a collection (without the default values). It is also possible to return the index type or the highest index value.

The returned value indicates the number of values (without the default values). Thus, in combination with the DM_ValueIndex function, loops over all indexed values respectively elements can be implemented easily.

Depending on the *value* parameter, the following results may occur:

value Type	*retvalue* Return Type	Remark
DT_refvec, *DT_list*, *DT_vector*	*DT_integer*	Highest index value
DT_matrix	*DT_index*	Highest index value
DT_hash	*DT_datatype*	Any index type (*anyvalue*)
otherwise	*DT_void*	Non-indexed value

```
DM_UInt DML_default DM_EXPORT DM_ValueCount
(
  DM_Value   *value,
  DM_Value   *retvalue,
  DM_Options  options
)
```

Parameters

-> DM_Value* value

This parameter refers to the value reference from which the number of values is fetched. It should be a managed value reference or function argument.

-> DM_Value * retvalue

If this parameter is not *NULL*, the count value is returned in it. This may be a managed value reference, however this is not mandatory.

-> DM_Options options

These are the options available:

Option	Meaning
DMF_GetLocalString	This option means that text values (IDs of type *DT_text*) should be returned as strings in the currently set language.

Option	Meaning
DMF_GetMasterString	This option means that text values (IDs of type *DT_text)* should be returned as a strings in the development language, regardless of which language the user is currently working with.
DMF_ DontFreeLastStrings	Strings are usually passed to the application in a temporary buffer, which is retained until the next call to the IDM. If strings in the application shall be valid longer, the option *DMF_ DontFreeLastStrings* has to be set. Then the memory will not be released until an IDM function returning a string from the IDM to the applicationis called without this option.

Return value

0 … INT_ MAX	Number of values (excluding the default values with the indexes *[0]*, *[0,*]* or *[*,0]*).
retvalue	Highest index value, may be either *void* (scalar value), an *integer* value (one-dimensional array), an *index* value (two-dimensional array), or a data type (associative array).

Example

Dialog File

```
dialog YourDialog
function integer CountIntegers(anyvalue List);

on dialog start
{
  variable matrix M := [
    [0,0]=>-1,[1,1]=>"ZIP",[1,2]=>"City",[2,1]=>60654,[2,2]=>"Chicago" ];
  print "#Integers in Hash: " + CountInteger(M);
  exit();
}
```

C Part

```
...

static DM_Value InvalidIndex;
static DM_Value InvalidValue;

DM_Integer DML_default DM_ENTRY CountInteger(DM_Value *List)
{
  DM_Value   index;
  DM_Value   count;
```

```
  DM_Value   value;
  DM_Integer icount = 0;

  /* initialize the managed values */
  if (DM_ValueCount(List, &count, 0)>0 && List->type == DT_matrix
      && count->type == DT_index)
  {
    index.type = DT_index;
    index.value.index.first = 0;
    index.value.index.second = 1;

    /* loop through [0,1],[1,1],[2,1],... */
    while(index.value.index.first<=count.value.index.second)
    {
      if (DM_ValueGet(List, &index, &value, 0)
          && value.type == DT_integer)
        icount++;
      index.value.index.first++;
    }
  }
  return icount;
}
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_ValueChange, DM_ValueGet, DM_ValueIndex

Built-in functions countof, itemcount

3.79 DM_ValueGet

This function allows to retrieve a single element value that belongs to a defined index from collections.

If the given index is *NULL*, the entire value is returned, which usually means copying the value.

If the *retvalue* parameter is an unmanaged value, it remains unmanaged. When strings are returned, they are only stored in a temporary buffer which may be cleared or overwritten the next time a DM function without the *DMF_DontFreeLastStrings* option is called.

```
DM_Boolean DML_default DM_EXPORT DM_ValueGet
(
  DM_Value   *value,
  DM_Value   *index,
  DM_Value   *retvalue,
  DM_Options  options
)
```

Parameters

-> DM_Value* value

This parameter refers to the value reference from which the element value is retrieved. It should be a managed value reference or function argument.

-> DM_Value * index

This parameter sets the index for which the element value is retrieved. This parameter does not need to be a managed value. If the parameter is set to *NULL*, the *value* parameter is copied into the return parameter *retvalue*.

-> DM_Value * retvalue

This parameter defines the value to be set. It may be a managed or an unmanaged value reference. If this value is *NULL*, the value or element is set to *DT_undefined*.

-> DM_Options options

These are the options available:

Option	Meaning
DMF_StaticValue	When the *value* parameter is automatically converted into a managed value reference, it is treated as a static respectively global value reference.
DMF_GetLocalString	This option means that text values (IDs of type *DT_text*) should be returned as strings in the currently set language.

Option	Meaning
DMF_GetMasterString	This option means that text values (IDs of type *DT_text)* should be returned as a strings in the development language, regardless of which language the user is currently working with.
DMF_ DontFreeLastStrings	Strings are usually passed to the application in a temporary buffer, which is retained until the next call to the IDM. If strings in the application shall be valid longer, the option *DMF_ DontFreeLastStrings* has to be set. Then the memory will not be released until an IDM function returning a string from the IDM to the applicationis called without this option.

Return value

DM_ TRUE	Getting the value has been successful.
DM_ FALSE	The value could not be retrieved. This may be due to an faulty call, an unmanaged or invalid value reference, or an incorrect indexing.

Availability

Since IDM version A.06.01.a

See also

Functions DM_ValueChange, DM_ValueCount, DM_ValueIndex

3.80 DM_ValueIndex

This function can be used to determine the corresponding index for a position in a collection. This is especially important for values of type *DT_hash* and *DT_matrix* in order to access all respective indexes the easiest way. The function only allows access to indexes that do not belong to default values.

The returned index can be stored in a managed *retvalue* parameter value or in an unmanaged one. In the latter case, the string value is allocated in the temporary buffer if necessary.

```
DM_UInt DML_default DM_EXPORT DM_ValueIndex
(
  DM_Value   *value,
  DM_UInt     indexpos,
  DM_Value   *retvalue,
  DM_Options  options
)
```

Parameters

-> DM_Value* value

This parameter refers to the value reference from which the index for the position will be retrieved. It must be a managed value reference or function argument.

-> DM_UInt indexpos

This parameter defines the position of the index and should be in the range *0 < indexpos <= DM_ValueCount()*.

-> DM_Value * retvalue

Wenn dieser Parameter nicht *NULL* ist, wird hier der entsprechende Index abgelegt. Es kann sich dabei um eine gemanagte oder auch um eine ungemanagte Wertereferenz handeln.

If this parameter is not *NULL*, the retrieved index is stored here. This may be a managed or an unmanaged value reference.

-> DM_Options options

These are the options available:

Option	Meaning
DMF_GetLocalString	This option means that text values (IDs of type *DT_text*) should be returned as strings in the currently set language.
DMF_GetMasterString	This option means that text values (IDs of type *DT_text)* should be returned as a strings in the development language, regardless of which language the user is currently working with.

Option	Meaning
DMF_ DontFreeLastStrings	Strings are usually passed to the application in a temporary buffer, which is retained until the next call to the IDM. If strings in the application shall be valid longer, the option *DMF_ DontFreeLastStrings* has to be set. Then the memory will not be released until an IDM function returning a string from the IDM to the applicationis called without this option.

Return value

DM_ TRUE	The value has an index at this position; the obtained index can afterward be found in **retvalue* if *retvalue != NULL*.
DM_ FALSE	The index could not be determined. This may be due to an faulty call, an unmanaged or invalid value reference, or an incorrect position.

Example

Dialog File

```
dialog YourDialog
function anyvalue FindData(hash DataHash, string Pattern,
                           anyvalue FirstIndex output);

on dialog start
{
  variable hash Stations := ["1"=>"ABC","2"=>"CBS","9"=>"HBO"];
  variable anyvalue Idx;

  print "Found(D)=" + FindData(Stations,"D",Idx);
  print " at " + Idx;
  exit();
}
```

C Part

```
...

static DM_Value InvalidIndex;
static DM_Value InvalidValue;

DM_Value * DML_default DM_ENTRY FindData(DM_Value *DataHash, DM_String
Pattern,
                                         DM_Value *FirstIndex)
{
  DM_Value index;
```

```
  DM_Value value;

  /* initialize the managed values */
  DM_ValueInit(&index, DT_void, NULL, 0);
  DM_ValueInit(&value, DT_void, NULL, 0);
  DM_StringInit(&retString, 0);

  count = DM_ValueCount(DataHash, NULL, 0);
  while(count>0)
  {
    /* loop through the hash */
    if (DM_ValueIndex(DataHash, count--, &index, 0)
        && DM_ValueGet(DataHash, &index, &value, DMF_GetLocalString)
        && value.type == DT_string)
    {
      if (strstr(value.value.string, Pattern))
      {
        /* return the first found index & value */
        DM_ValueChange(FirstIndex, NULL, &index, 0);
        return DM_ValueReturn(&value, 0);
      }
    }
  }

  /* return the invalid index & value */
  DM_ValueChange(FirstIndex, NULL, &InvalidIndex, 0);
  return &InvalidValue;
}

...

int DML_c AppMain __2((int, argc), (char **,argv))
{
  DM_Value data;

  ...

  data.type = DT_string;
  data.value.string = "NO-VALUE";
  DM_ValueChange(&InvalidValue, NULL, &data, DMF_StaticValue);
  data.type = DT_string;
  data.value.string = "INVALID-INDEX";
  DM_ValueChange(&InvalidIndex, NULL, &data, DMF_StaticValue);

  ...

  DM_StartDialog(...)
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_ValueChange, DM_ValueCount, DM_ValueGet

Method index

Built-in functions countof, itemcount

3.81 DM_ValueInit

With this function a value reference can be converted into a local or global value reference managed by the IDM. This allows the further manipulation of the value by **DM_Value…()** functions and its transfer as parameter or return value.

The value reference is initialized with the appropriate type. The collection data types *DT_list*, *DT_vector*, *DT_hash*, *DT_matrix* and *DT_refvec* are also permitted.

If the value reference is initialized as static or global via the *DMF_StaticValue*option, access is also possible outside the function call. Value lists and strings are not released at the end of the function. The initialization of arguments as static or global managed value references is not allowed.

String values are initialized with the *NULL* pointer. Collections are created without element values. All other value types are also initialized with a *0* value.

The function DM_ValueChange can be used to add or change values or part values respectively elements. A reinitialization using **DM_ValueInit()** is also possible.

```
DM_Boolean DML_default DM_EXPORT DM_ValueInit
(
  DM_Value   *value,
  DM_Type     type,
  DM_Value   *count,
  DM_Options  options
)
```

Parameters

-> DM_Value *value

This is the value reference to be initialized.

-> DM_Type type

This parameter specifies the requested initial type.

-> DM_Value *count

In this parameter the initial size of collections like *list* or *matrix* can be specified or the appropriate value type for *vector* values.

-> DM_Options options

Option	Meaning
0	The value reference will be initialized as a local value.
DMF_StaticValue	The value reference will be initialized as a global, static value.

Return value

DM_TRUE	The function has been completed successfully so the value reference is initialized.
DM_FALSE	The value reference could not be initialized.

Example

Dialog File

```
dialog Dialog

function void     AppendToList(anyvalue List input output, anyvalue Value);
function anyvalue FindMinMax(anyvalue IntegerList);
function string   ListToString(anyvalue List);
function void     SetElemSep(string Sep);

on dialog start
{
   variable vector[string] WeekDays:=["Mo","Tu","Wed","Thu","Fri","Sat"];
   variable list DaysPerWeek := [31,30,28,27];

   SetElemSep(" , ");
   AppendToList(WeekDays,"Sun");
   print ListToString(WeekDays);
   print FindMinMax(DaysPerWeek);
   exit();
}
```

C Part

```
...

static DM_String elemSep = NULL;

void DML_default DM_ENTRY SetElemSep(DM_String sep)
{
  DM_StringInit(&elemSep, DMF_StaticValue);
  DM_StringChange(&elemSep, sep, 0);
}

DM_Value* DML_default DM_ENTRY FindMinMax(DM_Value *IntegerList)
{
  DM_Value subval, minMaxList;  /* managed local values */
  DM_UInt  count;
  DM_UInt  minValueCount=0, maxValueCount=0;
  DM_Value index, mindex, data;  /* unmanaged values */
```

```
  /* initialize local managed values */
  DM_ValueInit(&subval, DT_void, NULL, 0);
  DM_ValueInit(&minMaxList, DT_list, NULL, 0);

  /* determine the itemcount of the list */
  count = DM_ValueCount(IntegerList, NULL, 0);
  index.type = DT_integer;
  index.value.integer = 1;

  while(count>0)
  {
    /* loop through the index 1,2,3,... */
    if (DM_ValueGet(IntegerList, &index, &subval, 0)
        && subval.type == DT_integer)
    {
      if (minValueCount==0 || subval.value.integer<minValue)
      {
        minValueCount++;
        /* store maximum value at [1] in minMaxList */
        mindex.type = DT_integer;
        mindex.value.integer = 1;
        DM_ValueChange(&minMaxList, &mindex, &subval, 0);
      }
      if (maxValueCount==0 || subval.value.integer>maxValue)
      {
        maxValueCount++;
        /* store maximum value at [2] in minMaxList */
        mindex.type = DT_integer;
        mindex.value.integer = 2;
        DM_ValueChange(&minMaxList, &mindex, &subval, 0);
      }
    }
    count--;
    index.value.integer++;
  }
  /* return the minMaxList (without compiler warnings) */
  return DM_ValueReturn(&minMaxList, 0);
}

...

void DML_default DM_ENTRY AppendToList(DM_Value *List, DM_Value *Value)
{
  DM_Value newList;

  /* demonstrate the returning of a list-value by two ways * /
  if (List->type == DT_list || List->type == DT_vector)
```

```
  {
    /* 1) returning a manipulated argument (auto-management) */
    DM_ValueChange(List, NULL, Value, DMF_AppendValue);
  }
  else
  {
    /* 2) creation of a managed local list-value */
    DM_ValueInit(&newList, DT_list, NULL, 0);
    DM_ValueChange(&newList, NULL, List, DMF_AppendValue);
    DM_ValueChange(&newList, NULL, Value, DMF_AppendValue);
    *List = newList;
  }
}

DM_String DML_default DM_ENTRY ListToString(DM_Value *List)
{
  DM_Value  index;
  DM_Value  value;
  DM_UInt   count;
  DM_String retString;

  /* initialize the managed values */
  DM_ValueInit(&index, DT_void, NULL, 0);
  DM_ValueInit(&value, DT_void, NULL, 0);
  DM_StringInit(&retString, 0);

  count = DM_ValueCount(List, NULL, 0);
  while(count>0)
  {
    if (DM_ValueIndex(List, count--, &index, 0)
        && DM_ValueGet(List, &index, &value, 0)
        && value.type == DT_string)
    {
      DM_StringChange(&retString, value.value.string, DMF_AppendValue);
      if (count>0)
        DM_StringChange(&retString, elemSep, DMF_AppendValue);
    }
  }
  return DM_StringReturn(retString, 0);
}
```

Availability

Since IDM version A.06.01.a

3.82 DM_ValueReturn

This function is used to safely return local **DM_Value** values from a function. When local variables and structures are used in a C function, they are invalid after they have been returned. This function can safely and easily return a local **DM_Value** variable.

If necessary, a temporary copy of the value to return is created (e.g. the string in it is copied) for this purpose. There is no copying for managed value references. In this case the returned **DM_Value** pointer should be passed to the caller with `return`.

```
DM_Value* DML_default DM_EXPORT DM_ValueReturn
(
  DM_Value   *value,
  DM_Options  options
)
```

Parameters

-> DM_Value* value

This parameter refers to the value reference that shall be returned. It may be a managed value reference, however this is not mandatory.

-> DM_Options options

Should be set to *0* since no options are available.

Return value

A pointer to a valid **DM_Value** structure is returned or *NULL* in case of an error. An error may occur, for instance, if the function is called in the wrong “runstate”, the managed string is invalid or copying failed.

Please Note

The functions DM_ValueInit and DM_ValueChange can be used to return *output* parameters.

Example

Dialog File

```
dialog YourDialog
function anyvalue StringOf(integer I);

on dialog start
{
  print StringOf(123);
  print StringOf(-42);
  exit();
}
```

C Part

```
...

DM_Value * DML_default DM_ENTRY StringOf(DM_Integer I)
{
  char     buf[10];
  DM_Value data;
  DM_Value negative;

  if (I>=0)
  {
    /* return an unmanaged value */
    sprintf(buf, "%d", (int)I);
    data.type = DT_string;
    data.value.string = buf;
    /* wrong: return &data => data is a local structure! */
    return DM_ValueReturn(&data, 0);
  }
  else
  {
    /* return a managed value */
    DM_ValueInit(&negative, DT_string, NULL, 0);  /* can be omitted */
    data.type = DT_string;
    data.value.string = "!!negative!!";
    DM_ValueChange(&negative, NULL, &data, 0);
    /* return &negative; => possible but generates compiler warning! */
    return DM_ValueReturn(&negative);
  }
}
```

Availability

Since IDM version A.06.01.a

See also

Functions DM_ValueChange, DM_ValueCount, DM_ValueGet, DM_ValueIndex

Method index

Built-in functions countof, itemcount

3.83 DM_WaitForInput

Using this function you can wait for the arrival of a special message without blocking the underlying window system.

This function is only available on MICROSOFT WINDOWS.

```
DM_UInt DML_default DM_EXPORT DM_WaitForInput
(
  DM_UInt msg,
  DM_Options options
)
```

Parameters

-> DM_UInt msg

In this parameter the message to be waited for is transferred.

-> DM_UInt timeout

This parameter indicates in seconds the period of waiting for the arrival of the message. *0* here represents an unlimited waiting time.

-> DM_Options options

For this parameter you have the following possibility:

Option	Meaning
DMF_IgnoreExtEvent	This option means that, during waiting for the specified event, the external events will be ignored. If this option has not been set, an external event finishes the waiting action.

Return Value

Value	Meaning
DMF_RetInput	This value means that the specified message has arrived.
DMF_RetTimeout	This value means that the specified time has been exceeded. This is why a timeout has occurred.
DMF_RetExtEvent	This value means that an external event has arrived.
DMF_RetError	On calling this function an error has arrived. The reason for this may be an invalid message or that the present state of the application does not allow a call. Calls to this function out of callback, canvas and format functions are not permitted.

Note

Calling this function, DM does not process any events until the specified message arrives. By doing so, you risk to overflow the DM internal event processing. Therefore you should use this function very carefully, otherwise the entire application may be shut down.

Example

Asynchronous analysis of a computer name via the function `gethostbyname`.

```
/*
** This function has the control. It enables you to calculate
** the free message, it installs the input handler and then
** calls the asynchronous function gethostbyname.
*/
static struct hostent FAR * TcpWin_gethostbyname __1(
(const char FAR *, name))
{
  HANDLE h = WSAAsyncGetHostByName (TcpWinHwnd,
    TcpWinMsgGetXByY,name,TcpWinBuffer,MAXGETHOSTSTRUCT);
  TcpWinHostent = (struct hostent FAR *) 0;

  if ((h != (HANDLE) 0)
  && (DM_InputHandler (TcpWinGetXByYHandler, (FPTR) 0,
        TcpWinMsgGetXByY, DMF_ModeMsgNotify,
        DMF_RegisterHandler, DMF_DontTrace)
    != (HWND) 0)
    && DM_WaitForInput (TcpWinMsgGetXByY, 0,
    DMF_IgnoreExtEvent | DMF_DontTrace))
    {
      DM_InputHandler (TcpWinGetXByYHandler, (FPTR) 0,
        TcpWinMsgGetXByY, DMF_ModeMsgNotify,
        DMF_WithdrawHandler,
        DMF_DontTrace | DMF_CheckFuncarg);
    }

    return (TcpWinHostent);
}
```

3.84 YiRegisterUserEventMonitor

This function is available only for the MICROSOFT WINDOWS version. It installs an event monitor function which can be used to interrupt the DM event loop.

Changes as of Version A.05.01.d

The monitors "YI_OBJ_MONITOR" or "YI_OBJFRAME_MONITOR" are called for additional Microsoft Windows controls.

If an ISA Dialog Manager object is composed of several Microsoft Windows Controls, the monitor "YI_OBJ_MONITOR" can be called for each of these Microsoft Windows Controls.

```
DM_Boolean DML_default DM_EXPORT YiRegisterUserEventMonitor
(
  int which,
  YiUserEventMonitor uem
)
```

Parameters

-> int which

This parameter defines which monitor functions are to be installed.

The following values are possible:

» *YI_APP_MONITOR*
An application-event monitor has been installed.

» *YI_OBJ_MONITOR*
An object-event monitor has been installed.

» *YI_OBJFRAME_MONITOR*
An frame-object-event monitor has been installed.

-> YiUserEventMonitor uem

This parameter is the address of the monitor function to be installed. Its definition and its use are explained below ("Notes").

Return value

TRUE	Monitor function was successfully installed.
FALSE	Monitor function could not be installed.

Remarks

1. YiRegisterUserEventMonitor installs a new monitor-function pointer. The old monitor function is deleted.
2. If a *NULL* pointer is transferred instead of a monitor-function pointer, the default function will be

installed again.

3. It is possible to install two different monitor functions depending on *which*.

3.84.1 YI_APP_MONITOR

This function is the application monitor which can be installed before the application. This monitoring function receives each message that MICROSOFT WINDOWS sends to the application. It has to be defined as follows:

```
LONG DML_default DM_CALLBACK AppMonitorFunc
(
  DM_ID id,
  MSG *pMsg
)
```

Parameters

-> DM_ID id

Currently not used. Please specify with 0.

-> MSG *pMsg

This parameter is the received message. The monitor function has to process this message in any case! For all messages which are not processed the function "YiDefAppMonitor" executing the standard DM processing has to be called.

It has the following definition:

```
LONG DML_pascal DM_EXPORT YiDefAppMonitor (DM_ID id, MSG *pMsg)
```

Return value

Return value which is expected by MICROSOFT WINDOWS or the return value of "YiDefAppMonitor".

3.84.2 YI_OBJ_MONITOR

This monitor function receives each message that MICROSOFT WINDOWS sends to the DM objects. It has to be defined as follows:

```
LONG DML_pascal DM_CALLBACK ObjectMonitorFunc
(
  DM_ID id,
  MSG *pMsg
)
```

Parameters

-> DM_ID id

This parameter is the identifier of the DM object.

-> MSG *pMsg

This parameter is the arrived message. The monitor function has to process this message in any case! For all messages which are not processed, the function "YiDefObjMonitor" that performs the default DM processing has to be called.

It is defined as follows:

```
LONG DML_pascal DM_EXPORT YiDefObjMonitor (DM_ID id, MSG *pMsg)
```

Return value

Return value which is expected by MICROSOFT WINDOWS or the return value of "YiDefObjMonitor".

Important Remarks

If an error occurs when implementing the monitoring functions, the system will crash. This occurs especially if the default function is not called or the wrong default function is called. Therefore, only self-defined messages or DDE messages are to be processed in the application monitoring, and all other messages are to be passed on to the function "YiDefAppMonitor". An object-monitor function should not be installed if possible. The object canvas is available for the definition of special objects.

If possible, you should not install any object-monitor function.

For the definition of special objects, you should use the object canvas.

3.84.3 YI_OBJFRAME_MONITOR

This monitor function receives every message that Microsoft Windows sends to the MSW -frame windows; this concerns only the DM objects that are composed of several MSW -windows.

This function is defined as follows:

```
LONG DML_pascal DM_CALLBACK ObjectFrameMonitorFunc
(
  DM_ID id,
  MSG *pMsg
)
```

Parameters

-> DM_ID id

This parameter is the identifier of the DM object.

-> MSG *pMsg

This parameter is the received message. The monitor function must process this message in any case! For all messages that are not processed, the function "YiDefObjFrameMonitor" must be called, which executes the standard DM processing.

It is defined as follows:

```
LONG DML_pascal DM_EXPORT YiDefObjFrameMonitor (DM_ID id, MSG *pMsg)
```

Return value

Return value which is expected by MICROSOFT WINDOWS or the return value of "YiDefObjFrameMonitor".

Important Remarks

If an error occurs when implementing the monitor functions, especially if the default function is not called or the wrong default function is called, a system crash will occur. Therefore, only self-defined messages or DDE messages should be processed in the application monitor. All other messages should be passed to the "YiDefObjFrameMonitor" function.

If possible, no object frame monitor function should be installed.

The canvas object is available for defining special objects.

4 Options for the Interface Functions

In the following table, you will find all options which can be specified in the parameter *options* of the DM interface functions. Usually, these options can be connected by a logical “or”, so that a function including more than one valid option can be called.

Option	Function	Meaning
DMF_AcceptChild	DM_GetValue DM_SetValue	Is valid for the attribute AT_options of the canvas on Motif. This option denotes a canvas which can have child objects.
DMF_AppendValue	DM_StringChange DM_ValueChange	Append a data value to a collection (if *index* == *NULL*) or string.
DMF_CheckFuncarg	DM_InputHandler	All function arguments are taken to search for the function to be uninstalled.
DMF_CreateInvisible	DM_CreateObject	The newly created object is created invisibly - independent of how the attribute AT_visible is set in the copy model.
DMF_CreateModel	DM_CreateObject	The object to be newly created is generated as a model.
DMF_DisableHandler	DM_DispatchHandler DM_ErrorHandler DM_InputHandler DM_NetHandler	Deactivates a registered handler function.
DMF_DontFreeLastStrings	DM_GetContent DM_GetValue DM_GetValueIndex DM_GetVectorValue DM_ValueCount DM_ValueGet DM_ValueIndex	Usually, the internal strings are overwritten on every call to the **DM_Get*** or **DM_Value*** function. By this option you can keep the strings valid.

Option	Function	Meaning
DMF_DontTrace	DM_DispatchHandler DM_InputHandler DM_QueueExtEvent DM_SendEvent DM_WaitForInput	Function call is not traced.
DMF_DontWait	DM_EventLoop	Event processing is not carried out "blockingly", i.e. if there is an event, it will be processed; if there is no event, go back immediately to the calling function.
DMF_EnableHandler	DM_DispatchHandler DM_ErrorHandler DM_InputHandler DM_NetHandler	Reactivates a previously deactivated handler function.
DMF_FatalNetErrors	DM_Initialize	Sets a compatible behavior to the IDM versions before A.05.01.d for the DISTRIBUTED DIALOG MANAGERS (DDM), enforcing an immediate termination on client and server side when a network, protocol or version error occurs.
DMF_ForceDestroy	DM_Destroy DM_StopDialog	The option means that the specified object is deleted. On DM_Destroy this option has to be set, if the object is to be deleted.
DMF_GetLocalString	DM_GetValue DM_GetValueIndex DM_GetVectorValue DM_ValueCount DM_ValueGet DM_ValueIndex	Text is returned as string in the language which is currently set.
DMF_GetMasterString	DM_GetValue DM_GetValueIndex DM_GetVectorValue DM_ValueCount DM_ValueGet DM_ValueIndex	Text is returned as string in the original language.

Option	Function	Meaning
DMF_GetTextID	DM_GetValue DM_GetValueIndex DM_GetVectorValue	Text is returned as text ID.
DMF_IgnoreExtEvent	DM_WaitForInput	External events are ignored.
DMF_IncludeIdent	DM_ErrMsgText	The name of the part which has produced the error appears in the error message. This may be the operation system, the window system or DM.
DMF_IncludeModule	DM_ErrMsgText	The name of the module in which the error occurred appears in the error message.
DMF_IncludeSeverity	DM_ErrMsgText	The severity of the error (warning, error, fatal error) is contained in the error text.
DMF_IncludeText	DM_ErrMsgText	The actual error text is contained in the error message.
DMF_InheritFromModel	DM_CreateObject	The object including the children specified for the model are created.
DMF_Inhibit	DM_SetContent DM_SetValue DM_SetValueIndex DM_SetVectorValue	No event is created for the setting of the attribute and thus no rule will be triggered "on Object .Attribute changed".
DMF_InhibitTag	DM_TraceMessage	The beginning of the line "[UM]" is not printed during tracing.
DMF_LogFile	DM_TraceMessage	Output not in tracefile, but in logfile. **Note** If the start option -IDMtracefile is set, everything will always be written in the tracefile. This is also valid if the option DMF_LogFile is set. This option does have the above described effect - output in logfile and not in tracefile - only, if -IDMtracefile is not set!

Option	Function	Meaning
DMF_NoCriticalSection	DM_QueueExtEvent DM_SendEvent	Prevents the function from using a “critical section” on MICROSOFT WINDOWS.
DMF_NoFocusFrame	DM_GetValue DM_SetValue	Is valid for the attribute AT_options of the canvas on Motif. This option denotes a canvas which is meant not to have a focus frame.
DMF_OmitActive	DM_GetContent DM_SetContent	The attribute AT_active is not to be transferred.
DMF_OmitSensitive	DM_GetContent DM_SetContent	The attribute AT_sensitive is not to be transferred.
DMF_OmitStrings	DM_GetContent DM_SetContent	The strings are not transferred.
DMF_OmitUserData	DM_GetContent	The attribute AT_userdata is not filled by DM_GetContent.
DMF_OperationMenu	DM_Control	Controls the display of the operation menu at the windows.
DMF_Printf	DM_TraceMessage	The parameter "string" is interpreted in the same way as in the C function "printf".
DMF_RegisterHandler	DM_DispatchHandler DM_ErrorHandler DM_InputHandler DM_NetHandler	A new handler is installed by means of this option.
DMF_ReplaceFunctions	DM_BindFunctions	A function table existing for the object is replaced completely with a new one.
DMF_SaveAll	DM_SaveProfile	Also saves inherited values of configurable records in the configuration file.
DMF_SetCodePage	DM_Control DM_ControlEx	The specified code page is used in the application and all texts are transformed in this code page.
DMF_SetFormatCodePage	DM_Control DM_ControlEx	Defines the code page in which format functions interpret and return strings.

Option	Function	Meaning
DMF_SetUserCodePage	DM_ControlEx	Defines the character code for **iconv** and thus indirectly influences the IDM code page **CP_ucp**. The code page **CP_ucp** is activated by **DMF_ SetCodePage**. (Only on platforms that support iconv).
DMF_ShipEvent	DM_SetContent DM_SetValue DM_SetValueIndex DM_SetVectorValue	An event is created for the setting of the attribute and in doing so, the rule ".attribute changed" which possibly exists is triggered.
DMF_SignalMode	DM_Control DM_ControlEx	This option specifies the way DM intercepts signals.
DMF_Silent	DM_BindCallBacks	No error messages about missing functions or about too many defined functions are printed.
DMF_SortBinary	DM_ValueChange	Binary sorting of a collection.
DMF_SortLinguistic	DM_ValueChange	Sorting of a collection according to linguistic rules.
DMF_SortReverse	DM_ValueChange	Sorting of a collection in reverse order.
DMF_StaticValue	DM_StringChange DM_StringInit DM_ValueChange DM_ValueGet DM_ValueInit	Conversion into a static or global value reference in the case of automatic conversion into a managed value reference.
DMF_Synchronous	DM_QueueExtEvent DM_SendEvent	This option can optimize internal processes, if the function is not called from a “signal handler”.
DMF_UpdateScreen	DM_Control DM_ControlEx	All actions which are executed internally are made visible on the screen.
DMF_UseUserData	DM_SetContent	In the DM_Content-Vector, the user-data is considered and assigned to the object.

Option	Function	Meaning
DMF_Verbose	DM_BindCallBacks DM_DataChanged	DM_BindCallbacks: Error messages about missing functions or about too many functions are printed DM_DataChanged: Activates tracing of this function.
DMF_WaitForEvent	DM_EventLoop	The function waits for exactly one event and then returns.
DMF_WindowListMenu	DM_Control	Controls the display of the menu of the window lists at windows.
DMF_WithDrawHandler	DM_DispatchHandler DM_ErrorHandler DM_InputHandler DM_NetHandler	This option uninstalls a handler function.
DMF_XlateString	DM_SetValue DM_SetValueIndex DM_SetVectorValue	The specified string is translated in the currently active language, before it is assigned to the object. This is only possible, if the text and the translation exist already.

Index

D

Y