# ISA Dialog Manager

## C Interface - Basics

A.06.03.c

In this manual the basic structure of the API (application programming interface) is depicted that the ISA Dialog Managers offers for applications written in C. The manual describes the data types as well as compiling and linking the applications.


**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 program interface (usually referred to as "API Application Interface") which is offered by the Dialog Manager (DM) for application programs written in C. This manual includes the functions which have to be written by the application developer. For this purpose the relevant datatypes and structures are explained.

This manual is intended for programmers who are familiar with the DM environment as well as with the used C compiler. You should also have good knowledge of the used operating system and the tools provided by the operating system.

Please read chapter "Calling subprograms" thoroughly. You should completely understand the rules describing the datatype mapping.

# 2 Basic Working Method

In this chapter we give an introduction to the basic working method of the C interface. You will get an idea of how programs written with Dialog Manager can be realized.

When realizing the application programs you should always bear in mind Seeheim's layer model which demands a strict separation of the various software layers. The presentation layer as well as the dialog layer can be realized almost completely in the Rule Language when using Dialog Manager.

The C functions should ideally have no knowledge of the interface, but should always receive all necessary information from the dialog in form of parameters, and should not make any calls to Dialog Manager afterwards, i.e. the C functions should not contain any "DM" functions. If functions are realized in this manner, the actual application logic and processing is included in the C functions, which are independent of the used interface. Only extensive accesses to objects with list character (tablefield, listbox and poptext) should be realized by means of special functions of Dialog Manager for reasons of efficiency in C. These functions make it possible to transport a number of data in one call, where normally several single calls are needed.

This procedure is absolutely necessary in any case when using the Distributed Dialog Manager, since function calls via the network are comparatively extensive and thus inefficient.

An exception is the actual main program of the application, because here Dialog Manager functions are accessed and knowledge of the interface is needed.

To be able to write the C program, which fits the developed interface, first some data structures in the C interface have to be clarified and the reproduction of the datatypes in Dialog Manager have to be defined. We are giong to illustrate the principle method by means of an example.

## 2.1 Dialog Definitions

A dialog system is to be created which first offers the user a table-like overview of names and addresses; now you may modify the overview in a sub-window. This overview consists of the first rows in a file. Since this file may contain a lot of rows, only parts of the rows will always be loaded and displayed. To realize such an overview, first a function is required to initialize the table, and second a reloading function is needed which will load new data in the table element when required.

A double-click on the table element is to open a modal detail window in which the data can be changed. According to the action, functions will be called which transfer this data via records as parameters to C functions for further processing.

### 2.1.1 Overview Window

The window is as follows:

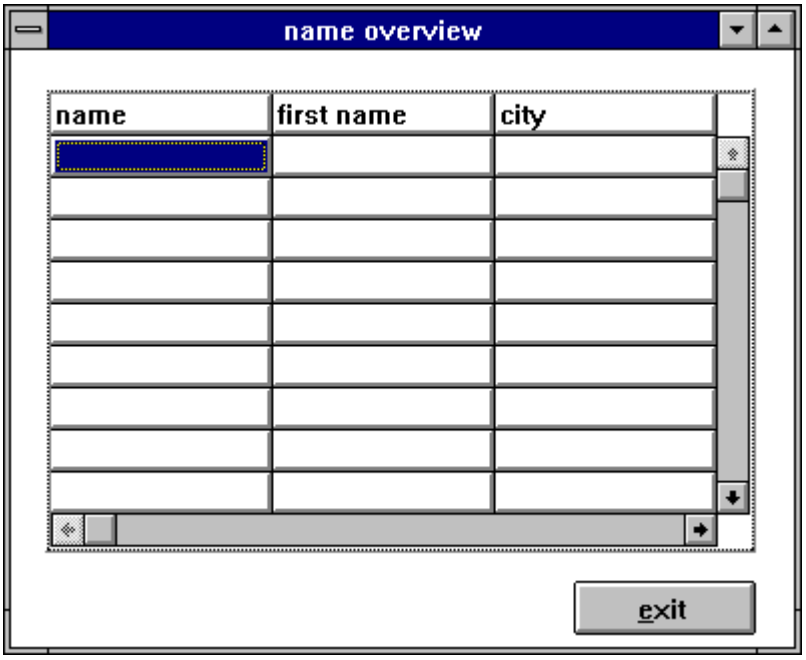


***Figure 1:*** *Window "name overview"*

In the dialog script this window is defined as follows:

```
window WnUebersicht
{
  .visible false;
  .active  false;
  .title   "name overview";
  .xleft   0;
  .width   50;
  .ytop    -2;
  .height  16;
  .iconic  false;
  .iconifyable true;
  child tablefield T1
  {
    .visible true;
    .xauto   0;
    .xleft   1;
    .xright  1;
    .yauto   0;
    .ytop    0;
    .ybottom 1;
    .posraster   true;
    .sizeraster  true;
    .fieldshadow false;
    .contentfunc CONTENT;
    .selection[sel_row]    true;
    .selection[sel_header] false;
```

```
    .selection[sel_single] false;
    .colcount 3;
    .rowcount 30;
    .rowheadshadow true;
    .rowheader 1;
    .colfirst  1;
    .rowfirst  2;
    .colwidth[0]  12;
    .rowheight[0] 1;
    .rowlinewidth[1] 3;
    .content[1,1] "name";
    .content[1,2] "first name";
    .content[1,3] "city";
    .xraster 10;
    .yraster 16;
  }
  child pushbutton PENDE
  {
    .xleft 36;
    .width 11;
    .yauto -1;
    .text  "&exit";
  }
}
```

### 2.1.2 Detail Window

The complete data belonging to a name is represented in a subwindow which has been defined as a dialogbox in which the user can modify the data.

The window in which data can be modified is as follows:

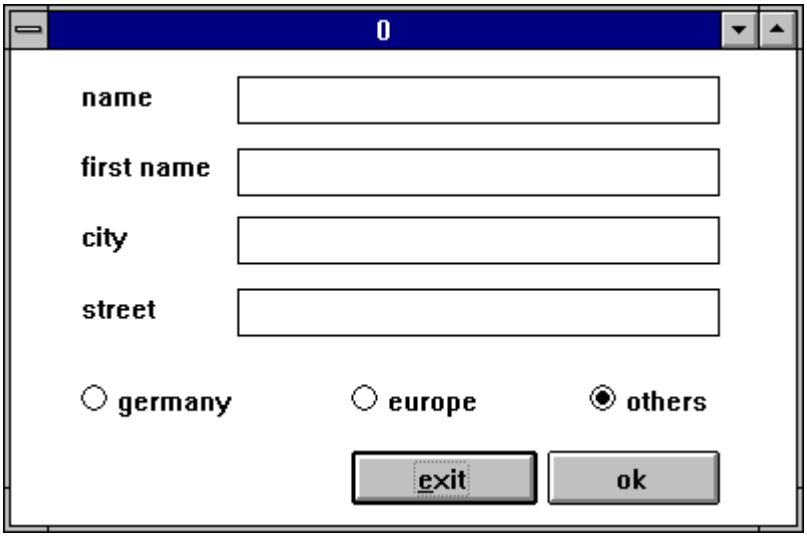


***Figure 2:*** *Detail Window*

The definition for the window in the script is the following:

```
window WnName
{
  .visible false;
  .active  false;
  .title   "0";
  .xleft   110;
  .width   50;
  .ytop    80;
  .height  13;
  .dialogbox   true;
  .iconifyable true;
  integer AktivesLand := 0;
  child Eintrag Lname
  {
    .ytop 0;
    .S.text "name";
    .E.active  false;
    .E.content "";
  }
  child Eintrag Lfirstname
  {
    .ytop 2;
    .S.text "first name";
    .E.content "";
  }
  child Eintrag Lcity
  {
    .ytop 4;
    .S.text "city";
```

```
  .E.content "";
}
child Eintrag Lstreet
{
  .ytop 6;
  .S.text "street";
  .E.content "";
}
child pushbutton PbAbbrechen
{
  .xleft 22;
  .width 11;
  .ytop  10;
  .text  "&exit";
}
child pushbutton PbOk
{
  .xauto  -1;
  .width  12;
  .xright 1;
  .ytop   10;
  .text   "OK";
}
child radiobutton GERMANY
{
  .active true;
  .text   "Germany";
  .xleft  3;
  .ytop   8;
  .posraster  true;
  .sizeraster true;
  .userdata   1;
  .LandesCode := 1;
}
child radiobutton EUROPE
{
  .active false;
  .text   "europe";
  .xleft  19;
  .ytop   8;
  .posraster  true;
  .sizeraster true;
  .userdata   2;
  .LandesCode := 2;
}
child radiobutton OTHER
{
```

```
    .active false;
    .text   "others";
    .xleft  33;
    .ytop   8;
    .posraster  true;
    .sizeraster true;
    .userdata   3;
    .LandesCode := 3;
  }
}
```

For an easier definition of this window a model has been introduced which consists of a groupbox with a statictext and an edittext as children. Instances of this model serve as children of the detail window. The corresponding definition in the dialog script is as follows:

```
model groupbox Eintrag
{
  .xleft  2;
  .width  45;
  .height 2;
  .borderwidth 0;
  child statictext S
  {
    .sensitive false;
    .xleft 0;
    .ytop  0;
  }
  child edittext E
  {
    .xleft 12;
    .width 30;
    .ytop  0;
  }
}
```

### 2.1.3 Messagebox

A messagebox will inform you once all entries have been read from the file.

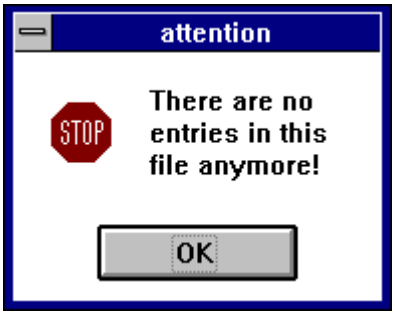


*Figure 3: Messagebox*

```
messagebox Mb
{
  .text  "There are no\nentries in this\nfile anymore!";
  .title "attention";
  .icon  icon_error;
  .button[1] button_ok;
  .button[2] nobutton;
  .button[3] nobutton;
}
```

## 2.2 Function Definitions in the Dialog Script

For the communication with the C program the following functions have been defined:

### FILLTAB()

By means of this function the tablefield is initially filled partly.

```
function c boolean FILLTAB(object  Table  input,
                           integer Number input,
                           integer Header input);
```

### CONTENT()

This function is the reloading function which is to reload the data in the tablefield.

```
function contentfunc CONTENT();
```

### FILECLOSE()

This function closes the used file when the dialog is finished.

```
function boolean FILECLOSE();
```

### GETADDR()

This function loads the entire data belonging to a row and is to transfer it to the dialog.

```
function c boolean GETADDR(record Address input output);
```

**PUTADDR()**

This function accepts the data changed by the dialog and saves it.

```
function c boolean PUTADDR(record Address input);
```

## 2.3 Definition of Records

The functions GETADDR and PUTADDR contain one record each as parameter. The elements of the parameter are defined as links to the corresponding elements in the window.

```
record Address
{
  string  NName      shadows Lname.E.content;
  string  FirstName  shadows Lfirstname.E.content;
  string  City       shadows Lcity.E.content;
  string  Street     shadows Lstreet.E.content;
  integer Country    shadows WnName.AktivesLand;
}
```

## 2.4 Rules

To call the functions, several rules have been defined.

### 2.4.1 Rules for the Program Start

On starting the program the tablefield is filled partially and the relevant window is made visible.

```
on dialog start
{
  FILLTAB(T1, 20);
  T1.rowcount := 50;
  WnUebersicht.visible := true;
}
```

### 2.4.2 Rules for the Program End

By selecting the Exit pushbutton the program will be quit. The program will also be quit, if the main window is closed by the Close entry in the system menu.

```
on PENDE select
{
```

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

### 2.4.3 Rules for the Radiobuttons

If one of the radiobuttons is selected, its value will be noted for the relevant window.

```
on TABLEDEMO.RADIOBUTTON select
{
  this.window.AktivesLand := this.LandesCode;
}
```

### 2.4.4 Rules for the Cancel Button

On selecting the Cancel pushbutton the relevant window will be closed.

```
!! If the Abbrechen pushbutton is selected,
!! the affiliated window will be made invisible.
on PbAbbrechen select
{
  this.window.visible := false;
}
```

### 2.4.5 Rules for the OK Button

If the OK pushbutton is selected, the function PUTADDR will be called and the data of the window is transferred to this function by means of records.

```
!! If the OK button is selected,
!! the changes are transferred to the C program
!! and the window will be made invisible.
on PbOk select
{
  PUTADDR(Address);
  this.window.visible := false;
}
```

### 2.4.6 Rules for the Double Click in the Tablefield

If the tablefield is selected by a double click, all data belonging to this row will be retrieved from the file by the function GETADDR and will be transferred to the *shadows* reference. Then the detail window will be opened.

```
!! If a double-click is made in the table element's interior,
!! the corresponding entry should be editable
!! in the separate window.
!! To do so, the C function must be called,
!! which fetches the data and transfers it to the window.
on T1 dbselect
{
  !! Check first whether there really is
  !! a selected item in the table.
  if (first(this.activeitem) > 1) then
    Address.NName     := this.content[first(this.activeitem), 1];
    Address.FirstName := this.content[first(this.activeitem), 2];
    Address.City      := this.content[first(this.activeitem), 3];
    !! Take this entry and transfer it.
    GETADDR(Address);
    WnName.title   := "Name Nummer: " + itoa((first(this.activeitem) - 1));
    WnName.visible := true;
  else
    !! No useful entry found, make a tone.
    beep();
  endif
}
```

### 2.4.7 Rules for the File End

If all values have been read from the file, the C program sends an external event to Dialog Manager. Then a messagebox will appear on the screen which informs the user.

```
on T1 extevent 9999
{
  querybox(Mb, this.window);
}
```

## 2.5 Definition of the C Programs

To be able to write a C program for this defined surface in DM, you have to know the following:

- the important DM data structures which are used for the communication between the C program and Dialog Manager
- how DM data types are mapped onto C data types

- how the DM definition is turned into a definition in the C program
- how the main program looks which is to start a DM dialog
- how the C subprograms look which have been called out of Dialog Manager
- how C functions are linked to Dialog Manager.

Each of these points are explained in the following chapters.

## 2.5.1 Communication

### 2.5.1.1 Parameters

Dialog Manager offers above all the possibility to transfer up to eight parameters to a function. This restriction is weakened, considering that also so-called records can be transferred. These records are similar to structures in C, since the users themselves can define the structure and contents of these records and can thus adopt them to their needs. Thus several data can be transferred as a bundle.

The parameters can be transferred in three different ways:

- If only the value of a parameter is needed in the application (“call by value”), it can be transferred as an input parameter (default setting).
- If the contents of a parameter is to be set in the function (“call by reference”), it has to be defined as output parameter.
- If a parameter is filled partly (e.g. a record) and is to be filled further on, it is an input as well as an output parameter.

The terms “input” and “output” are always considered from the perspective of the application.

When using records as parameters, please also refer to the chapter “Functions with Records as Parameters”.

### 2.5.1.2 Return Values

The functions in C have data types themselves and can give as a result a return value. Even with Dialog Manager there is the possibility to declare functions of an integral data type to return values. A `function c boolean PUTADDR(record Address input);` can return either *true* or *false* as a return value.

## 2.5.2 Mapping of the Dialog Data Types

Based on the dialog description the used data types can be mapped on data types which can be used in C. This mapping from the dialog to C is unique and is shown in the following incomplete table.

| Dialog Data Type | C Data Type |
|---|---|
| *object* | *DM_ID* |
| *integer* | *DM_Integer* |
| *string* | *DM_String* |
| *boolean* | *DM_Boolean* |

### 2.5.3 The Main Program

Programs which are to work with Dialog Manager require special main programs; in principle, these main programs are always the same and can therefore be copied from the provided examples.

On the construction of the main program you have to distinguish whether you develop a local application or a server in a distributed environment. For a local application you have to provide a function called **AppMain()**, for a distributed application you have to provide two functions called **AppInit()** and **AppFinish()**.

These main programs must include the header files created by Dialog Manager in order to be able to access the definitions of Dialog Manager.

#### 2.5.3.1 Local Applications

The structure of the function **AppMain()** is as follows:

- First the function DM_Initialize has to be called to initialize Dialog Manager. This function absolutely has to be the first function to be called in the C program. All other functions will lead to errors.
- After having initialized Dialog Manager the dialog belonging to the application can be loaded by means of the function DM_LoadDialog.
- When the dialog was loaded successfully, the functions included in the dialog have to be transferred from the C program to Dialog Manager. This is done by calling the function DM_BindFunctions, which is meant for functions which have no records as parameters and via the C function *RecMInit<name of dialog or module>*, which is meant for functions which have records as parameters.

The addresses are linked to the function calls.

- After having linked the functions to the dialog, application-specific initializations have to be carried out.
- Afterward the dialog for the user has to be started by means of the function DM_StartDialog. On calling this function the starting rule in the dialog on `dialog start` has to be carried out and all windows which have been defined as visible in the dialog are made visible.
- Finally the control is transferred to Dialog Manager by calling the function DM_EventLoop. This function normally returns only at the program end, after processing the rule on `dialog finish`, if

available. Instructions after the call of this function will thus be carried out only on finishing the application.

This function is called **AppMain()** and roughly is as follows:

```
int DML_c AppMain __2((int, argc), (char far * far *, argv))
{
  DM_ID dialogID;

  /* Initialization of Dialog Manager */
  if (!DM_Initialize (&argc, argv, 0))
  {
    DM_TraceMessage("could not initialize", DMF_LogFile);
    return (1);
  }

  /* Loading the dialog file */
  switch(argc)
  {
    case 1:
      dialogID = DM_LoadDialog ("bsp.dlg", 0);
      break;
    case 2:
      dialogID = DM_LoadDialog (argv[1], 0);
      break;
    default:
      DM_TraceMessage("too many arguments", DMF_LogFile);
      return(1);
    break;
  }
  if (!dialogID)
  {
    DM_TraceMessage("could not load dialog", DMF_LogFile);
    return(1);
  }

  /* Entry of the function's addresses without records */
  DM_BindFunctions (FuncMap, FuncCount, dialogID, 0, 0);

  /* Entry of the function's addresses with records */
  RecMInitTABLEDEMO(dialogID, 0);

  /* Dialog start and entering the event loop */
  if (DM_StartDialog (dialogID, 0))
    DM_EventLoop (0);
  else
    return (1);
```

```
  return (0);
}
```

### 2.5.3.2 Distributed Applications

For distributed applications two C functions which can initialize or finish the application have to be provided in the application. The initializing function is called **AppInit()**, the finishing function is called **AppFinish()**.

The structure of the function **AppInit()** is as follows:

- This function has to transfer the functions from the C program to Dialog Manager. This is done by calling the function DM_BindFunctions and the C function *RecMInit<name of dialog or module>* and the C function created by means of the simulation program for functions having a record as parameter.
- After having linked the functions to the application, application-specific initializations have to be carried out.

In the function **AppFinish()**, those actions which bring about a controlled finishing of an application have to be carried out. Calls to Dialog Manager are not needed here, the server application will be finished automatically after the function has been returned.

#### Example

*Function AppInit()*

```
int DML_c DM_CALLBACK AppInit __4((DM_ID, appl), (DM_ID, dialog),
                                  (int, argc), (char far * far *, argv))
{
  DM_BindFunctions(ApplFuncMap, ApplFuncCount, appl, 0, DMF_Silent);
  return 0;
}
```

*Function AppFinish()*

```
int DML_c DM_CALLBACK AppFinish __2((DM_ID, appl), (DM_ID, dialog))
{
  return 0;
}
```

## 2.5.4 Auxiliary Means for the C Programming

In order to avoid having to define twice the definitions set in the dialog for the parameters of the C functions, the simulator of Dialog Manager can generate the data needed for the C program from a dialog file. This data should by no means be changed since it represents the interface between Dialog Manager and the application.

The prototypes of the functions and, if needed, the mappings of the records of Dialog Manager on structures for the C program are defined in the files. The definitions of the prototypes should be used as function definitions in the C program to guarantee a consistent and flawless function call.

### 2.5.4.1 Functions without Records as Parameters

The simulation program of Dialog Manager creates an include file which contains all prototypes of the functions defined in Dialog Manager. This file can then be linked to the corresponding C sources. The corresponding C compiler checks the function definitions during the realization of the function and then informs about differences.

By using the command line option **+writeproto <name of header file>**.

the prototype files are created.

For our example the command line is as follows:

```
idm  +writeproto  bsp.h  bsp.dlg
```

By using this call, the file **bsp.h**. is created.

```
DM_Boolean DML_default DM_ENTRY FILLTAB __((DM_ID Table,
                                            DM_Integer Number,
                                            DM_Integer Header));

void DML_default DM_CALLBACK CONTENT __((DM_ContentArgs *args));

DM_Boolean DML_default DM_ENTRY FILECLOSE __((void));
```

Instead of this option you could also use the option **+writefuncmap**, which creates the prototypes as well as the function table. The function table is put into a C file so that the included function *BindFunctions_<name of dialog>* would have to be called to link the functions.

In the function **AppMain()** the functions, which are to be provided for Dialog Manager, are registered in the “FuncMap”. This is a structure which provides the addresses of the functions in order to be able to access these definitions in the C program in Dialog Manager. The defined variable “FuncCount” indicates the number of defined functions in the table. Note that only those functions which have no records as parameters are registered in **AppMain()**, because those are linked to the dialog by a different mechanism.

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

static DM_FuncMap far FuncMap[] = {
  { "CONTENT",  (DM_EntryFunc) CONTENT   },
  { "FILLTAB",  (DM_EntryFunc) FILLTAB   },
  { "FILECLOSE",(DM_EntryFunc) FILECLOSE },
};
```

### 2.5.4.2 Functions with Records as Parameters

Via a command line option the IDM simulation program creates an include file. This file contains the prototypes of the functions defined in DM with records as parameters and the function needed for the call. The C file being created has to be compiled and linked to the application. The include file being created should be used for the actual function binding, for here also the structures are defined.

This is done by means of the command line option **+writetrampolin <base name> <dialog name>**

Base name indicates the name of a file to which the suffixes ".h" and ".c" are attached in order to create the files needed to call C functions. In the **.h** file, the prototypes of functions and the mappings of records on C structures, among other things, are defined.

In the **.c** file first the functions with records as parameters are registered in the *RecMap[]* and thus are introduced to Dialog Manager.

Second, the functions which have to be called in the function **AppMain()**, are created to bind the functions which are defined here to the main dialog.

For our example the command line is as follows:

```
idm +writetrampolin  tram_bsp  bsp.dlg
```

By using this call the following files are created:

- **tram_bsp.h**
- **tram_bsp.c**

The include file **tram_bsp.h** in our example is as follows:

```
#ifndef _INCLUDED_bsp_tram_
#define _INCLUDED_bsp_tram_

typedef struct {
  DM_String  NName;
  DM_String  FirstName;
  DM_String  City;
  DM_String  Street;
  DM_Integer Country;
} RecAddress;

DM_Boolean DML_default DM_ENTRY GETADDR __((RecAddress *));
DM_Boolean DML_default DM_ENTRY PUTADDR __((RecAddress *));
DM_Boolean RecMInitTABLEDEMO __((DM_ID dialog, DM_ID modID));

#endif
```

The created C and header files have to be compiled and linked to the application. The header files should be used to be able to access the definitions of the transfer structure set in the dialog and to adopt the definitions of the functions into the C program.

### 2.5.5 Functions for the Tablefield

Functions which fill or query the object contents with list characters are an exception with regard to the layer model. In this case a **DM_** function is reasonable in order for the applications to run effectively. The tablefields have to be filled by means of a vector. This area is then filled by the application and assigned to the object. By doing so, a constant flicker of the tablefield during filling will be avoided at local applications, and for distributed applications additionally a considerable increase in performance will be achieved compared to the single assignments. In detail the method is as follows:

- If the amount of data is fixed, this memory area can be allocated statically. Otherwise, a dynamic area can be created by means of the function DM_Malloc.
- After creating the memory area the attributes to be set are filed in this area.
- The data is transferred to the object of Dialog Manager in a parameter of the type **DM_VectorValue** in the element *vector*.
- The DM object can then be filled by means of the functions DM_SetVectorValue or DM_SetContent. To do so, the **DM_value** structures must contain the type *DT_index* for two-dimensional objects or *DT_integer* for one-dimensional objects and indexes of those elements which are to be set first and last. Note that the function **DM_SetVectorValue()** can set only rectangular areas and therefore also the number of the objects (*count* in **DM_VectorValue**) to be set have to correspond with these indexes. The actual values are transferred in the structure **DM_VectorValue**. It includes the data type, the number of attributes to be set and the vector of the contents to be set.

The values are read out analogously to the query of the values in the object:

- getting the values from the object by means of the function DM_GetVectorValue
- access to the structure by normal C
- releasing the temporary area via the function DM_FreeVectorValue.

#### 2.5.5.1 Function FILLTAB()

This function handles the initial filling of the tablefield. For this purpose the tablefield's ID and the number of elements to be filled are passed to this function.

```
/*
 * Function for filling the tablefield on dialog start
 * Parameters:
 *  table  is the ID of the tablefield which is filled
 *  number is the number of data sets which are loaded
 *  header is the number of headers of the tablefield
 * Return value:
 *  Boolean value indicating if the data was loaded successfully
 */
DM_Boolean DML_default DM_ENTRY FILLTAB __3((DM_ID, table),
                                            (DM_Integer, number),
                                            (DM_Integer, header))
```

```
{
  DM_Boolean retval = DM_FALSE;
  int i;
  int spos;
  int vpos;
  int length;

  static char ** strvec;
  static char *  buffer;
  static char *  start;
  static char *  end;

  DM_VectorValue vector;
  DM_Value startIdx;
  DM_Value endIdx;

  strvec = (char **) DM_Malloc ( (ROWS * COLUMNS) * sizeof (char*) );
  buffer = (char *) DM_Malloc (STR_LEN * sizeof (char) );

  if (! (f = fopen("c_bsp.dat", "r+")) )
  {
    DM_TraceMessage("Cannot open In-File", DMF_LogFile);
    return(retval);
  }

  /* first cell of tablefield which is filled */
  startIdx.type = DT_index;
  startIdx.value.index.first = (ushort) header;
  startIdx.value.index.second = 1;

  /* last cell of tablefield which is filled */
  endIdx.type = DT_index;
  endIdx.value.index.first = (ushort) number;
  endIdx.value.index.second = COLUMNS;

  spos = 0;
  vpos = 0;

  vector.type = DT_string;
  vector.vector.stringPtr = strvec;

  while (!feof(f) && (spos++ < ROWS))
  {
    if (fgets(buffer, STR_LEN - 1, f))
    {
      i = 0;
      end = buffer;
```

```
      while (*end && isspace(*end))
        end++;
      while (*end && (i++ < COLUMNS))
      {
        start = end;
        length = 1;
        while (*end && !isspace(*end))
        {
          length++;
          end++;
        }
        if (*end)
          *end++ = '\0';
        strvec[vpos] = (char *) DM_Malloc((length + 1)* sizeof(char));
        strcpy(strvec[vpos],start);
        vpos++;
        while (*end && isspace(*end))
          end++;
      }
      while (i++ < COLUMNS)
      {
        strvec[vpos] = (char *) DM_Malloc(sizeof(char));
        strvec[vpos++] = "\0";
      }
    }
  }
  vector.count = (ushort) vpos;
  if (DM_SetVectorValue(table, AT_field, &startIdx,
      (DM_Value *) 0, &vector, 0))
  {
    retval = DM_TRUE;
  }

  /* Release only with Dialog Manager functions! */
  while (--vpos >= 0)
  {
    DM_Free(strvec[vpos]);
  }
  DM_Free(buffer);
  return retval;
}
```

### 2.5.5.2 Function CONTENT()

This function is responsible for reloading the tablefield. If you scroll in an area which has not been filled yet, Dialog Manager will call this function automatically.

The parameters are fixed by the DM and therefore cannot be changed. These reloading functions receive as parameters the structure **DM_ContentArgs**, in which the necessary information is saved.

In *object* the ID of the tablefield is transferred. The fields *visfirst* and *vislast* contain the first and last visible row, the fields *loadfirst* and *loadlast* contain the rows to be loaded first and last. These, however, are only minimal values, the application may at any time load more than the indicated rows. In the following example always five rows are loaded.

```
/*
 * Function for filling dynamically the tablefield at the scroll event,
 * if rows in the tablefield are reached which have not been filled yet.
 */
void DML_default DM_CALLBACK CONTENT __1((DM_ContentArgs *, args))
{
  int i;
  int spos;
  int vpos;
  int length;

  static char ** strvec;
  static char *  buffer;
  static char *  start;
  static char *  end;

  DM_VectorValue vector;
  DM_Value startIdx;
  DM_Value endIdx;

  /* first cell of tablefield which is filled */
  startIdx.type = DT_index;
  startIdx.value.index.first = args->loadfirst - 1;
  startIdx.value.index.second = 1;

  /* last cell of tablefield which is filled */
  endIdx.type = DT_index;
  endIdx.value.index.first = startIdx.value.index.first +
                             (ushort)CONT_ROWS - 1;
  endIdx.value.index.second = COLUMNS;

  vector.type = DT_string;
  vector.vector.stringPtr = strvec;

  spos = 0;
  vpos = 0;

  /*
   * Reading in the contents and editing of data
   * HERE: DELETED!
```

```
     */

    vector.count = (ushort) vpos;

    DM_SetVectorValue(args->object, AT_field,
                      &startIdx, &endIdx,
                      &vector, 0);

    /*
     * Release of the memory areas allocated by Dialog Manager
     * only by means of Dialog Manager functions!
     */
    while (--vpos >=0 )
      DM_Free(strvec[vpos]);

    DM_Free(buffer);

    /*
     * If the end of the file should be reached, the Dialog Manager will be
     * notified of it through an external event so that the DM can react.
     * Here a messagebox is output.
     * As you should do without DM_ functions in callback functions,
     * we show you an example how the Dialog Manager can react to events.
     * The DM_QueueExtEvent function is the only function
     * which is processed safely.
     */
    if(feof(f))
    {
      DM_Value *null = 0;
      DM_QueueExtEvent (args->object,9999,0,null, DMF_DontTrace);
    }
}
```

### 2.5.5.3 Function FILECLOSE()

This is a function for closing a file when the dialog is finished.

```
DM_Boolean DML_default DM_ENTRY FILECLOSE __((void))
{
  if (! (fclose (f)) )
    return (DM_TRUE);
  return (DM_FALSE);
}
```

### 2.5.6 Functions with Records as Parameters

In contrast to the functions for the tablefield the following functions can do without knowledge about the dialog and even should do so. These functions are therefore written without any call to Dialog Manager in standard C. Only the entry in the “FuncMap” shows that they are functions which are called by Dialog Manager.

#### 2.5.6.1 Function GETADDR()

This function is to provide the remaining set of data for a given name on the surface. To be able to access the definitions of the dialog, the corresponding C structure has been created by means of **+writetrampolin**.

```
/*
 * Function to fetch the complete data from a data set selected in
 * a tablefield. In this case a function with file processing should
 * be written. To facilitate this process the record in question is
 * filled with a fixed value.
 * Parameter:
 *  Address is a record which has been partly filled in Dialog Manager
 *             and which is completed by the missing entries ("street")
 * Return value:
 *  Boolean value which indicates if the data has been fetched successfully
 */
DM_Boolean DML_default DM_ENTRY GETADDR __1((RecAddress *, Address))
{
  Address->Street = "new street";
  return (DM_TRUE);
}
```

#### 2.5.6.2 Function PUTADDR()

This function is to save the data changed by the user. To be able to access the definitions of the dialog, an appropriate C structure has been created by means of the simulation program via the option **+writetrampolin**.

```
/*
 * Function to write in the file the data changed in the dialog.
 * This function is not programmed here.
 */
DM_Boolean DML_default DM_ENTRY PUTADDR __1((RecAddress *, Address))
{
  Address->Street = "";
  return(DM_TRUE);
}
```

## 2.6 Compiling and Linking

Finally the created sources have to be compiled. The individual compiler options are dependent on the environment. You should use the makefiles provided in the examples and adapt them to the local conditions.

In principle the following steps are needed to build an executable application:

1. By means of the DM simulation program the necessary data is generated.
   - » For dialogs with functions without records as parameters:

     ```
     idm  +writeproto  <header name>  <dialog name>
     ```

     here

     ```
     idm  +writeproto  bsp.h  bsp.dlg
     ```

   - » For dialogs with functions with records as parameters:

     ```
     idm  +writetrampolin  <base name>  <dialog name>
     ```

     here

     ```
     idm  +writetrampolin  bsp_tram  bsp.dlg
     ```

2. The created headers and the C file are compiled by means of the C compiler and the usual DM compiler options.
3. Afterward the compiled files are linked to an executable.

For an executable dialog at least one C executable and one dialog file of Dialog Manager (binary or ACII file) are necessary.

# 3 Data Types

For the communication between Dialog Manager and application different data types are defined. These data types are made available by binding the file **IDMuser.h**. They are responsible for the transfer of values to the DM as well as for returning the values of the DM to the application. These data types are then used within composed structures and with function parameters. These data types are necessary since the definitions are often not compatible with other tools.

In the next chapter first the basic data types, then the DM data types derived from the basic data types and finally the composed structures are introduced.

## 3.1 Basic Data Types

In the following chapters all basic data types defined by Dialog Manager and their meanings are introduced. These basic data types are defined to file the same contents on all platforms supported by Dialog Manager. They make sure that, for example, a number is always within the range from -2147483648 to 2147483647, independently of the integer number on the corresponding architecture.

### 3.1.1 Basic Data Type DM_Int

By means of this data type an "int" data type is defined the same way as is done in the normal C. The size of this data type depends on the used basis system; 2 bytes on PCs, and 4 bytes on most of the Unix systems.

**Definition**

```
typedef int DM_Int;
```

### 3.1.2 Basic Data Type DM_UInt

By means of this data type an "int" data type without sign is defined. The size of this data type depends on the used basic system; 2 bytes on PCs, and 4 bytes on most of the Unix systems.

**Definition**

```
typedef unsigned int DM_UInt;
```

### 3.1.3 Basic Data Type DM_Int1

By means of this data type a number of 1 byte with sign is defined. Thus the number has a value range from -128 to 127.

**Definition**

```
typedef char  DM_Int1;
```

### 3.1.4 Basic Data Type DM_UInt1

By means of this data type a number of 1 byte without sign is defined. The number thus has a value range from 0 to 255.

#### Definition

```
typedef unsigned char  DM_UInt1;
```

### 3.1.5 Basic Data Type DM_Int2

By means of this data type a number of 2 bytes with sign is defined. The number has a value range from -32768 to 32767.

#### Definition

```
typedef short DM_Int2;
```

### 3.1.6 Basic Data Type DM_UInt2

By means of this data type a number of 2 bytes without sign is defined. It has a value range from 0 to 65535.

#### Definition

```
typedef short DM_UInt2;
```

### 3.1.7 Basic Data Type DM_Int4

By means of this data type a number of 4 bytes with sign is defined. It has a value range from -2147483648 to 2147483647.

#### Definition

On hardware architectures with 2- or 4-byte addressing, the definition is as follows:

```
typedef long DM_Int4;
```

On hardware architectures with 8-byte addressing (e.g. DEC Alpha), the definition is as follows:

```
typedef int DM_Int4;
```

### 3.1.8 Basic Data Type DM_UInt4

By means of this data type a number of 4 bytes without sign is defined. It has a value range from 0 to 4294967295.

#### Definition

On hardware architectures with 2- or 4-byte addressing, the definition is as follows:

```
typedef unsigned long DM_UInt4;
```

On hardware architectures with 8-byte addressing (e.g. DEC Alpha), the definition is as follows:

```
typedef unsigned int DM_UInt4;
```

### 3.1.9 Basic Data Type FPTR

By means of this data type a pointer is defined on any area you want. This pointer depends on the system, but is at least a 4-byte address. Its definition depends on the basis system.

**Definition**

```
typedef void far *FPTR;
```

or

```
typedef char far *FPTR;
```

## 3.2 Dialog Manager Data Types

In the following chapters all datatypes defined by Dialog Manager and their meanings are introduced. These datatypes occur as types of parameters of interface and application functions as well as elements of the composed structures defined by Dialog Manager.

### 3.2.1 Data Type DM_Attribute

By means of this datatype you can define how to store attributes.

**Definition**

```
typedef DM_UInt4  DM_Attribute;
```

### 3.2.2 Data Type DM_Boolean

By means of this datatype a boolean value is defined which can have the value TRUE or FALSE.

**Definition**

```
typedef DM_UInt  DM_Boolean;
```

### 3.2.3 Data Type DM_Class

By means of this datatype you can define how to store a class.

**Definition**

```
typedef DM_UInt  DM_Class;
```

### 3.2.4 Data Type DM_Enum

By means of this datatype you can define how to store the enumeration type.

**Definition**

```
typedef DM_UInt  DM_Enum;
```

### 3.2.5 Data Type DM_ErrorCode

By means of this datatype you can define how to store internally the error codes.

**Definition**

```
typedef DM_UInt4  DM_ErrorCode;
```

### 3.2.6 Data Type DM_Event

By means of this datatype you can define how to store the number of an event.

**Definition**

```
typedef DM_UInt  DM_Event;
```

### 3.2.7 Data Type DM_Integer

By means of this datatype you can define a number of 4 bytes.

**Definition**

```
typedef DM_Int4  DM_Integer;
```

### 3.2.8 Data Type DM_ID

By means of this datatype you can define how to store objects internally.

**Definition**

```
typedef DM_UInt4  DM_ID;
```

### 3.2.9 Data Type DM_Method

By means of this datatype you can define how to store a method.

**Definition**

```
typedef DM_UInt4  DM_Method;
```

### 3.2.10 Data Type DM_Options

By means of this datatype you can define how the options of the interface functions are to be transferred to Dialog Manager.

**Definition**

```
typedef DM_UInt  DM_Options;
```

### 3.2.11 Datatype DM_Pointer

By means of this datatype you can define how to store a pointer on a structure of your choice.

**Definition**

```
typedef FPTR  DM_Pointer;
```

### 3.2.12 Data Type DM_Scope

By means of this datatype you can define how to store the type of an object (instance, model or default).

**Definition**

```
typedef DM_UInt  DM_Scope;
```

### 3.2.13 Data Type DM_String

By means of this datatype you can define a pointer on a string.

**Definition**

```
typedef char *  DM_String;
```

### 3.2.14 Data Type DM_Type

By means of this datatype you can define how to store the datatype of an attribute.

**Definition**

```
typedef DM_UInt  DM_Type;
```

### 3.2.15 NULL-DM_ID

By means of DM_ID 0 you can reset resource attributes. This DM_ID is called "NULL object". This NULL object can be used to reset all resource attributes. DM_SetValue may receive the NULL object and can be used for DM_GetValue or for function arguments.

The NULL object itself is type-independent, i.e. there is only one NULL object for all resources and objects. The NULL object can be used on all datatypes which are an object internally. With GetValue the real type of the NULL object depends on the attribute type.

#### Effects

- Interface functions can have a NULL object as return value. The NULL object can appear as ("input" and "output") parameter.
- If DM_GetValue receives a NULL object as return value, it will always have the most exact data-type possible, e.g. DT_text instead of DT_instance.
- If an attribute is to be set to null by the application, the datatype has to be indicated as exactly as possible, e.g. DT_accelerator instead of DT_instance.

## 3.3 Composed Structures for Setting and Querying Attributes

In the following chapters all structures defined by Dialog Manager and their meanings are introduced. These structures are responsible for the communication between the application and Dialog Manager.

### 3.3.1 Structure DM_Index

By means of this structure a two-dimensional index value is described. This value is used if elements in the interior of the tablefield are accessed.

#### Definition

```
typedef struct {
  DM_UInt2 first;
  DM_UInt2 second;
} DM_Index;
```

#### Meaning of the Elements

**DM_UInt2 first**

In this element the first part of the index is filed.

**DM_UInt2 second**

In this element the second part of the index is filed.

### 3.3.2 Structure DM_ValueUnion

By means of this structure the data in the dialog can be queried as well as set. With this structure, you can query and set exactly o n e attribute value in the dialog. This structure is defined as union, i.e. there is always one valid value within this structure, all other structures are not accessible at this moment.

```
typedef union {
#if defined(DM_COMPAT_CARDINAL)
#define DM_Cardinal DM_UInt
  DM_Cardinal   cardinal;
#endif
  DM_Scope      scope;
  DM_Boolean    boolean;
  DM_String     string;
  DM_Integer    integer;
  DM_Pointer    pointer;
  DM_Class      classid;
  DM_Event      event;
  DM_Attribute  attribute;
  DM_Type       type;
  DM_Enum       enumval;
  DM_Method     method;
  DM_Index      index;
  DM_ID         id;
} DM_ValueUnion;
```

### Meaning of the Elements

#### DM_UInt cardinal

This entry is only available for Version 2 of Dialog Manager due to compatibility reasons. If you want to address this entry, you have to define also the symbol DM_COMPAT_CARDINAL, besides the usual definitions on compiling the sources with the C compiler. As soon as this symbol has been defined, you can access values of the type enumeration (DT_enum), type (DT_type) and scope (DT_scope) by means of this entry.

#### DM_Scope scope

The data will be transferred in this element, if the type of object (instance, model or default) is to be queried, whereby value 1 corresponds to a default, value 2 to a model and value 3 to an instance.

#### DM_Boolean boolean

The Boolean attributes are filed in this element. The element can have the value TRUE or FALSE.

#### DM_String string

In this element the values of attributes with textual character are filed. This element then becomes a pointer to the corresponding string.

#### DM_Integer integer

In this element the values of the attributes with numeral character are filed.

#### DM_Pointer pointer

Pointers which have been defined by the application are transferred in this element. The contents of this pointer is not known to Dialog Manager, it merely passes on the corresponding pointer.

**DM_Class classid**

The class of an object is transferred in this element. It can adopt all values beginning with "DM_ Class" in the include file "IDMuser.h". You can find a complete table of the available classes in the “Object Reference”.

**DM_Event event**

In this element you can save events. These events are defined in the include file "IDMuser.h". For a detailed description, please refer to the manual “Rule Language”. You can find a complete table of the available events in the chapter “Events” of that manual.

**DM_Type type**

The data types are saved in this element. These data types are also defined in the include file "IDMuser.h". All data types begin with the prefix "DT_". You can find a complete table of the available data types in the “Object Reference”.

**DM_Enum enumval**

In this element all data types are transferred which are treated internally as enumeration types. These constants all begin with the prefix "DM_". You can find a complete table of the available enumeration constants in the “Object Reference”.

**DM_Index index**

In this element two values are transferred for the row and column in question, i.e. two numbers.

**DM_Method method**

In this element methods are transferred. You can find a complete list of the available methods in the “Method Reference”.

**DM_ID id**

In this element the object ID of a DM object is transferred.

### 3.3.3 Structure DM_Value

The structure is responsible for the actual data exchange between application and Dialog Manager. It contains the DM_ValueUnion for passing on the actual values. In addition it receives further information on the attribute which has been queried or set.

```
typedef struct {
  DM_Boolean     inherit : 1;
  DM_Boolean     changed : 1;
  DM_UInt1       type;
  DM_ValueUnion  value;
} DM_Value;
```

## Meaning of Elements

### DM_Boolean inherit

By means of the element *inherit* you can get information on whether the object itself knows the value or whether the object has inherited this value from its model or default. Is this element set to TRUE, after querying an attribute, the object has only inherited the value; is the element set to FALSE, the object itself has the value.

### DM_Boolean changed

The structure element *changed* is used mainly internally in the DM.

After having set an attribute value you can query with this element whether the attribute in question has really received a new value by the value allocation. In this case, *changed* is set to TRUE. If the attribute has already had this allocated value before, *changed* will be set to FALSE.

### DM_UInt1 type

If a value is queried by DM_Value, the DM sets the structure element *type* according to the queried attribute. If, however, a value is set in Dialog Manager from within the application, then the application has to file the correct type in this element. The allocation of the following structure DM_ValueUnion depends on the value saved here.

The following allocations are possible:

| Identifier | Possible Type | Meaning |
|---|---|---|
| DT_accel | DM_ID | accelerator reference |
| DT_attribute | attribute | attribute designation |
| DT_boolean | boolean | boolean value (true or false) |
| DT_class | classid | describes the classes of the various objects |
| DT_color | DM_ID | color reference |
| DT_cursor | DM_ID | cursor reference |
| DT_datatype | cardinal | indicates data types as values of variables |
| DT_enum | cardinal | necessary for the definition of icons and buttons for objects messagebox, tablefield and setup object |
| DT_event | cardinal | variables/attributes can assume various events as values. Events are specified in the event line of a rule. (Only one event can be specified as value.) |
| DT_font | DM_ID | font reference |
| DT_func | DM_ID | function reference |

| Identifier | Possible Type | Meaning |
|---|---|---|
| *DT_hash* | *pointer* | associative array |
| DT_index | Index | two-dimensional indices can be stored as one value (e.g. for tablefield -> attributes containing coordinates of the tablefield element) (values of the DT_index type are defined by [I, J]) |
| DT_instance | DM_ID | instance reference |
| DT_integer | integer | integer value |
| *DT_list* | *pointer* | value list |
| *DT_matrix* | *pointer* | two-dimensional array |
| DT_method | method | method |
| DT_object | DM_ID | object reference |
| DT_pointer | pointer | any pointer |
| *DT_refvec* | *pointer* | reference list of DM_ID values where the same ID can only occur once |
| DT_rule | DM_ID | rule reference |
| DT_scope | cardinal | defines whether the value is a default, a model or an object |
| DT_string | string | passes a string |
| DT_text | DM_ID | text reference |
| DT_tile | DM_ID | tile reference |
| DT_var | DM_ID | variable reference |
| *DT_vector* | *pointer* | array of the same value type |

Possible values for *DT_Scope* are:

1 default object

2 model

3 instance

### 3.3.4 Structure DM_VectorValue

This structure is responsible for querying and setting vectorial attributes efficiently. It is similar to the DM_Value structure. In the external structure the data type and the number of the values to be set and queried are indicated. In the interior structure the actual values are passed on.

```
typedef struct {
  DM_Type  type;
  DM_UInt  count;

  union {
#if defined(DM_COMPAT_CARDINAL)
  DM_Cardinal *  cardinalPtr;
#endif
  DM_Scope *      scopePtr;
  DM_Event *      eventPtr;
  DM_Type *       typePtr;
  DM_Enum *       enumPtr;
  DM_Boolean *    booleanPtr;
  DM_String *     stringPtr;
  DM_Integer *    integerPtr;
  DM_Class *      classidPtr;
  DM_ID *         idPtr;
  DM_Index *      indexPtr;
  DM_Attribute *  attributePtr;
  DM_Method *     methodPtr;
  DM_Pointer *    pointerPtr;
  DM_Value *      valuePtr;
  } vector;
} DM_VectorValue;
```

#### Meaning of the Elements

**DM_UInt1 type**

If a value is queried in DM via DM_VectorValue, DM sets the structure element *type* according to the queried attributes. If, however, a value in DM is set from within the application, the application has to file the correct type in this element. The allocation of the following structure "vector" depends on the value saved here.

**DM_UInt count**

In this element you can query the number of elements DM has saved in the substructure "vector". On setting attributes from within the application, the application must have filed the number of attributes to be set.

**DM_UInt *cardinalPtr**

This entry is only available for Version 2 of Dialog Manager due to compatibility reasons. If you want to address this entry, you have to define also the symbol DM_COMPAT_CARDINAL,

besides the usual definitions on compiling the sources with the C compiler. As soon as this symbol has been defined, you can access values of the type enumeration (DT_enum), type (DT_type) and scope (DT_scope) by means of this entry.

**DM_Scope *scopePtr**

The data is transferred in this element, if the type of object (instance, model or default) is to be queried. Value 1 corresponds to a default, value 2 to a model and value 3 to an instance.

**DM_Boolean *booleanPtr**

The boolean attributes are filed in this element, whereby the element can have the value TRUE or FALSE.

**DM_String *stringPtr**

In this element the values of attributes with textual character are filed. This element then becomes a pointer to the corresponding string.

**DM_Integer *integerPtr**

In this element the values of the attributes with numeral character are filed.

**DM_Pointer *pointerPtr**

Pointers which have been defined by the application are transferred in this element. The contents of this pointer is not known to Dialog Manager, it merely passes on the corresponding pointer.

**DM_Class *classidPtr**

The class of an object is transferred in this element. It can adopt all values beginning with "DM_ Class" in the include file "IDMuser.h". You can find a complete table of the available classes in the “Object Reference”.

**DM_Event *eventPtr**

In this element you can store events. These events are defined in the include file "IDMuser.h". For a detailed description, please refer to the manual “Rule Language”. You can find a complete table of the available events in the chapter “Events” of that manual.

**DM_Type *typePtr**

The data types are stored in this element. These data types are also defined in the include file "IDMuser.h". All data types begin with the prefix "DT_". You can find a complete table of the available types in the manual “Rule Language”.

**DM_Enum *enumPtr**

In this element all datatypes are transferred which are treated internally as enumeration types. These constants all begin with the prefix "DM_". You can find a complete list of the available enumeration constants in the “Object Reference”.

**DM_Method *methodPtr**

In this element methods are transferred. You can find a complete list of the available methods in the “Method Reference”.

**DM_Index *indexPtr**

In this element two values are transferred for the relevant row and column, i.e. two numbers.

**DM_ID *idPtr**

In this element the object ID of a DM object is transferred.

**DM_Value *valuePtr**

In this element the values of a DM_Value structure are passed on.

#### Note

If values are fetched from the dialog to the application by using the structure, DM allocates the needed memory space and later releases it again. If, however, values are set in the dialog using this structure from within the application, the application has to allocate the vectors in the suitable size and has to release memory.

### 3.3.5 Structure DM_MultiValue

This structure is responsible for setting and querying efficiently several attributes at several objects by one single call to Dialog Manager. This structure contains information at which object each filed attribute is to be set or queried.

```
typedef struct {
  DM_ID         object;
  DM_Attribute  attribute;
  DM_Value      index;
  DM_Value      data;
  DM_Options    options;
} DM_MultiValue;
```

#### Meaning of the Elements

**DM_ID object**

This element receives the object in which the attribute is to be queried or replaced. The value may be 0; in this case the identifier of the object indicated in the function call is used.

**DM_Attribute attribute**

This element contains the attribute to be queried or replaced.

**DM_Value index**

This element contains the index of the attribute to be queried or replaced in suitable form.

**DM_Value data**

This element contains the queried or new value of the attribute.

#### DM_Options options

This element contains the options which are valid for this attribute on setting or querying (DMF_...).

### 3.3.6 Structure DM_Content

This structure is used to set and query combined vectorial attributes for tablefields and listboxes. By means of this structure you can query or set by one access the selectability, the actual string and the relevant userdata for all relevant fields.

```
typedef struct {
  DM_String      string;
  DM_Boolean     active : 1;
  DM_Boolean     sensitive : 1;

  DM_UInt1       udtype;
  DM_ValueUnion  udvalue;
} DM_Content;
```

#### Meaning of Elements

#### DM_String string

In this element the contents actually to be displayed is indicated.

#### DM_Boolean active

This element indicates whether the corresponding entry is to be displayed pre-selected (true) or not (false).

#### DM_Boolean sensitive

This element indicates whether the corresponding entry is selectable by the user or not.

#### DM_UInt1 udtype

In this element the data type of the attribute "userdata" is indicated.

#### DM_Value udvalue

This element contains the actual value of the attribute "userdata".

## 3.4 Object Callback Structure DM_CallBackArgs

This structure is always passed on to callback functions as parameter.

```
typedef struct {
  DM_UInt4        evbits
  DM_ID            object;
  DM_ID            accel;
  DM_Int2            index;
```

```
  DM_UInt2        skeyno;
} DM_CallBackArgs;
```

### Meaning of Elements

#### DM_UInt4 evbits

In this element the eventmask of the triggering event is passed on. Since for every event exactly one bit is set in this mask, these events can occur in combined form as well.

#### DM_ID object

In this element the object to which this function is bound is passed on.

#### DM_ID accel

This element defines the DM identifier of the pressed key.

#### DM_UInt2 index

In this element the line or cell of the relevant object will be passed on, if the event is an object with an index (e.g. selection in a listbox or in a poptext).

#### DM_UInt2 skeyno

This element is no longer used and thus always = 0.

## 3.5 Object Reloading Structure DM_ContentArgs

This structure is always passed on to reloading functions as parameter.

```
typedef struct {
  DM_ID           object;
  DM_UInt2        reason;
  DM_UInt2        visfirst;
  DM_UInt2        vislast;
  DM_UInt2        loadfirst;
  DM_UInt2        loadlast;
  DM_UInt2        count;
  DM_UInt2        header;
} DM_ContentArgs;
```

### Meaning of Elements

#### DM_ID object

In this element the object to which this function is bound is passed on.

#### DM_UInt2 reason

In this element the cause of the call is indicated. Currently, only the value CFR_load is possible here. This value indicates that the contents is to be reloaded in the tablefield.

**DM_UInt2 visfirst**

**DM_UInt2 vislast**

In this elements the first and last visible row or column is indicated - depending on the attribute value of .direction. If the attribute .direction is defined by 1, reloading is processed row by row; consequently "visfirst" and "vislast" denote the first and last visible row. If the attribute .direction is defined by 2, reloading is processed column by column; consequently "visfirst" and "vislast" denote the first and last visible column.

**DM_UInt2 loadfirst**

**DM_UInt2 loadlast**

In these elements the first and last row or column to be loaded are indicated - depending on the attribute value .direction. If the attribute .direction is defined by 1, reloading is processed row by row; consequently "loadfirst" and "loadlast" denote the first and last row to be loaded. If the attribute is defined by 2, reloading is processed column by column; consequently "loadfirst" and "loadlast" denote the first and last column to be loaded.

**DM_UInt2 count**

In this element the number of rows = .rowcount (.direction = 1) or columns = .colcount (.direction = 2) defined at the tablefield is passed on.

**DM_UInt2 header**

In this element the number of headers defined at the tablefield is indicated.

## 3.6 Data Function Structure DM_DataArgs

This structure is always passed as a parameter to data functions.

```
typedef struct
{
  DM_ID         object;
  DM_Method     task;
  DM_Attribute  attribute;
  DM_Method     method;
  DM_String     identifier;
  DM_Integer    position;  /* reserverd for future use */

  int           argc;
  DM_Value     *argv;

  DM_Value      pad;       /* reserverd for future use */

  DM_Value      index;
  DM_Value      data;
  DM_Value      retval;
```

```
} DM_DataArgs;
```

### Meaning of elements

#### DM_ID object

In this element, the object ID of the Data Model is passed to the data function.

#### DM_Method task

This element specifies the reason for the call. Currently the following values are possible: *MT_get*, *MT_set* or *MT_call*. For *MT_get*, the data function should return the necessary data of a Model attribute. *MT_set* forwards changes from the View to the data function. *MT_call* is used to implement data actions, i.e. calls that lead to “manifold” manipulations of the data. It is reasonable to always report changes to data via **DM_DataChanged()**.

#### DM_Attribute attribute

For the call reasons *MT_get* and *MT_set*, this element contains the attribute on which the function shall be applied.

#### DM_Attribute method

For the call reason *MT_call*, this element contains the method that the data function shall execute. This is triggered by invoking a data action through the method `:calldata(<Method>, <Arg1>, ...<Arg15>)`.

#### DM_String identifier

This element contains the attribute identifier (from the *attribute* element) or the method identifier (from the *method* element) as a string to facilitate the implementation for user-defined attributes or to enable a module-independent comparison of attributes or methods.

#### int argc

In this element, for the call reason *MT_call* (data action invoked via **:calldata()**), the number of arguments is set.

#### DM_Value *argv

In this element, for the call reason *MT_call* (data action invoked via **:calldata()**), the argument list with the count *argc* is stored. Returning or modifying arguments is not intended here respectively will have no effect.

#### DM_Value index

This element contains the index value for the attribute access that the call reason *MT_get* or *MT_set* refers to. If the *index* element has the data type *DT_void*, it refers to the entire value. For indexed data values, the relationship kinds and also the processing of incomplete index values (e.g. *[0]*, *[?,0]* oder *[0,?]*) should be taken into account.

**DM_Value data**

For the call reason *MT_set*, this element contains the data value to be processed by the data function.

**DM_Value retval**

For the call reason *MT_get*, in this element the data value for the specified attribute and the given index has to be returned.

## 3.7 Toolkit Datastructure DM_ToolkitDataArgs

This structure can be passed as data parameter to the function DM_GetToolkitDataEx. It serves as a means of communication in both directions and allows the transfer and return of different value types. DM_TKAM_* defines in the *argmask* field of the structure linked with logical OR serve as marker for a set value.

The “argmask” field must be initialized before calling DM_GetToolkitDataEx.

The DM_Rectangle structure is an auxiliary structure for the coordinates and size of a rectangular “object shape”.

```
typedef struct {
       DM_Int4 x;
       DM_Int4 y;
       DM_UInt4 width;
       DM_UInt4 height;
       } DM_Rectangle;

/*
** T O O L K I T A R G U M E N T S
**
** Extend the DM_GetToolkitData() and DM_SetToolkitData() functions
** with additional calling informationen and output data.
*/
#define DM_TKAM_index (1 << 0)
#define DM_TKAM_data (1 << 1)
#define DM_TKAM_handle (1 << 2)
#define DM_TKAM_widget (1 << 3)
#define DM_TKAM_rectangle (1 << 4)
#define DM_TKAM_tile (1 << 5)
#define DM_TKAM_dpi (1 << 6)
#define DM_TKAM_scaledpi (1 << 7)
#define DM_TKAM_tile_req (1 << 8)

typedef struct {
       DM_UInt4 argmask;
       DM_Value index;
       DM_Value data;
```

```
#if defined(WSIWIN) || defined(WIN32)
	HANDLE handle;
#endif
# if defined(MOTIF) || defined(QT)
# ifdef _XtIntrinsic_h
	Widget widget;
# else
	DM_Pointer widget;
# endif
#endif
	DM_Rectangle rectangle;

	/* structure members for DM_TKAM_tile */
	struct {
	DM_UInt2 gfxtype; /* the specific DM_GFX picture type */
#if defined(WSIWIN) || defined(WIN32)
	DM_UInt2 datatype;
	union {
		struct {
			HANDLE data;
			HBITMAP mask;
			HPALETTE palette;
		};
		LPUNKNOWN iunk;
	};
#endif
#if defined(QT)
	DM_Pointer pixmap;
#endif
#if defined(MOTIF)
# ifdef _X11_XLIB_H_
XImage* ximage;
# else
	DM_Pointer ximage;
# endif
# ifdef X_H
	Pixmap pixmap;
	Pixmap trans_mask;
# else
DM_Pointer pixmap;
DM_Pointer trans_mask;
# endif
#endif
	DM_UInt dpi;
} tile;
```

```
/* structure members for DM_TKAM_dpi */
DM_UInt dpi;
/* structure members for DM_TKAM_scaledpi */
struct {
       DM_UInt dpi;
       DM_UInt factor; /* in percent */
       } scale;

       /* structure member for DM_TKAM_tile_reqtype */
       DM_UInt2 tile_req; /* the possible DM_GFX picture types (bitwise or'ed)
*/
} DM_ToolkitDataArgs;
```

### Meaning of elements

**DM_UInt4 argmask**

Input/Output: Must be initialized before the call. If fields are used as input, then the corresponding bits must be set. DM_GetToolkitDataEx fills in fields of this structure depending on the queried attribute and adds the corresponding bits to the ""“argmask”"". In the following list the corresponding bit is listed in brackets behind the field.

**DM_Value index**

Input: Index of a table cell

Bits: DM_TKAM_index

**DM_Value data**

Output: DM_ID and data type of an IDM object.

Bits: DM_TKAM_widget, DM_TKAM_widget

**DM_Rectangle rectangle**

Output: coordinates and size of a rectangular object shape or width and height of an image - depending on the argmask set.

Bits: DM_TKAM_rectangle, DM_TKAM_tile

**DM_UInt2 tile.gfxtype**

Output: Set to the GFX Type that fits best.

Bits: DM_TKAM_tile

**DM_UInt tile.dpi**

Output: The DPI value for which the images loaded by the IDM were designed.

Bits: DM_TKAM_tile

**DM_UInt dpi**

Output: The standard DPI value of the system (mostly 96). (MICROSOFT WINDOWS: see below)

Bits: DM_TKAM_dpi, DM_TKAM_tile

**int scale.dpi**

Output: The DPI value set by the system according to the set scaling. (MICROSOFT WINDOWS: see below)

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

**int scale.factor**

Output: system scale factor (MICROSOFT WINDOWS: see below)

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

**DM_UInt2 tile_req**

Input: Desired data type when querying a tile resource.

Bit: DM_TKAM_tile_reqtype

### 3.7.1 Specific structure elements for Microsoft Windows

The parameters specific to the MICROSOFT WINDOWS window system are defined as follows:

**HANDLE handle**

Input: Depending on the queried attribute, a Microsoft Windows handle can be set here as an additional input parameter.

Bits: DM_TKAM_handle

**DM_UInt2 tile.gfxtype**

Output: Is set to the GFX type that is returned. Possible values are:

- DM_GFX_BMP: GDI Bitmap Handle (HBITMAP) in "tile.data"
- DM_GFX_WMF: GDI Metafile Handle (HMETAFILE) in "tile.data"
- DM_GFX_EMF: GDI Enhanced Metafile Handle (HENHMETAFILE) in "tile.data"
- DM_GFX_ICO: GDI Icon Handle (HICON) in "tile.data"
- DM_GFX_D2D1BMP: Direct2D Bitmap (ID2D1Bitmap *) in "tile.iunk"
- DM_GFX_D2D1SVG: Direct2D SVG Document (ID2D1SvgDocument *) in "tile.iunk"
- DM_GFX_D2D1EMF: Direct2D Metafile (ID2D1GdiMetafile *) in "tile.iunk"

Bits: DM_TKAM_tile

**DM_UInt2 tile.datatype**

Output: Setting to “DMF_TIkData*”” type that is returned. Possible values are:

- DMF_TIkDataIsIcon: Microsoft Windows Icon Handle in "tile.data"
- DMF_TIkDataIsWMF: Microsoft Windows Metafile Handle in "tile.data"
- DMF_TIkDataIsEMF: Microsoft Windows Enhanced Metafile Handle in "tile.data"
- DMF_TIkDataIsD2D1Bmp: Microsoft Direct2D Bitmap (ID2D1Bitmap *) in "tile.iunk"
- DMF_TIkDataIsD2D1SVG: Microsoft Direct2D SVG Documnet (ID2D1SvgDocument *) in "tile.iunk"

- DMF_TlkDataIsD2D1EMF: Microsoft Direct2D Metafile (ID2D1GdiMetafile *) in "tile.iunk"
- sonst: Microsoft-Windows Bitmap Handle in "tile.data"

Bits: DM_TKAM_tile

### HANDLE tile.data

Output: Set to the data, the exact type depends on **tile.gfxtype** or **tile.datatype**:

- HICON: if *DM_GFX_ICO* resp. *DMF_TlkDataIsIcon*
- HBITMAP: if *DM_GFX_BMP* resp. *0*
- HMETAFILE: if *DM_GFX_WMF* resp. *DMF_TlkDataIsWMF*
- HENHMETAFILE: if *DM_GFX_EMF* resp. *DMF_TlkDataIsEMF*

Bits: DM_TKAM_tile

### HBITMAP tile.mask

Output: Set to the monochrome mask if one exists. This can only occur with *DM_GFX_BMP*.

Bits: DM_TKAM_tile

### HPALETTE tile.palette

A output: Set to the color palette if one is available.

Bits: DM_TKAM_tile

### LPUNKNOWN tile.iunk

Output: The MICROSOFT DIRECT2D interface pointer may be stored in this element.

Bits: DM_TKAM_tile

### DM_UInt dpi

Output: Set to the Microsoft Windows default DPI value. This is the value that applications that are DPI Unaware use. This value is defined by Microsoft Windows as a constant *USER_DEFAULT_SCREEN_DPI* (value: 96).

The IDM also uses this value for its pixel coordinates.

Bits: DM_TKAM_dpi, DM_TKAM_tile

### int scale.dpi

Output: Set to the current DPI value of the object. If the AT_DPI attribute is queried, the return value of DM_GetToolkitDataEx and this value are identical. This value is dynamic for IDM for Windows 11 if the application is DPI Aware. Otherwise, DPI Unaware or IDM for Windows 10, the value is fixed at 96.

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

### int scale.factor

Output: Integer percentage value calculated from **scale.dpi** and **dpi**.)

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

**DM_UInt2 tile_req**

Input: Used to obtain a specific data type when querying a tile resource.
The following values are available:

- DM_GFX_BMP: GDI Bitmap Handle (HBITMAP)
- DM_GFX_WMF: GDI Metafile Handle (HMETAFILE)
- DM_GFX_EMF: GDI 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 data types can be specified linked with “bitwise or”. The value DM_GFX_BMP is preselected as a fallback.

Bits: DM_TKAM_tile_req

### 3.7.2 Specific structure elements for Qt

The parameters specific to the QT window system are defined as follows:

**DM_Pointer widget**

Input: Depending on the queried attribute, a QWidget can be set here as an additional input parameter.

Bits: DM_TKAM_widget

**DM_Pointer tile.pixmap**

Output: Set to the QPixmap of the tile. Data type in **tile.gfxtype is always** *DM_GFX_QPIXMAP*.

Bits: DM_TKAM_tile

### 3.7.3 Specific structure elements for Motif

The parameters specific to the MOTIF window system are defined as follows:

**Widget/ DM_Pointer widget**

Input: Depending on the queried attribute, an XWidget can be set here as an additional input parameter.

Bits: DM_TKAM_widget

**XImage*/ DM_Pointer tile.ximage**

Output: complete image information as XImage, if **tile.gfxtype** = *DM_GFX_XIMAGE*

Bits: DM_TKAM_tile

**Pixmap/DM_Pointer tile.pixmap**

Output: image information as pixmap, if **tile.gfxtype** = *DM_GFX_PIXMAP*

Bits: DM_TKAM_tile

**Pixmap/DM_Pointer tile.trans_mask**

Output: the transparency clipmask associated with the image information in tile.pixmap if **tile.gfxtype** = *DM_GFX_PIXMAP*

Bits: DM_TKAM_tile

## 3.8 Structures and Definitions for the Binding of Functions

In the following chapters all structures defined by Dialog Manager and their meanings are described. These structures are responsible for the communication between the application and Dialog Manager.

### 3.8.1 Definitions DML_c, DML_pascal and DML_default

By means of these definitions the calling conventions of the application functions on PC-based systems (MS Windows) are described. These definitions depend on the used window system and on the compiler.

```
#ifdef ICC
# define DML_pascal
# define DML_c
#else
# define DML_pascal pascal
# define DML_c      cdecl
#endif

#ifdef DOS
# define DML_default DML_pascal
#else
# define DML_default DML_c
#endif
```

### 3.8.2 Defnition DM_ENTRY

By means of this definition the application function can be defined in such a way that the definition can be applied to all systems supported by Dialog Manager without changes. To do so, all functions called by the DM have to be of the type DM_ENTRY.

The definition itself depends on the platform and is as follows:

```
#  if defined(MSW)
#      define DM_ENTRY        far
#  endif

#  if defined(WIN32)
```

```
#      define DM_ENTRY
#  endif

#  if defined(MOTIF)
#      define DM_ENTRY        far
#  endif
```

### 3.8.3 Definition DM_CALLBACK

By means of this structure the callback functions can be defined so that the definition can be applied to all systems supported by Dialog Manager without changes. To do so, the callback function called by the DM have to be of the type DM_CALLBACK.

The definition itself depends on the platform and is as follows:

```
#  if defined(MSW)
#      define DM_CALLBACK    far
#  endif

#  if defined(WIN32)
#      define DM_CALLBACK
#  endif

#  if defined(MOTIF)
#      define DM_CALLBACK    far
#  endif
```

### 3.8.4 Definition DM_EXPORT

By means of these functions all functions included in the interface of Dialog Manager are defined.

The definition itself depends on the platform and is as follows:

```
#  if defined(MSW)
#      define DM_EXPORT        far
#  endif

#  if defined(WIN32)
#      define DM_EXPORT
#  endif

#  if defined(MOTIF)
#      define DM_EXPORT        far
#  endif
```

### 3.8.5 Definition DM_EntryFunc

This type definition is responsible for passing on different kinds of functions in one single function table to Dialog Manager.

```
typedef void (DM_ENTRY *DM_EntryFunc) __((void));
```

### 3.8.6 Structure DM_FuncMap

By means of this structure the addresses defined in the dialog and the functions realized in the application are introduced to Dialog Manager. Afterwards DM is able to call the application functions.

```
typedef struct {
  DM_String     symbol;
  DM_EntryFunc  address;
} DM_FuncMap;
```

#### Meaning of Elements

**DM_String symbol**

In this element the function name - as it has been described in the dialog script - is passed on to Dialog Manager.

**DM_EntryFunc address**

In this element the address of the real functions is passed on to Dialog Manager.

## 3.9 Structures for Canvas Functions

The structure introduced in the following chapters serves as parameter for all canvas functions. This structure as well as the relevant functions are window-system independent.

### 3.9.1 Structure DM_CanvasUserArgs for Microsoft Windows

For the window system Microsoft Windows the definition of the canvas structure is as follows:

```
typedef struct
{
  DM_ID    object;
  int      reason;
  HWND     hwnd;
  UINT     message;
  WPARAM   wParam;
  LPARAM   lParam;
  LRESULT  mresult;

  int      x;
  int      y;
```

```
int      width;
int      height;
HANDLE   hdc;

FPTR     userdata;
} DM_CanvasUserArgs;
```

### Meaning of the Elements

#### DM_ID object

This element defines the DM ID of the canvas object.

#### int reason

This element indicates the reason for calling the canvas function. Currently there are the following possibilities:

| Reason | Meaning |
|---|---|
| *CCR_expose* | There is a redraw event for the canvas. |
| *CCR_input* | The user has made an input either by using the mouse or the key-board. |
| *CCR_reschg* | A canvas resource has been changed. |
| *CCR_resize* | The size of the canvas has been changed |
| *CCR_start* | The canvas has been made visible on the screen. |
| *CCR_stop* | The canvas is no longer visible on the screen. |
| *CCR_winmsg* | A Microsoft Windows message has arrived. The return value of the canvas function is very important. TRUE means that the message has been processed; "mresult" is returned to Microsoft Windows. FALSE means that the message has not been processed; Dialog Manager executes the default action. |

#### HWND hwnd

HandleID of the Microsoft Windows window.

#### UINT message

#### WPARAM wParam

#### LPARAM lParam

#### LRESULT mresult

These are the original message parameters of Microsoft Windows. If no message is available, the element will have the value "message" 0. The elements *wParam*, *lParam* and *mresult* are only

valid, if the element "message" does not equal 0.

**int x**

This element defines the x-coordinate of the used space with regard to the canvas window.

**int y**

This element defines the y-coordinate of the used space with regard to the canvas window.

**int width**

This element defines the width of the used space with regard to the canvas window.

**int height**

This element defines the height of the used space with regard to the canvas window.

**HANDLE hdc**

This element is the **device-context-handle**, which is needed to draw in the canvas.

**FPTR userdata**

In this element the callback function can save any data referring to the user. The save values are also passed on to the canvas function on each call. For further information please refer to the “Attribute Reference”, attribute "AT_CanvasData".

### Note

The allocation of this structure always depends on the current task. The elements "object", "reason" and "hwnd" are always allocated; the allocation of the other structure elements can be taken from the following table:

| Task | Allocated Element | Meaning |
|---|---|---|
| CCR_expose | x, y, width, height | dimension of the update rectangular |
| | hdc | device-context handle to draw in the canvas |
| CCR_input | x, y | these fields are allocated, if the original event has been a "ButtonPress" event |
| CCR_reschange | x, y | coordinate of the upper left corner of the used space with regard to the actual canvas window |
| | width, height | dimension of the canvas' useable area |
| CCR_resize | x, y | coordinate of the upper left corner of the used space with regard to the actual canvas window |
| | width, height | dimension of the canvas' useable area |

| Task | Allocated Element | Meaning |
|---|---|---|
| CCR_start | x, y | coordinate of the upper left corner of the used space with regard to the actual canvas window |
| | width, height | dimension of the canvas' useable area |
| CCR_winmsg | message, wParam, lParam, mresult | These elements pass on the original windows messages. The possible values and the meaning of these elements can be taken from the Microsoft Windows manual. |

### 3.9.2 Structure DM_CanvasUserArgs for Motif

For the window system Motif the definition of the canvas structure is as follows:

```
typedef struct {
  DM_ID    object;
  int      reason;

#ifdef _XLIB_H_
  Xevent   *xevent;
  Window   window;
#else
  FPTR     xevent;
  long     window;
#endif

#ifdef _XtIntrinsic_h
  Widget   widget;
#else
  FPTR      widget;
#endif

  DM_Int2   x;
  DM_Int2   y;
  DM_UInt2  width;
  DM_UInt2  height;

  FPTR      userdata;
} DM_CanvasUserArgs;
```

### Meaning of Elements

#### DM_ID object

This element defines the DM ID of the object which has triggered the function process, i.e. the canvas to which the called function belongs.

#### int reason

In this element the DM informs the application about the reason why the canvas function has been called.

Currently there are the following possibilities:

| Reason | Meaning |
|---|---|
| *CCR_expose* | There is a redraw event for the canvas. |
| *CCR_focus_in* | The canvas has received the input focus. |
| *CCR_focus_out* | The canvas has lost the focus. |
| *CCR_input* | The user has made an input either by using the mouse or the keyboard. |
| *CCR_reschg* | A canvas resource has been changed. |
| *CCR_resize* | The size of the canvas has been changed |
| *CCR_start* | The canvas has been made visible on the screen. |
| *CCR_stop* | The canvas is no longer visible on the screen. |
| *CCR_xevent* | There is an arbitrary, a different X event for the canvas. |

#### Xevent *xevent

In this parameter the original X event is passed on to the application. If no X event is available, this parameter is a *NULL* pointer. These events are explained in the corresponding X manuals.

#### Window window

In this element the affiliated X-Windows window is passed on to the application. For further information please refer to the relevant X-Windows manuals.

#### Widget widget

In this element the Xt-Widget information is passed on to the object.

#### DM_Int2 x

In this element the x coordinate of the used space with regard to the canvas window is passed on.

#### DM_Int2 y

In this element the y coordinate of the used space with regard to the canvas window is passed on.

### DM_UInt2 width

In this element the width of the used space with regard to the canvas window is passed on.

### DM_UInt2 height

In this element the height of the used space with regard to the canvas window is passed on.

### FPTR userdata

In this element the callback function can save any data referring to the user. The save values are also passed on to the canvas function on each call. For further information please refer to the “Attribute Reference”, attribute "AT_CanvasData".

#### Note

The allocation of this structure always depends on the current task. The elements "object" and "reason" are always allocated, the elements "window", "widget" and "xevent" are, if available, always allocated; the allocation of the other structure elements can be taken from the following table:

| Task | Allocated Element | Meaning |
|---|---|---|
| *CCR_expose* | x, y, width, height | dimension of the update rectangular |
| *CCR_input* | x, y | these fields are allocated, if the original event was a "ButtonPress" event |
| *CCR_reschange* | x, y | coordinate of the upper left corner of the used space with regard to the actual canvas window |
| | width, height | dimension of the canvas' useable area |
| *CCR_resize* | x, y | coordinate of the upper left corner of the used space with regard to the canvas window, if the values are already available, otherwise 0 |
| | width, height | dimension of the used space in the canvas, if the values are already available, otherwise 0 |
| *CCR_start* | x, y | coordinate of the upper left corner of the used space with regard to the actual canvas window |
| | width, height | dimension of the canvas' useable area |

## 3.10 Structures for Input Handler Functions

By means of so-called input handlers, the application is able to react to events which have not been provided by the Dialog Manager. This reaction results from the input handler functions which have been installed in the Dialog Manager by the application and which are called by the DM in due course. The structures depend on the window system.

### 3.10.1 Structure DM_InputHandlerArgs for Microsoft Windows

For the window system Microsoft Windows the definition of the input-handler structure is as follows:

```
typedef struct
{
  HWND        hwnd;
  UINT        message;
  WPARAM      wParam;
  LPARAM      lParam;
  LRESULT     mresult;
  DM_Pointer  userdata;
} DM_InputHandlerArgs;
```

#### Meaning of Elements

**HWND hwnd**

In this element the window handle is passed on to the object which the message has been sent to.

**UINT message**

In this element the message which has been sent to the object is passed on. For information on the possible allocations, please refer to the manuals of Microsoft Windows.

**WPARAM wParam**

In this element information about the message is passed on in the same state as it has been received by the window system.

**LPARAM lParam**

In this element information about the message is passed on in the same state as it has been received by the window system.

**LRESULT mresult**

In this element the return value of the input-handler function is passed on. This element will only be evaluated on a special calling mode.

**DM_Pointer userdata**

In this element the pointer which has been indicated on installing the input handler is passed on. The meaning of the element is application-specific, i.e. the application can file any needed information.

### 3.10.2 Structure DM_InputHandlerArgs for Motif

For this system no structure has been defined, since no window-specific information has to be passed on to the application.

## 3.11 Structures and Definitions for the Formatting of Input

For the formatting of entries in edittexts, a number of structures have been defined which will be introduced in the following chapters. These structures directly or indirectly are all parameters of a format function defined in the application.

### 3.11.1 Definitions for the Formatting

The different tasks of the format function are available via the definition in the include file **IDMuser.h**. The individual tasks denote the following:

| Task Definition | Task |
|---|---|
| FMTK_parseformat | The indicated format string is to be parsed and the corresponding structures are to be established. |
| FMTK_cleanformat | The allocated structures belonging to the format are to be released. |
| FMTK_create | The format function is informed that the format is used for a valid object. If it is necessary, the structures belonging to the format are allocated and initialized. |
| FMTK_destroy | The object belonging to format is destroyed. The format-specific structures can thus be released again. |
| FMTK_setcontent | The format function is informed that a new contents has been set in the object via the program. The format function has to check the contents against the format and to correct the corresponding format information. |
| FMTK_setselection | The format function is to set the new positions of the cursor and of the beginning of the selection area with regard to the contents string. |
| FMTK_setmaxchars | The format function is informed that the attribute *.maxchars* is set for the object. The function has to carry out the relevant actions. |
| FMTK_formatcontent | The format function is informed that the contents is to be formatted so that the contents in the window system can be set. |
| FMTK_enter | This task is called if the input field belonging to this format has lost the input focus. The format function can then carry out the necessary actions on contacting the input field. |

| Task Definition | Task |
|---|---|
| FMTK_leave | This task is called if the input field belonging to this format has lost the input focus. The format function can carry out the necessary actions on leaving the input field. |
| FMTK_modify | The task is called if the user has made an input in the text. The function has to process the entry and the corresponding action in the displayed text. |
| FMTK_keynavigate | The task is called if the user has moved the cursor in the text. The function has to process the navigation and has to carry out the corresponding action in the displayed text. |
| FMTK_setcursorabs | The task is called if the user has positioned the cursor absolutely in the displayed text. The function then has to carry out the corresponding actions on the internal structures. |

### 3.11.2 Structure DM_FmtContent

By means of this structure the actual contents is saved. This structure also has to be updated on the corresponding calls to the format function.

```
typedef struct {
  DM_String        string;
  DM_UInt2         size;
  DM_UInt2         length;
  DM_UInt2         curpos;
  DM_UInt2         selpos;
  DM_UInt2         maxchars;
} DM_FmtContent
```

#### Meaning of Elements

**DM_String string**

In this element the actual contents is stored.

**DM_UInt2 size**

In this element the size of the memory allocated in the element *string* is stored.

**DM_UInt2 length**

In this element the actual length of the contents string is stored.

**DM_UInt2 curpos**

In this element the cursor position is stored in the contents string.

**DM_UInt2 selpos**

In this element the selection position is stored in the contents string.

**DM_UInt2 maxchars**

In this element the maximal length of the contents string is stored.

### 3.11.3 Structure DM_FmtRequest

By means of this structure the data is exchanged between Dialog Mananger and the format function. The union contained in the structure is allocated according to the task.

```
typedef struct {
  DM_UInt1 task;

  union {
  struct { /* FMTK_parseformat */
  DM_UInt1 codepage;
  DM_String formatstr;
  } parseformat;

  struct { /* FMTK_setcontent */
  DM_UInt1 codepage;
  DM_String string;
  } setcontent;

    struct { /* FMTK_setselection */
      DM_UInt2 contcurpos;
      DM_UInt2 contselpos;
    } setselection;

  struct { /* FMTK_setmaxchars */
  DM_UInt2 maxchars;
  } setmaxchars;

  struct { /* FMTK_modify */
  DM_String string;
  DM_UInt2 strlength;
  DM_UInt2 dpycurpos;
  DM_UInt2 dpyselpos;
    } modify;

  struct { /* FMTK_keynavigate */
  DM_Int2 yoffset;
  DM_Int2 xoffset;
  DM_Boolean movecur;
  DM_Boolean movesel;
```

```
    } keynavigate;

  struct { /* FMTK_setcursorabs */
  DM_UInt2 dpycurpos;
  DM_UInt2 dpyselpos;
  } setcursorabs;

  } targs;

  DM_ID object;

} DM_FmtRequest;
```

### Meaning of Elements

#### DM_UInt1 task

In this element the task to be carried out by the format function is stored. The value stored here controls the allocation of the following union.

#### Structure parseformat for Task FMTK_parseformat

##### DM_UInt1 codepage

In this element the codepage in which the formatted string is to be output is passed on. The necessary constants are defined in the include file "IDMuser.h" and they all begin with the prefix "CP_".

##### DM_String formatstr

In this element the actual format string is passed on.

#### Structure setselection for Task FMTK_setcontent

##### DM_UInt1 codepage

In this element the codepage in which the formatted string is to be output is passed on. The necessary constants are defined in the include file "IDMuser.h" and they all begin with the prefix "CP_".

##### DM_String string

In this element the new contents of the object to be formatted is passed on.

#### Structure setselection for Task FMTK_setselection

##### DM_UInt2 contcurpos

In this element the actual cursor position with regard to the contents is passed on.

##### DM_UInt2 contselpos

In this element the actual selection position with regard to the actual contents is passed on.

### Structure setmaxchars for Task FMTK_setmaxchars

#### DM_UInt2 maxchars

In this element the new maximum number of characters is passed on.

### Structure modify for Task FMTK_modify

#### DM_String string

In this element the string without terminating null byte is passed on. The string is to be inserted at the actual cursor position in the present contents.

#### DM_UInt2 strlength

In this element the length of the string to be inserted is passed on.

#### DM_UInt2 dpycurpos

In this element the actual cursor position in the displayed string is passed on. At this position the indicated string is to be inserted.

#### DM_UInt2 dpyselpos

In this element the actual selection position in the displayed string is passed on. If this value equals the value in dpycurpos, the indicated string will be inserted at the corresponding position; if this value does not equal the value in dpycurpos, the string enclosed by dpycurpos and dpyselpos will be replaced by the indicated string.

### Structure keynavigate for Task FMTK_keynavigate

#### DM_Int2 yoffset

This element contains information on the navigation in y-direction. A positive value here means a downward navigation, a negative value denotes an upward navigation.

#### DM_Int2 xoffset

This element contains information on the navigation in x-direction. A positive value here means a navigation to the right, a negative value denotes a navigation to the left.

#### DM_Boolean movecur

This element contains information on whether the cursor position is to be changed by a cursor movement (TRUE) or not (FALSE).

#### DM_Boolean movesel

This element contains information on whether the selection position is to be changed by a cursor movement (TRUE) or not (FALSE).

### Structure setcursorabs bei Task FMTK_setcursorabs

#### DM_UInt2 dpycurpos

In this element the new absolute cursor position with regard to the indicated string is passed on.

#### DM_UInt2 dpyselpos

In this element the new absolute selection position with regard to the indicated string is passed on.

### 3.11.4 Structure DM_FmtFormat

This structure contains data which is only accessible to the Dialog Manager-internal format function and which represents internal information about the current format independently of the contents string.

```
typedef struct {
  DM_UInt2        maxchars;
  char            secretChar;
} DM_FmtFormat;
```

#### Meaning of Elements

#### DM_UInt2 maxchars

In this element the maximum length of the contents string defined by the format string is indicated.

#### char secretChar

In this element the character used during the covered formatting is saved.

### 3.11.5 Structure DM_FmtDisplay

This structure is needed to display formatted strings. It informs the window system on the text to be displayed. This structure contains also information on the position in the string at which the cursor is to be set and on the position at which the selection mark is to be set.

```
typedef struct {
  DM_Boolean         overwritemode : 1;
  DM_UInt1           codepage;
  DM_UInt2           curpos;
  DM_UInt2           selpos;
  DM_UInt2           length;
  DM_UInt2           size;
  DM_String          string;
} DM_FmtDisplay;
```

#### Meaning of Elements

#### DM_Boolean overwritemode

This element contains information on whether the window system is in the overwrite or insert mode.

#### DM_UInt1 codepage

This element contains information on the codepage of the display string.

**DM_UInt2 curpos**

In this element the current cursor position in the display string is stored.

**DM_UInt2 selpos**

In this element the current selection position in the display string is stored.

**DM_UInt2 length**

In this element the length of the current display string is stored.

**DM_UInt2 size**

In this element the length of the allocated memory of the element "string" is stored.

**DM_String string**

In this element the current display string is stored.

## 3.12 Mapping of DM Data Types

The data types used in the dialog script are mapped according to a fixed pattern.

If these data types are declared only as **input parameters** of a function, they must be declared in the C function according to the following table.

| DM datatypes | C datatypes |
|---|---|
| anyvalue | DM_Value * |
| attribute | DM_Attribute |
| boolean | DM_Boolean |
| class | DM_Class |
| enum | DM_Enum |
| event | DM_Event |
| integer | DM_Int4 |
| method | DM_Method |
| object | DM_ID |
| pointer | DM_Pointer |
| scope | DM_Scope |
| string | DM_String |
| type | DM_Type |

But if they are also or only declared as **output parameters** of a function, they usually have to be declared as pointers on these data structures in the C function.

| DM datatypes | C dataypes |
|---|---|
| anyvalue | DM_Value * |
| attribute | DM_Attribute * |
| boolean | DM_Boolean * |
| class | DM_Class * |
| enum | DM_Enum * |
| event | DM_Event * |
| integer | DM_Int4 * |
| method | DM_Method * |
| object | DM_ID * |
| pointer | DM_Pointer * |
| scope | DM_Scope * |
| string | DM_String * |
| type | DM_Type * |

### Example

*Dialog Script*

```
boolean function1(boolean, string, object);

void function2(boolean input, integer output);

string function3(string input output,
                 integer output,
                 object input output);
```

*C Declaration*

```
DM_Boolean DML_default DM_ENTRY function1
(
  DM_Boolean par1,
  DM_String par2,
  DM_ID par3,
)
```

```
void DML_default DM_ENTRY function2
(
  DM_Boolean par1,
  DM_Int4 *par2
)

DM_String DML_default DM_ENTRY function3
(
  DM_String* par1,
  DM_Int4 *par2,
  DM_ID *par3
)
```

On declaring function parameters you have to consider the following:

If a parameter is declared only as output parameter, and if it is not assigned useful values by DM, the parameter should not be accessed in the C function by reading. If it is declared for input as well as for output, DM assigns the corresponding value to it.

On using the output parameters several elements (e.g. object attributes) can be changed in one function call.

# 4 Collections and String Handling

Collections can be used as arguments or return values of C functions or for Datamodel callbacks. The data type *index* (*DT_index*) is also allowed as return type for C functions.

In principle, **DM_Value** structures, which may or should contain collections, must only be manipulated and queried via special DM functions, since the IDM allocates memory and manages strings for this purpose.

### See also

**DM_ValueInit()**, **DM_ValueChange()**, **DM_ValueGet()**, **DM_ValueCount()**, **DM_ValueIndex()** and **DM_ValueReturn()**

To facilitate the handling of the **DM_Value** structure and strings (*DM_String)* in C functions, the following concept has been introduced.

The C interface of the IDM can distinguish value references (*DM_Value** and *DM_String**) according to their use or initialize them for a special use:

1. The value reference is an argument of the function call and yet unmanaged.
2. The value reference is a static or global value and is managed by the IDM.
3. The value reference is a local value managed by the IDM. Returning it to the IDM is allowed. After the function call is completed, it will be released.
4. The value reference refers to a value managed by the IDM that is invalid, for example because it has been manipulated improperly by the user.
5. The value reference refers to an unmanaged value (sort like up to now).

This concept is supported by the functions of the kind **DM_Value*()** and **DM_*Return()**. It allows the manipulation of values (e.g. changing a string, adding values to a list or hash), the access of elements or indexes of value lists as well as the return of value lists or strings without having to care explicitly about the administration and allocation.

### Rules for Usage

For values and **DM_Value** structures of the kind 1–3 the following must be observed:

- Any direct manipulation of the values in the structure must be avoided. Only read access to scalar values is permitted.
- Direct copying of this structure into another memory area or to another address as well as using such a copy is prohibited! For this no correct administration and access can be guaranteed!
- Only copying of a managed value into a yet unmanaged return parameter is allowed.
- Once a local or static usage has been set, it cannot be changed anymore.

» When returning local values, strings or indexes, the functions **DM_ValueReturn()**, **DM_StringReturn()** and **DM_IndexReturn()** should be used so the IDM can ensure that the values of local variables are returned correctly. For static values and strings this is not necessary.

As values managed by the IDM may also contain strings, it should be noted that these strings are held and returned in the application code page. Changing this code page by a **DM_Control()** call does not change the code page of the managed strings. The application code page should therefore be set correctly as early as possible.

The functions **DM_ValueInit()**, **DM_ValueChange()**, **DM_ValueGet()**, **DM_ValueCount()**, **DM_ValueIndex()**, **DM_ValueReturn()**, **DM_StringInit()**, **DM_StringChange()**, **DM_StringReturn()** and **DM_IndexReturn()** can only be used for local and distributed (DDM) C functions. Their use in format functions is not permitted.

A faulty manipulation of a value managed by the IDM or a status change by the user will be reported in the C interface, when recognized, by the error codes *"DME_InvalidContext"* or *"DME_BadContext"*. Usually input and output parameters should not point to the same value reference either, which in turn is indicated by a *"DME_InvalidArg"* error.

Managed local value references are cleaned up after the corresponding function call and, where necessary, substructures and strings are released.

For particularly time-critical functions, the use of managed values and strings should probably be avoided or the time behavior should be checked, since their use will in any case mean an increased administration overhead in the IDM.

# 5 Writing Programs

## 5.1 Main Program

This chapter describes how the functions defined in DM are realized in C and how they have to be linked to DM.

In principle each C program having the name "AppMain" in the non-distributed version has the name "AppInit" in the distributed version.

This main program is called by the Dialog Manager and can then carry out the necessary initialization steps.

### 5.1.1 Local C Programs

In the local version of Dialog Manager the program "AppMain" in principle is the virtual program. It has to carry out the following steps:

- initialization of Dialog Manager (DM_Initialize)
- loading a dialog (DM_LoadDialog)
- initialization of application
- passing on the function addresses (DM_BindFunctions)
- reading in the user profile (DM_LoadProfile)
- starting the dialog (DM_StartDialog)
- starting the processing (DM_EventLoop)

After having carried out these steps successfully, you can work with the program.

### 5.1.2 Distributed C Programs

In the distributed version of Dialog Manager the C main program "AppInit" has to carry out the following steps:

- passing on function addresses (DM_BindFunctions)
- initialization of application.

## 5.2 Normal Application Functions

Since the type of parameter and the type of the function call is decisive for the successful execution of the program especially on PC-based systems, you should absolutely make use of the DM ability to generate prototypes from the functions defined in the dialog. If you install the generated prototypes in the form of an include file in your program, the C compiler will check whether the realization of the C

function agrees with the declaration. If this is not the case, the C compiler outputs either an error or a warning on compiling, according to the severity of the error. Fatal errors can so be traced very early. Besides the prototypes this option also creates a C file containing a function to link functions so that the function table does not have to be edited separately.

The option to generate the prototype and the C file by means of the simulation program is:

```
+writefuncmap
```

The simulator has the following command line:

```
idm +writefuncmap <basis name of file> <name of dialog file>
```

The name of the generated C function is formed according to the following pattern:

```
BindFunctions_<name of dialog, of module or of application>
```

### Example

In this example the necessary function prototypes are to be created from a dialog definition having the following functions.

Input command:

```
idm +writefuncmap locallst list.dlg
```

The dialog script list.dlg is as follows:

```
dialog DEMO
{
   .xraster   8;
   .yraster   20;
}
function void InitTestAppl(object);
```

The created C file is as follows:

```
#include "IDMuser.h"
#include "locallst.h"

#define FuncCount_DEMO (sizeof(FuncMap_DEMO) /
    sizeof(FuncMap_DEMO[0]))

static DM_FuncMap FuncMap_DEMO[] =
{
  { "InitTestAppl", (DM_EntryFunc) InitTestAppl },
}

DM_Boolean DML_default BindFunctions_DEMO __3(
    (DM_ID, dialogID),
    (DM_ID, moduleID),
    (DM_Options, options))
{
    return (DM_BindCallBacks (FuncMap_DEMO,FuncCount_DEMO,
```

```
        dialogID,options))
}
```

The created H file is as follows:

```
#ifndef __INCLUDED_locallst_h_
#define __INCLUDED_locallst_h_
#ifdef __cplusplus
    extern "C" {
#endif
void DML_default DM_ENTRY InitTestAppl __((DM_ID));
/*
**  prototype of the function to bind the module functions
**  if the 'moduleID' is not known call it with (DM_ID)0
*/
DM_Boolean DML_default BindFunctions_DEMO __((DM_ID dialogID, DM_ID moduleID,
DM_Options options));

#ifdef __cplusplus
    }
#endif

#endif /* __INCLUDED_locallst_h_ */
```

If only a file with function prototypes is to be created, you can also use the option

```
+writeproto <name of include file>.
```

The simulator has the following command line:

```
idm +writeproto <name of include file> <name of dialog file>
```

## 5.3 Functions with Records as Parameters

To provide functions for dialogs including records with an application, the C modules created by DM have to be compiled and linked. These functions may never occur in a function table which is defined in the application, since these functions are linked via the created module. This way of creating is necessary for each dialog module containing definitions for functions. These modules are created by calling the simulaton with the option +writetrampolin:

```
idm +writetrampolin <outfile> <dialogscript>
```

This statement creates the modules needed for the call of functions from a dialog script. The indicated dialog script may be a module here. According to the type of functions using such records the relevant header files (C and/or COBOL) are created.

On implementing these functions the created header file has to be linked to the relevant module. This is done by the following statement:

```
include
```

If such functions and records are used in an application, these records have to be initialized by calling the function

**RecInit<dialog name>**

or

**RecMInit<module name>**

### Note

These functions have not to be linked until DM_BindCallbacks has been called!

If such structures are used in Dialog Manager applications, i.e. if the distributed DM is used, the application for which the files are to be created has to be indicated additionally.

The name of the initialization function is then formed from the name of the application.

Consequently

**RecMInit<ApplicationName>**

has to be called

```
idm +writetrampolin <contfile> +application <ApplicationName>
  <dialogscript>
```

### Example

From the dialog

```
dialog RecTest
{
}
function void WriteAddr(record Address input);
function integer ReadAddr(record Address output, integer);

record Address
{
  string[25] Name shadows W1.Lname.E.content;
  string[15] FirstName shadows W1.Lfirstname.E.content;
  string[25] City shadows W1.Lcity.E.content;
  string[30] Street shadows W1.Lstreet.E.content;
}
```

the following include file is created:

```
#ifndef __INCLUDED_xxx_h_
#define __INCLUDED_xxx_h_

#ifdef __cplusplus
    extern "C" {
#endif
```

```
typedef struct {
    DM_String Name;
    DM_String FirstName;
    DM_String City;
    DM_String Street;
} RecAddress;

void DML_default DM_ENTRY WriteAddr __((RecAddress *));
DM_Integer DML_default DM_ENTRY ReadAddr __((RecAddress *, DM_Integer));
DM_Boolean RecInitRecTest __((DM_ID dialog));

#ifdef __cplusplus
    }
#endif

#endif /* __INCLUDED_xxx_h_ */
```

### 5.3.1 Dynamic Binding of Record Functions

#### Availability

Since IDM version A.06.01.d

"Dynamic binding" refers to a binding type for application functions that does not require explicitly setting the function pointer of the application functions by means of the functions DM_BindFunctions or DM_BindCallBacks. This includes, for example, connecting functions from dynamic libraries (transport *"dynlib"*) or calling COBOL functions by their names through the "BindThruLoader Functionality". However, since application functions with **record parameters** require an additional stub function for marshalling the record structure, no structure has been transferred so far when these record functions were called.

As of IDM version A.06.01.d, the structure of the record parameters for dynamically bound record functions is transferred without a stub function. Such record functions can be used without generating a C or COBOL code. It is however useful to generate the function prototypes as well as the structure definition via **+writeheader <basefilename>**.

#### Example

```
dialog D
record RecAdr {
  string[80] Name := "";
  string[120] Street := "";
  string[40] City := "";
  integer ZIP := 0;
}
```

```
application Appl {
  .transport "dynlib";
  .exec "libadr.so";
  .active true;
  function boolean Search(string Pattern, record RecAdr output);
}

application ApplCobol {
  .local true;
  .active true;
  function cobol boolean New(record RecAdr) alias "NEWADR";
}

on dialog start {
  if not Appl.active orelse not ApplCobol.active then
    print "Application(s) not properly connected!";
  elseif not Search("Joe", RecAdr) then
    print "Joe not found, adding him";
    RecAdr.Name := "Joe";
    RecAdr.Street := "765 East Walnut Lane";
    RecAdr.City := "Oak Park";
    RecAdr.ZIP := 48237;
    New(RecAdr);
  else
    print "Joe found, lives in: "+RecAdr.City;
  endif
  exit(();
}
```

The prototypes can now simply be generated with **+writeheader** instead of **+writeproto**, **+write-funcmap** and **+/-writetrampolin**:

```
pidm adr.dlg +application Appl +writeheader adr       // generates adr.h
pidm adr.dlg +application ApplCobol +writeheader adr  // generates adr.cpy
```

The actual implementation in C and COBOL can now find the function prototypes or record definitions for “RecRecAdr” in **adr.h** (C header file) and **adr.cpy** (COBOL copy file).

### 5.3.2 Note for Using DM Functions

Within a C function that uses a ***record*** as parameter, this DM function must always be called with the option *DMF_DontFreeLastStrings*, since otherwise strings contained in the ***record*** are also released; these strings would no longer be available.

#### See Also

Chapter “Handling of String Parameters” in manual “C Interface - Functions” and the respective function descriptions.

## 5.4 Object Callback Functions

Functions which are declared as object callback functions in the dialog descriptions, have to be declared in C as follows:

```
DM_Boolean DML_default DM_CALLBACK Functionname
(
  DM_CallBackArgs *data
)
```

if the function in the dialog script is defined as

```
function callback Functionname() for Events;
```

or as

```
DM_Boolean DML_c DM_CALLBACK Functionname
(
  DM_CallBackArgs *data
)
```

if the function in the dialog script is defined as

```
function c callback Functionname() for Events;
```

The parameter of this function is already fixed by DM and cannot be changed. This function is always called, if an event indicated with the function has occurred at the corresponding object.

The return value TRUE here means that rules are to be processed normally after having called this function; the return value FALSE prevents such a rule processing.

### Example

The definition of a callback function in the dialog script is as follows:

```
function callback CheckFilename() for deselect, modified;
```

The object is assigned at the relevant object:

```
child edittext File
{
  .xleft 14;
  .ytop 0;
  .width 20;
  .function CheckFilename;
  .content "list.dlg";
}
```

The following function is to check whether the user input represents a valid file name.

```
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 can not 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);

        /*
         * Everything is ok. So let the rule be
         * processed normally
         */
        return(TRUE);
    }

    /* To many errors don't continue the rule processing */
    return (FALSE);
}
```

## 5.5 Reloading Functions

Functions which are declared as reloading functions in the dialog description have to be declared in C as follows:

```
DM_Boolean DML_default DM_CALLBACK Functionname
(
```

```
  DM_ContentArgs *data
)
```

if the function is defined in the dialog script as follows

```
function contentfunc Functionname();
```

or as

```
DM_Boolean DML_c DM_CALLBACK Functionname
(
  DM_ContentArgs *data
)
```

if the function is defined in the dialog script as

```
function c contentfunc Functionname ();
```

The parameter of this function is already fixed by Dialog Manager and cannot be changed. This function is called whenever the tablefield, which has defined this function as reloading function, has been scrolled to an area in which contents in DM is no longer available and therefore the contents has to be reloaded again by the application.

### Example

In the dialog a reloading function is defined as follows:

```
!! Fills up the tablefield on scrolling
!! if it becomes necessary
function contentfunc CONTENT();
```

This function is then linked to a tablefield

```
child tablefield T1
{
  .visible true;
  .xauto 0;
  .xleft 1;
  .xright 1;
  .yauto 0;
  .ytop 0;
  .ybottom 1;
  .posraster true;
  .sizeraster true;
  .fieldfocusable false;
  .fieldshadow false;
  .contentfunc CONTENT;
  .edittext null;
  .selection[sel_row] true;
  .selection[sel_header] false;
  .selection[sel_single] false;
  .colcount 3;
  .rowcount 30;
```

```
  .rowheadshadow true;
  .rowheader 1;
  .colfirst 1;
  .rowfirst 2;
  .colwidth[0] 12;
  .rowheight[0] 1;
  .rowlinewidth[0] 1;
  .rowlinewidth[1] 3;
  .sensitive[1,1] false;
  .content[1,1] "Name";
  .sensitive[1,2] false;
  .content[1,2] "Vorname";
  .sensitive[1,3] false;
  .content[1,3] "Wohnort";
  .xraster 10;
  .yraster 16;
}
```

The realization of the function C then is as follows:

```
/*
** Function to fill up the tablefield dynamically
** on scrolling, if rows are reached
** which have not been filled yet.
*/

void DML_default DM_CALLBACK CONTENT __1(
(DM_ContentArgs *, args))
{
  int i;
  int spos;
  int vpos;
  int length;
  FILE *f ;

  char ** strvec;
  char * buffer;
  char * start;
  char * end;

  DM_VectorValue vector;
  DM_Value startIdx;
  DM_Value endIdx;

  /*
  ** Opening file from which the tablefield content
  ** is to be read.
```

```
*/
if (!(f = fopen("bsp.dat","r")))
{
  DM_TraceMessage("Cannot open In-File", DMF_LogFile);
  return;
}
/*
** Allocation of memory to be able to store
** the rows already read before assigning it to the object.
** temporarily To do so, the DM_functions are used.
*/
strvec  = (char **) DM_Malloc ( (CONT_ROWS * COLUMNS)
      * sizeof (char*) );
buffer  = (char *) DM_Malloc (STR_LEN * sizeof (char) );

/*
** first tablefield row to be filled
*/
startIdx.type = DT_index;
startIdx.value.index.first = args->loadfirst - 1;
startIdx.value.index.second = 1;
/*
** first tablefield row to be filled
*/
endIdx.type = DT_index;
endIdx.value.index.first = startIdx.value.index.first +
      (ushort) CONT_ROWS - 1;
endIdx.value.index.second = COLUMNS;

vector.type = DT_string;
vector.vector.stringPtr = strvec;

spos = 0;
vpos = 0;

/*
** Reading in the file to the buffer.
** In doing so, after every blank a new element
** in the tablefield is started.
*/
while ( !feof(f) && (spos++ < CONT_ROWS) )
{
  if (fgets(buffer ,STR_LEN-1, f))
  {
    i = 0;
    end = buffer;
```

```
    while (*end && isspace(*end))
      end++;

    while (*end && (i++ < COLUMNS))
    {
      start = end;
      length = 1;

      while (*end && !isspace(*end))
      {
        length++;
        end++;
      }
      if (*end)
        *end++ = '\0';
      strvec[vpos]=(char *) DM_Malloc( (lenght+1) *
          sizeof(char));
      strcpy(strvec[vpos],start);
      vpos++;
      while(*end && isspace(*end))
        end++;
    }

      while (i++ < COLUMNS)
    {
      strvec[vpos] = (char *) DM_Malloc(sizeof(char));
      strvec[vpos++]="\0";
    }
  }
}
/*
** Setting the number of valid entries in the vector.
*/
vector.count = (ushort) vpos;

/*
** Assigning the established vector to the tablefield.
*/
DM_SetVectorValue(args->object, AT_field,
      &startIdx, &endIdx, &vector, 0);

/*
** Releasing memory which has been allocated
** in this function.
*/
```

```
  while (--vpos >= 0 )
    DM_Free(strvec[vpos]);
  DM_Free(strvec);
  DM_Free(buffer);

}
```

## 5.6 Data Functions

Functions that are declared as data functions in the dialog description must be declared in C as follows:

```
DM_Boolean DML_default DML_CALLBACK <function name> (DM_DataArgs *args);
```

if the function is defined as

```
{ export | reexport } function datafunc <function name> ();
```

in the dialog script

or as

```
DM_Boolean DML_c DML_CALLBACK <function name> (DM_DataArgs *args);
```

if the function is defined as

```
{ export | reexport } function c datafunc <function name> ();
```

in the dialog script.

The parameter of this function is predefined by the ISA Dialog Manager and cannot be changed. It represents a Data Model (Model component) that provides the presentation objects (View component) with data values or stores and manages them.

This function is called when a synchronization between View and Model components is required. This may either happen automatically, according to the control options in the *.dataoptions[]*attribute of the involved components. The call can also be triggered by explicit invocation of the **:collect()**, **:propagate()**, **:apply()** and **:represent()** methods. For each Model attribute there is a separate call.

### 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);
}
```

**See also**

Function DM_DataChanged

## 5.7 Canvas Functions

Functions which are declared as canvas functions in the dialog descriptions have to be programmed in C independently of the window system. The underlying window system is usually accessed to carry out the desired actions.

```
DM_Boolean DML_default DM_CALLBACK Functionname
(
  DM_CanvasUserArgs *canvasargs
)
```

if the function is defined in the dialog script as

```
function canvasfunc Functionname();
```

or as

```
DM_Boolean DML_c DM_CALLBACK Functionname
(
  DM_ContentArgs *data
)
```

if the function is defined in the dialog script as

```
function c canvasfunc Functionname();
```

### Example

In the dialog a canvas function is defined as follows:

```
function canvasfunc CallbackCanvas();
```

The assignment to the canvas is as follows:

```
child canvas Canvas
{
  .borderwidth 1;
  .xauto 0;
  .xleft 1;
  .xright 1;
  .yauto 0;
  .ytop 1;
  .ybottom 5;
.canvasfunc CallbackCanvas;
}
```

A realization for the window system Motif then is as follows:

```
DM_Boolean DML_default DM_CALLBACK CallbackCanvas (data)
DM_CanvasUserArgs *data;
{
  switch (data->reason)
  {
    case CCR_expose:
    if (data->xevent)
    {
      DM_TraceMessage("CCR_expose width %d height %d",
          DMF_LogFile | DMF_InhibitTag | DMF_Printf,
          data->xevent->xexpose.width,
          data->xevent->xexpose.height);

      if (data->xevent->xexpose.count == 0)
        DoExpose(data->widget, x0, y0);
    }
    break;

    case CCR_input:
    switch (data->xevent->type)
    {
      case ButtonPress:
      switch (data->xevent->xbutton.button)
      {
        case Button1:
        x0 = data->xevent->xbutton.x;
```

```
          y0 = data->xevent->xbutton.y;
          DoExpose(data->widget, x0, y0);
          break;
        }
        break;
      }
      break;

      case CCR_start:
      DM_TraceMessage("CCR_start ** ", DMF_LogFile |
          DMF_InhibitTag);
      /* fallthrough */

      case CCR_resize:
      {
        Arg args[2];

        XtSetArg(args[ 0 ], XmNwidth, NULL);
        XtSetArg(args[ 1 ], XmNheight, NULL);
        XtGetValues(data->widget, args, 2);

        width = args[0].value;
        height = args[1].value;
        DM_TraceMessage("Resize height %d width %d event %ld",
            DMF_LogFile | DMF_InhibitTag | DMF_Printf,
            height, width, (long) data->xevent);
      }
      break;

      case CCR_stop:
      break;
    }
    /*
    ** On Motif the return value of the canvas function is ignored.
    ** On Windows the message processing is stopped if the
    ** return value is TRUE.
    ** So the default return value of the canvas function is FALSE.
    */
    return FALSE;
  }
```

## 5.8 Input-Handler Functions

Functions which are installed as input-handler functions have to be programmed in C. Their task is to process additional input sources and to incorporate the entries in an appropriate way in the dialog.

The sources of such external messages differ according to the underlying window system. With these functions you can wait on additional file descriptors for messages coming from other processes under Motif on Unix. On Microsoft Windows you can wait in this input handler for special messages which, in turn, may come from other processes. This function depends on the window system and the operating system, therefore the definitions which apply to this function differ according to the relevant system:

### Definition for MOTIF

```
DM_Boolean DML_default DM_CALLBACK Functionname
(
  FPTR userdata,
  int fdescr,
  DM_UInt iomode
)
```

### Definition for WINDOWS

```
DM_Boolean DML_default DM_CALLBACK Functionname
(
  DM_InputHandlerArgs far *pInpArgs,
  DM_UInt msg,
  DM_UInt iomode
)
```

### Example

*Task*

How can an application intercept when the user shuts down Windows, e.g. to display a messagebox asking the user whether he wants to save his changes or not?

*Solution*

The ISA Dialog Manager sends no event in this case, but a DM_InputHandler for the message WM_ENDSESSION may be installed. In this input handler **no** DM function must be called except for **DM_QueueExtEvent**. But this is rather useless as the application gets no chance to continue its work.

*Dialog File*

```
dialog WinExit {}

messagebox M
{
  .title "Message";
  .text "Do you want to save ?";
  .icon icon_warning;
  .button [1] button_yes;
  .button [2] button_no;
  .button [3] nobutton;
}
```

```
window
{
  .title "Test Exit Windows";
  .width 200;
  .height 100;

  on close
  {
    if (querybox (M) = button_yes) then
      !! Save data
    endif
    exit ();
  }
}
```

*C Source*

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

DM_Boolean DML_default DM_CALLBACK ExitHandler
  __((DM_InputHandlerArgs far *pInpArgs, DM_UInt msg,
     DM_UInt iomode));

int DML_c DM_CALLBACK AppMain __2(
  (int, argc),
  (char **, argv))
{
  DM_ID dialogID;
  HWND hwnd;

  DM_Initialize(&argc, argv, 0);

  dialogID = DM_LoadDialog ("winexit.dlg", 0);
  if (dialogID == (DM_ID) 0)
  {
    /* TODO: error handling */
    return (1);
  }

  DM_StartDialog(dialogID, 0);

  /*
   * Install an input handler for WM_ENDSESSION.
   * The input handler asks the user whether to save or not
   * and does the required action.
   * Note: this can only be done in C, no ISA DM calls are possible,
   * since Windows exits after return from the message.
```

```
   */

  hwnd = DM_InputHandler (ExitHandler, (DM_Pointer) 0,
    (DM_UInt) WM_ENDSESSION, DMF_ModeMsgManage,
    DMF_RegisterHandler,(DM_Options) 0);
  if (hwnd == (HWND) 0)
  {
    /* TODO: error handling */
    return (2);
  }

  DM_EventLoop(0);
  return (0);
}

DM_Boolean DML_default DM_CALLBACK ExitHandler __3(
  (DM_InputHandlerArgs far *, pInpArgs),
  (DM_UInt, msg),
  (DM_UInt, iomode))
{
  /*
   * Check whether really exit and ask user whether to
   * save or not.
   * ATTENTION: no DM function call allowed, therefore you
   * need to have all required informations in C data.
   * Application is terminated after return, therefore
   * do all actions here.
   */

  pInpArgs->mresult = (LRESULT) 0;

  if (pInpArgs->wParam != 0)
  {
    /*
     * Windows wants to exit, ask user.
     * Note: this is the same messagebox as in the dialog,
     * but we cannot call any DM function, therefore
     * do it the Windows way.
     */

    if (MessageBox ((HWND) 0, "Do you want to save ?",
      "Message", MB_ICONEXCLAMATION | MB_YESNO)
      == IDYES)
    {
      /*
       * Save, but no DM function call.
       */
```

```
    }

    /*
     * Uninstall handler after work is done.
     */
    return (FALSE);
  }
  else
  {
    /*
     * No exit required, let handler be installed.
     */
    return (TRUE);
  }

  /*
   * Note: the application is stopped immediately after
   * returning from this function.
   */
}
```

## 5.9 Format Functions

The format functions have to be defined in the application as follows:

```
DM_Boolean DML_default DM_CALLBACK Functionname
(
  DM_FmtRequest far * req,
  DM_FmtFormat far * fmt,
  FPTR *fmtPriv,
  DM_FmtContent far * cont,
  FPTR *contPriv,
  DM_FmtDisplay far * dpy
)
```

The format function is defined in the dialog script as follows:

```
function formatfunc Functionname;
```

### Parameters

#### DM_FmtRequest far * req

This parameter passes on the actual task which is taken on by the format function. The task in the element "task" is transferred to this structure.

| Task | Description |
|---|---|
| FMTK_parseformat | In this task the indicated format string is to be parsed and structures belonging to the format are to be initialized. |
| FMTK_cleanformat | This task is to release the filed private format information because the format is no longer needed. |
| FMTK_create | This task is to carry out the initialization of the private data for the actual contents string so that the contents string can be accessed after this task. |
| FMTK_destroy | This task is to release the private data filed in *contPriv*. |
| FMTK_setcontent | A new contents string is set. This contents string has to be transferred to *cont*. If required, the data has to be updated in *contPriv.* If a *display structure* is available the new contents string has to be formatted and the result is filed in the *display structure*. |
| FMTK_setselection | New positions of the cursor and of the beginning of a selection area with regard to the contents string are set. These values have to be transferred to *cont*. If required, the data in *contPriv* have to be updated. If *dpy* is available, the corresponding positions with regard to the formatted representation string have to be updated. |
| FMTK_setmaxchars | A new maximum length of the contents string is set. This has to be transferred to *cont*. If required, the contents string and the data in have to be updated in *contPriv.* If *dpy* is available, the new contents string has to be formatted and the result has to be filed in *dpy*. |
| FMTK_formatcontent | If the display structure is available, the new contents string has to be formatted and the result has to be filed in this structure. |
| FMTK_enter | This task will be called, if the editable text using this format receives the focus. The task does not have to carry out special tasks. |
| FMTK_leave | This task will be called, if the editable text using this format loses the focus. The task does not have to carry out special tasks. |
| FMTK_modify | This task is called, if the formatted display string is to be modified in the display structure. The desired modification has to be carried out so that the new formatting results in the modified display string. The display string has also to be modified accordingly. |

| Task | Description |
|---|---|
| FMTK_keynavigate | This task is called, if the positions in the representation string are to be changed relatively to the previous positions. The relevant positions in the contents string in "cont" are to be calculated so that they correspond to the new positions in the display string. The new positions in the display string have to be converted in the new positions in the contents string and have then to be filed in the display structure. |
| FMTK_setcursorabs | This task is called, if the positions in the display string are set to a new value. The relevant positions in the contents string are to be calculated so that they correspond to the new positions in the display string. The new positions in the display string have to be converted and be filed in the display structure. |

**DM_FmtFormat far * fmt**

In this parameter a pointer on format-specific information is passed on.

**FPTR *fmtPriv**

In this parameter the format function can file format-specific, private data. This data is only known to the application and is thus not considered by Dialog Manager.

**DM_FmtContent far * cont**

In this parameter a pointer is transferred to a structure in which the actual contents of the field is to be saved. The relevant elements always have to be updated.

**FPTR *contPriv**

In this parameter contents-specific, private data can be filed by the format function. The contents of this data is only known to the application and is thus not considered by the Dialog Manager.

**DM_FmtDisplay far * dpy**

In this parameter a pointer is transferred to the structure in which the actual display information is saved. If required, this structure always has to be updated in order for the affiliated edittext in the window system to be displayed correctly.

### Example

```
/* function for special handling of edittext user input */
function formatfunc My_Formatter ();

/* format resource using a format function */
format MyFormat "NN.NN.NN" My_Formatter;
```

The function is assigned to an input field in the following way:

```
/* here the date can be entered                          */
/* the single entered keys are checked by the formatfunc */
```

```
child edittext E1
{
  .width 10;
  .xleft 10;
  .ytop 1;
  .format MyFormat;
}
```

This function is realized in C as follows:

```
DM_Boolean DML_default 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)
  {
    /*
    ** implement a format for inserting a date
    ** like 'dd.mm.yy'
    ** there should be some more code for correct handling
    ** (especially when 'delete' or 'backspace' is pressed)
    **
    ** if the input is less than max
    ** the input is ok and can be processed
    ** by the DefaultFormatter
    */
    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:
    /*
    ** if no special handling is neccessary
    ** use standard formatter
    */
    retval = DM_FmtDefaultProc(req, fmt,
        (DM_FmtFormatDef **) (FPTR) fmtPriv,
        cont, (DM_FmtContentDef **) (FPTR) contPriv, dpy);
    break;
    }
  return (retval);
}
```

### See Also

Resource format

Chapter “Format Function” in manual “Rule Language”

# 6 Attributes and Definitions

In order to access the DM from the application, a number of symbolic names are available in IDMuser.h. In the following, these names are explained.

## 6.1 Attribute Definitions and their Data Types

For information on the attributes which are permitted for the individual object types, please refer to the “Object Reference”. In principle, however, they are derived from the names used in the Rule Language by typing “AT_” before the names.

**Example**

Rule Language *.visible* => C Interface *AT_visible*

## 6.2 Definitions for the Attribute Data Types

For information on the data types which are permitted for the individual attributes, please refer to the “Attribute Reference”. In principle, however, the names are derived from the names used in the Rule Language by typing “DT_” before the names.

**Example**

Rule Language *string* => C Interface *DT_string*

## 6.3 Access to User-Defined Attributes

Since the assignment of attribute names is dynamical to the internal attribute coding, the determination of the attribute coding is possible only during runtime. The attribute coding is only valid within a dialog, i.e. identical user-defined attributes have different internal identifications in different dialogs.

In order to be able to determine independently the attribute name known in the program, you have to do the following:

**Example**

Dialog script:

```
pushbutton P1
{
    integer Position[10];
}
```

In C:

```
DM_Attribute attr = AT_undefined;
DM_Value data;
```

```
DM_Value index;

index.type = DT_string;
index.value.string = ".Position";
    /* attribute name (compare example above)*/

if (DM_GetValueIndex(dialog, AT_label, &index, &data, 0)
    && (data.type == DT_attribute))
{
    attr = data.value.attribute;
}
```

This pattern can be used for every user-defined attribute.

Now you can access the attribute (.Position), e.g. by means of the function DM_GetValue:

```
DM_GetValue (P1, attr, 5, ...)
```

This call reads the fifth value from the user-defined attribute .Position.

## 6.4 Class Definition

The following class definitions are contained in the include file "IDMuser.h":

| Class Identifier | Meaning |
|---|---|
| DM_ClassAccel | Accelerator |
| DM_ClassApplication | Application |
| DM_ClassCanvas | Canvas |
| DM_ClassCheck | Checkbox |
| DM_ClassColor | Color |
| DM_ClassCursor | Cursor |
| DM_ClassDialog | Dialog |
| DM_ClassEdittext | Editable Text |
| DM_ClassGroupbox | Groupbox |
| DM_ClassListbox | Listbox |
| DM_ClassFont | Font |
| DM_ClassFunc | Function |
| DM_ClassImage | Image |

| Class Identifier | Meaning |
|---|---|
| DM_ClassImport | Import |
| DM_ClassMenubox | Menubox |
| DM_ClassMenuitem | Menuitem |
| DM_ClassMenusep | Menuseparator |
| DM_ClassMessagebox | Messagebox |
| DM_ClassModule | Module |
| DM_ClassNotebook | Notebook |
| DM_ClassNotepage | Notepage |
| DM_ClassPoptext | Poptext |
| DM_ClassPush | Pushbutton |
| DM_ClassRadio | Radiobutton |
| DM_ClassRect | Rectangle |
| DM_ClassRule | Rule |
| DM_ClassScroll | Scrollbar |
| DM_ClassSetup | Setup |
| DM_ClassStatext | Statictext |
| DM_ClassTablefield | Tablefield |
| DM_ClassText | Text |
| DM_ClassTile | Tile |
| DM_ClassVar | Variable |
| DM_ClassWindow | Window |

# 7 Compiling and Linking DM Programs

In this chapter you will learn how to compile and link programs developed by DM.

## 7.1 Include Files

All source files being related to DM in some way have to contain the include file **IDMuser.h**provided by DM. Depending on where this file has been installed, you can integrate it in the following way:

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

This method can be used if the file has been installed in the search path of the C compiler. This search path can be extended by the instruction -I<directory>.

```
# include "path/IDMuser.h"
```

Using this method you can integrate this file if it has not been installed in the search path of the C compiler. This method is less flexible and should therefore be avoided.

## 7.2 Compiler Flags

To guarantee that DM works properly and that the application uses the correct definition in the include file **IDMuser.h**, various definitions have to be set when the compiler is called. These definitions contain the operating system, the version of the operating system, the name of the compiler used and the type of the toolkit used.

For example, to compile a file on an HP 9000-800 on Motif with the operating system version 10.0, the necessary symbols are as follows:

```
cc -DHPUX -DHP9800 -DVERMAJOR=10 -DVERMINOR=0 -DMOTIF
```

Alternatively, **before IDMuser.h** is integrated, another file containing these definitions can be included.

```
# define HPUX
# define HP9800
# define VERMAJOR  10
# define VERMINOR  0
# define MOTIF
```

## 7.3 Special C-Compiler

Since there are different C compilers on different systems the file IDMuser.h contains compiler-dependent definitions.

**Example**

```
-DMSC
```

is the symbol to be defined if the Microsoft C compiler is used.

If the C compiler used understands function prototypes, the function definitions of **IDMuser.h** should be used for the DM functions.

## 7.4 Pascal Calling Convention for Microsoft Windows

If the Microsoft C compiler is being used to compile and link C programs, there are two possibilities to call the functions:

- "Pascal-Calling-Convention"
- "C-Calling-Convention"

The normal mode for Microsoft C compiler is C Calling Convention, the normal mode for Microsoft Windows is Pascal Calling Convention.

The difference between these two modes is

- the sequence of pushing the arguments onto the stack, and
- the type of stack restoring (calling or called function).

Since both Microsoft Windows and DM work internally with Pascal Calling Convention, all functions with no specified calling convention are called by the Pascal Calling Convention.

The calling convention can be set with one of the following possibilities:

### Explicit Declaration

The specified calling convention is used.

*Example*

| | |
|---|---|
| `function C integer foo (integer);` | C-Calling-Convention |
| `function pascal integer foo (integer);` | Pascal-Calling-Convention |

### Implicit Declaration

DM assumes internally the Pascal Calling Convention.

*Example*

| | |
|---|---|
| `function integer foo (integer);` | Pascal-Calling-Convention |

### Note

Do not declare the functions by yourself! You can have DM generate function prototypes with the command given below. The file "protofile" should be linked in each module (see example) in which functions are called by the DM and **FunctionMap** is defined.

### Example

```
idm +writeproto <protofile> <dialogscript>
```

(Please note that "idm" for MS Windows has to be started on MS Windows!)

### Note

If undefinable symbols occur during linking, the calling conventions might be responsible.

## 7.5 Overview

The specifications of the correct options and libraries have to be taken from the file **MakeDefs**.

If you want to link your program, you have to use one of the libraries specified below or a toolkit-specific DM library.

If your application is to run on Motif, you have to link the library **libIDM.a** to your program.

| Toolkit | Necessary Dialog Manager Libraries |
|---|---|
| MOTIF | -lIDM or libIDM.a and -lIDMinit.a |
| MSW | dm.lib and dminit.lib |

There are two libraries for each system:

- a library which is used during the program development:
  - **"libIDM.a"** (for Motif)
  - **"dm.lib"**or **dmdll.lib** (for MS Windows)
- a runtime library used for distributing a program:
  - **"libIDMrt.a"** (for Motif)
  - **"dmrt.lib"** or **dmrtdll.lib** (for MS Windows)

The difference between the two libraries is that in the development library you can make several security checks, and that it reads and processes ASCII files.

Only few checks are executed in the runtime library, and there is no possibility to read or process an ASCII file. This is why the runtime library consumes considerably less storage space.

**For programs that will be distributed you have to use the runtime library!**

## 7.6 Compiling on PCs

If you are working on a PC you have to specify different command-line options depending on the used operation system, window system and compiler when you want to compile.

In the directory "demos" in the makefile you can receive the options which are currently valid.

These options are compulsory for all functions to be called by Dialog Manager.

# Index

## D

## E



## F



## G



## H



## I

## L



## M



## N



## O



## P