ISA Dialog Manager

C-Schnittstelle - Grundlagen

A.06.03.b

In diesem Handbuch wird die prinzipielle Struktur der Schnittstelle des ISA Dialog Managers für in C entwickelte Anwendungen dargestellt. Das Handbuch beschreibt die Datentypen sowie das Übersetzen und Linken der Anwendungen.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

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

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

Inhalt

1 Einführung

Dieses Handbuch beschreibt die Anwendungsprogramm-Schnittstelle (gewöhnlich als "API-Application Interface" bezeichnet), die vom Dialog Manager (DM) für in C geschriebene Anwendungsprogramme angeboten wird. Dabei werden in diesem Handbuch die Funktionen beschrieben, die vom Anwendungsentwickler zu schreiben sind. Dazu werden die notwendigen Datentypen und Strukturen vorgestellt.

Die Zielgruppe für dieses Handbuch sind Programmierende, die sowohl mit der DM-Entwicklungsumgebung als auch mit dem eingesetzten C-Compiler vertraut sind. Voraussetzung sind außerdem gute Kenntnisse des verwendeten Betriebssystems und der Werkzeuge, die dieses Betriebssystem bietet.

Bitte beachten Sie insbesondere das Kapitel "Aufruf von Unterprogrammen". Sie sollten die Regeln, die die Datentypabbildung beschreiben, vollständig verstanden haben.

2 Prinzipielle Arbeitsweise

In diesem Kapitel wird die prinzipielle Arbeitsweise des C-Interfaces vorgestellt. Dabei soll die Philosophie deutlich gemacht werden, in der mit Hilfe des Dialog Managers geschriebene Programme realisiert werden können.

Bei der Realisierung von Anwendungsprogrammen sollte man immer das Schichtenmodell nach Seeheim im Kopf behalten, das eine strenge Trennung der verschiedenen Softwareschichten vorschreibt. Bei Benutzung des Dialog Managers zur Programmierung von Benutzeroberflächen können dabei die Präsentations- und die Dialogschicht fast vollständig in der Regelsprache realisiert werden.

Die C-Funktionen sollten im Idealfall keinerlei Wissen über die Oberfläche haben sondern immer alle benötigte Informationen vom Dialog in Form von Parametern übergeben bekommen und dann keinerlei Aufrufe an den Dialog Manager durchführen; sie sollten also keine "DM_"-Funktionen enthalten. Wenn Funktionen auf diese Art und Weise realisiert werden, ist die eigentliche Anwendungslogik und -verarbeitung in den C-Funktionen enthalten, die unabhängig von der verwendeten Oberfläche sind. Lediglich umfangreiche Zugriffe auf Objekte mit Listencharakter (Tablefield, Listbox und Poptext) sollten aus Gründen der Effizienz in C mit speziellen Funktionen des Dialog Managers realisiert werden. Diese Funktionen ermöglichen den Transport einer großen Anzahl von Daten mit einem Aufruf, wo sonst mehrere Einzelaufrufe nötig wären.

Diese Vorgehensweise ist auf jeden Fall beim Einsatz des verteilten Dialog Managers notwendig, da Funktionsaufrufe über das Netzwerk vergleichsweise teuer und daher ineffizient sind.

Eine Ausnahme hierzu stellt nur das eigentliche Hauptprogramm der Anwendung dar, da hier Zugriffe auf den Dialog Manager gemacht werden und Wissen über die Oberfläche notwendig ist.

Um das C-Pogramm passend zu der entwickelten Oberfläche schreiben zu können, müssen zunächst einige Datenstrukturen im C-Interface geklärt und die Abbildung der im Dialog Manager vorhandenen Datentypen auf die in C vorhandenen Datentypen definiert werden. Anhand eines Beispiels, das über die nachfolgenden Kapitel verteilt ist, soll die prinzipielle Vorgehensweise gezeigt werden.

2.1 Dialogdefinitionen

Im Dialog Manager Editor soll ein Dialogsystem gebaut werden, das zunächst dem Anwender eine tabellarische Übersicht über Namen und Wohnort anbietet, die er dann in einem Unterfenster modifizieren kann. Diese Übersicht besteht aus den ersten Zeilen einer Datei. Da diese Datei sehr viele Zeilen enthalten kann, sollen diese immer nur teilweise geladen und angezeigt werden. Um das zu realisieren, wird zunächst eine Funktion benötigt, die die Initialisierung der Tabelle übernimmt, und eine Nachladefunktion, die bei Bedarf neue Daten in das Tabellenelement lädt.

Bei einem Doppelklick auf das Tabellenelement soll ein modales Detailfenster geöffnet werden, in dem die Daten geändert werden können. Je nach Aktion werden dabei Funktionen aufgerufen, die über Records als Parameter diese Daten an C-Funktionen zur weiteren Verarbeitung weitergeben.

2.1.1 Übersichtsfenster

Das Übersichtsfenster hat folgendes Aussehen:

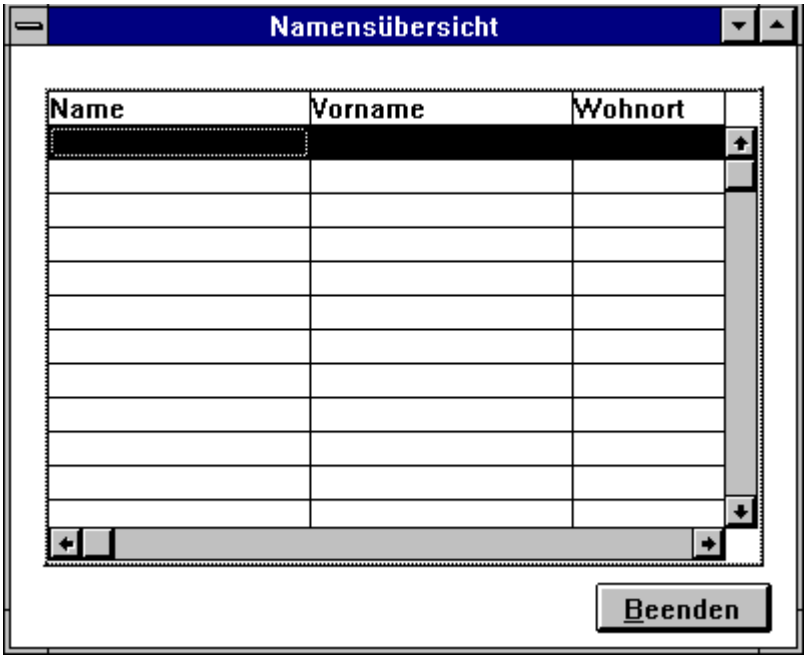

Abbildung 1: *Fenster „Namensübersicht“*

Im Dialogskript ist dieses Fenster wie folgt definiert:

```
window WnUebersicht
{
  .visible false;
  .active  false;
  .title   "Namens\374bersicht";
  .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] "Vorname";
    .content[1,3] "Wohnort";
    .xraster 10;
    .yraster 16;
  }
  child pushbutton PENDE
  {
    .xleft 36;
    .width 11;
    .yauto -1;
    .text  "&Beenden";
  }
}
```

2.1.2 Detailfenster

Die vollständigen zu einem Namen gehörenden Daten werden in einem separaten, als Dialogbox definierten Unterfenster dargestellt, in dem sie vom Anwender verändert werden können.

Das Fenster, in dem Daten verändert werden können, sieht wie folgt aus:

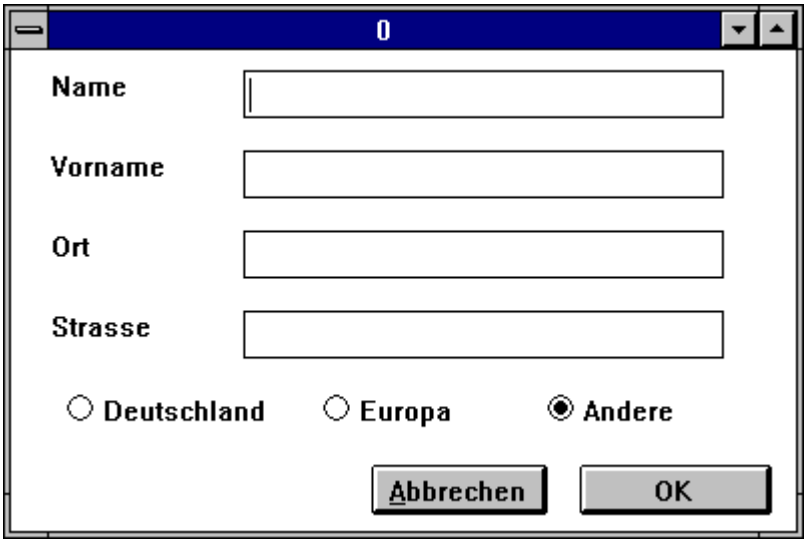

Abbildung 2: *Detailfenster*

Die zugehörige Definition im Skript sieht wie folgt aus:

```
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 "Vorname";
    .E.content "";
  }
  child Eintrag Lcity
  {
    .ytop 4;
    .S.text "Ort";
```

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

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

Zur einfacheren Generierung dieses Fensters wurde ein Modell eingeführt, das aus einer Groupbox mit einem statischen und editierbaren Text als Kind besteht. Instanzen dieses Modells dienen als Kinder des Detailfensters.

Die entsprechende Definition des Groupbox-Modells im Dialogskript sieht wie folgt aus:

```
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

Wenn alle Einträge aus der Datei gelesen wurden, soll eine ***messagebox*** den Benutzer darauf hinweisen.

Abbildung 3: messagebox

```
messagebox Mb
{
  .text  "Keine Eintr\344ge\nmehr in der Datei";
  .title "Achtung";
  .icon  icon_error;
  .button[1] button_ok;
  .button[2] nobutton;
  .button[3] nobutton;
}
```

2.2 Funktionsdefinitionen im Dialogskript

Für die Kommunikation mit dem C-Programm wurden mehrere Funktionen definiert:

FILLTAB()

Mit Hilfe dieser Funktion soll das Tablefield initial teilweise gefüllt werden.

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

CONTENT()

Diese Funktion ist die Nachladefunktion, die die Daten in das Tablefield nachladen soll.

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

FILECLOSE()

Diese Funktion schließt die benutzte Datei, wenn der Dialog beendet wird.

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

GETADDR()

Diese Funktion soll die zu einer Zeile gehörenden vollständigen Daten laden und an den Dialog übergeben.

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

PUTADDR()

Diese Funktion soll die vom Dialog geänderten Daten übernehmen und abspeichern.

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

2.3 Definition von Records

Die Funktionen GETADDR und PUTADDR erhalten jeweils einen Record als Parameter, dessen Elemente als Verweise (Links) auf die entsprechenden Elemente im Fenster definiert sind.

```
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 Regeln

Um die Funktionen aufzurufen, wurden mehrere Regeln definiert.

2.4.1 Regeln für den Programmstart

Beim Programmstart wird das Tablefield teilweise gefüllt und das zugehörige Fenster sichtbar gemacht.

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

2.4.2 Regeln für das Programmende

Durch Selektion des Ende-Pushbuttons wird das Programm beendet. Das Programm wird ebenfalls beendet, wenn das Hauptfenster über den Schließen-Eintrag im Systemmenü geschlossen wird.

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

2.4.3 Regeln für die Radiobuttons

Wenn einer der Radiobuttons selektiert wird, wird dessen Wert am zugehörigen Fenster gemerkt.

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

2.4.4 Regeln für den Abbrechen-Button

Bei Selektion des Abbrechen-Pushbuttons wird das zugehörige Fenster geschlossen.

```
!! Wenn der Abbrechen-Pushbutton selektiert wird,
!! wird das zugehörige Fenster unsichtbar gemacht.
on PbAbbrechen select
{
  this.window.visible := false;
}
```

2.4.5 Regeln für den OK-Button

Wird der OK-Pushbutton selektiert, so wird die Funktion PUTADDR aufgerufen und ihr die Daten des Fensters per Record übergeben.

```
!! Wenn der OK Button selektiert wird,
!! werden die Änderungen an das C-Pogramm übergeben
!! und das Fenster unsichtbar gemacht.
on PbOk select
{
  PUTADDR(Address);
  this.window.visible := false;
}
```

2.4.6 Regeln für den Doppelklick in das Tablefield

Wenn das Tablefield mit einem Doppelklick selektiert wird, so werden alle Daten, die zu dieser Zeile gehören, mit Hilfe der Funktion GETADDR aus der Datei geholt und an das Detailfenster über den *shadows*-Verweis übergeben. Anschließend wird das Detailfenster sichtbar gemacht.

```
!! Wenn ein Doppelklick in das Innere des Tabellenelementes erfolgt,
!! soll der entsprechende Eintrag in dem separaten Fenster
!! bearbeitet werden können.
!! Dazu muss die C-Funktion aufgerufen werden,
!! die die Daten holt und dann an das Fenster übergibt.
on T1 dbselect
{
  !! Zuerst nachschauen, ob in der Tabelle
  !! wirklich etwas selektiert ist.
  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];
    !! Diesen Eintrag nehmen und übergeben.
    GETADDR(Address);
    WnName.title   := "Name Nummer: " + itoa((first(this.activeitem) - 1));
    WnName.visible := true;
  else
    !! Kein sinnvoller Eintrag getroffen, Ton ausgeben.
    beep();
  endif
}
```

2.4.7 Regeln für das Dateiende

Wenn aus der Datei alle Werte gelesen wurden, sendet das C-Programm ein externes Event an den Dialog Manager. Es wird dann eine ***messagebox*** auf den Bildschirm gebracht, die den Benutzer darüber informiert.

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

2.5 Definition der C-Programme

Um nun zu dieser im Dialog Manager definierten Oberfläche ein C-Programm schreiben zu können, muss man folgendes wissen:

- Welche wichtigen Datenstrukturen gibt es im Dialog Manager, die der Kommunikation zwischen dem C-Pogramm und dem Dialog Manager dienen?
- Wie werden Dialog Manager-Datentypen auf C-Datentypen abgebildet?
- Wie wird aus der Dialog Manager Definition eine Definition im C-Programm?
- Wie sieht das Hauptprogramm aus, das einen Dialog des Dialog Managers starten soll?
- Wie sehen vom Dialog Manager aufgerufene C-Unterprogramme aus?
- Wie werden C-Funktionen an den Dialog Manager gebunden?

Diese einzelnen Punkte werden in den folgenden Kapiteln erläutert.

2.5.1 Kommunikation

2.5.1.1 Parameter

Der Dialog Manager bietet zunächst die Möglichkeit, bis zu acht Parameter an eine Funktion zu übergeben. Diese Einschränkung relativiert sich, wenn man in Betracht zieht, dass auch sogenannte Records übergeben werden können. Diese Records sind mit einer Struktur in C gleichzusetzen, da man deren Aufbau und Inhalt selber definieren und damit an seine Bedürfnisse anpassen kann. In einem Record können also mehrere Daten zusammengefasst übergeben werden.

Die Parameter können auf drei verschiedene Arten übergeben werden:

- Wird in der Anwendung nur der Wert eines Parameters benötigt („call by value“), so kann man ihn als Input-Parameter übergeben (Standardeinstellung).
- Soll der Inhalt eines Parameters in der Funktion gesetzt werden können („call by reference“), so muss man ihn als Output-Parameter definieren.
- Wenn ein Parameter teilweise gefüllt ist (z.B. ein Record) und in der Anwendung weiter gefüllt werden soll, dann ist es sowohl ein Input- als auch ein Output-Parameter.

Die Bezeichnung „input“ und „output“ wird immer aus der Sicht der Anwendung betrachtet.

Bitte beachten Sie bei Verwendung von Records als Parameter auch das Kapitel „Funktionen mit Records als Parametern“.

2.5.1.2 Rückgabewerte

In C besitzen Funktionen selber einen Datentyp und können als Ergebnis einen Return-Wert zurückgeben. Auch mit dem Dialog Manager besteht die Möglichkeit, Funktionen von einem integralen Datentyp zu deklarieren, um Werte zurückgeben zu können. Eine `function c boolean PUTADDR (record Address input);` kann zum Beispiel *true* oder *false* als Return-Wert zurückliefern.

2.5.2 Abbildung der Dialogdatentypen

Ausgehend von der Beschreibung des Dialogs können die dort verwendeten Datentypen auf in C benutzbare Datentypen abgebildet werden. Diese Abbildung vom Dialog nach C ist eindeutig und kann der nachfolgenden unvollständigen Tabelle entnommen werden.

Dialogdatentyp	C-Datentyp
object	*DM_ID*
integer	*DM_Integer*
string	*DM_String*
boolean	*DM_Boolean*

2.5.3 Das Hauptprogramm

Bei Programmen, die mit dem Dialog Manager arbeiten sollen, sind spezielle Hauptprogramme notwendig, die vom Prinzip her immer gleich sind und daher aus den mitgelieferten Beispielen kopiert werden können.

Beim Aufbau des Hauptprogramms muss man allerdings unterscheiden, ob man eine lokale Anwendung oder einen Server in einer verteilten Umgebung entwickelt. Bei einer lokalen Anwendung muss eine Funktion mit Namen **AppMain()**, bei einer verteilten Anwendung müssen zwei Funktionen mit Namen **AppInit()** und **AppFinish()** bereitgestellt werden.

Diese Hauptprogramme müssen die vom Dialog Manager erzeugten Header-Dateien enthalten, damit auf die Definitionen des Dialog Managers zugegriffen werden kann.

2.5.3.1 Lokale Anwendungen

Der Aufbau der Funktion **AppMain()** sieht dabei wie folgt aus:

- Zunächst muss die Funktion DM_Initialize zum Initialisieren des Dialog Managers aufgerufen werden. Diese muss unbedingt die erste Funktion im C-Programm sein, die aufgerufen wird. Alle anderen Funktionen führen zu Fehlern.
- Nach der Initialisierung des Dialog Managers kann der zu der Anwendung gehörende Dialog mit Hilfe der Funktion DM_LoadDialog geladen werden.
- Wenn der Dialog erfolgreich geladen wurde, müssen die im Dialog enthaltenen Funktionen vom C-Programm an den Dialog Manager übergeben werden. Dieses erfolgt über den Aufruf der Funktion DM_BindFunctions für Funktionen, die keine Records als Parameter haben und über die mit Hilfe des Simulationsprogramms generierten C-Funktion *RecMInit<Name des Dialogs oder Moduls>* für Funktionen, die Records als Parameter haben.
  Hierdurch werden die Adressen an die Funktionsaufrufe gebunden.

- Nach der Anbindung der Funktionen an den Dialog sollten anwendungsspezifische Initialisierungen durchgeführt werden.
- Anschließend muss der Dialog für den Anwender mit Hilfe der Funktion DM_StartDialog gestartet werden. Beim Aufruf dieser Funktion wird die Startregel im Dialog `on dialog start` ausgeführt und alle im Dialog als sichtbar definierten Fenster sichtbar gemacht.
- Danach wird die Kontrolle an den Dialog Manager übergeben, indem die Funktion DM_EventLoop aufgerufen wird. Diese Funktion kehrt normalerweise erst beim Programmende, nach Abarbeitung der Regel `on dialog finish` (wenn sie vorhanden ist) zurück. Anweisungen nach dem Aufruf dieser Funktion werden also erst beim Beenden der Anwendung ausgeführt.

Diese Funktion hat den Namen **AppMain()** und sieht in etwa wie folgt aus:

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

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

  /* Laden der Dialogdatei */
  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);
  }

  /* Eintrag der Adressen der Funktionen ohne Records */
  DM_BindFunctions (FuncMap, FuncCount, dialogID, 0, 0);

  /* Eintrag der Adressen der Funktionen mit Records */
```

```
  RecMInitTABLEDEMO(dialogID, 0);

  /* Starten des Dialoges und Eintritt in die Event-Loop */
  if (DM_StartDialog (dialogID, 0))
    DM_EventLoop (0);
  else
    return (1);

  return (0);
}
```

2.5.3.2 Verteilte Anwendungen

Bei verteilten Anwendungen müssen zwei C-Funktionen in der Anwendung bereitgestellt werden, um die Anwendung initialisieren bzw. beenden zu können. Die Initialisierungsfunktion heißt **AppInit()**, die Beendigungsfunktion **AppFinish()**.

Der Aufbau der Funktion **AppInit()** sieht dabei wie folgt aus:

- Diese Funktion muss die im Dialog enthaltenen Funktionen vom C-Programm an den Dialog Manager übergeben. Dies erfolgt über den Aufruf der Funktion DM_BindFunctions und der mit Hilfe des Simulationsprogramms generierten C-Funktion *RecMInit<Name des Dialogs oder Moduls>* für Funktionen, die einen Record als Parameter haben.
- Nach der Anbindung der Funktionen an die Applikation sollten anwendungsspezifische Initialisierungen durchgeführt werden.

In der Funktion **AppFinish()** müssen dann die Aktionen durchgeführt werden, die ein kontrolliertes Beenden der Anwendung ausführen. Aufrufe an den Dialog Manager sind hierbei nicht notwendig, die Server-Anwendung wird automatisch nach der Rückkehr der Funktion beendet.

Beispiel

Funktion 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;
}
```

Funktion AppFinish()

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

2.5.4 Hilfsmittel für die C-Programmierung

Damit die im Dialog getroffenen Definitionen für Parameter der C-Funktionen nicht zweimal definiert werden müssen, kann der Simulator des Dialog Managers aus einer Dialogdatei die für das C-Programm notwendigen Dateien generieren. Diese Dateien sollten auf keinen Fall verändert werden, da sie die Schnittstelle vom Dialog Manager zur Anwendung darstellen.

In den Dateien werden die Prototypen der Funktionen und ggf. die Abbildungen der Records des Dialog Managers auf Strukturen für das C-Programm definiert. Die Definition der Prototypen sollte als Funktionsdefinition im C-Programm benutzt werden, um einen konsistenten und fehlerfreien Funktionsaufruf zu gewährleisten.

2.5.4.1 Funktionen ohne Records als Parameter

Das Simulationsprogramm des Dialog Managers erzeugt eine Include-Datei, die alle Prototypen der im Dialog Manager definierten Funktionen enthält. Diese kann dann in die entsprechenden C-Sourcen eingebunden werden. Der jeweilige C-Compiler überprüft die Funktionsdefinitionen mit der Realisierung der Funktion und teilt dann Unterschiede mit.

Das Erzeugen dieser Prototypendatei erfolgt durch die Startoption **+writeproto <Name der Header-Datei>**.

Für unser Beispiel sieht die Kommandozeile wie folgt aus:

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

Durch diesen Aufruf entsteht die Datei **bsp.h**.

```
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));
```

Anstelle dieser Option kann auch die Startoption **+writefuncmap** verwendet werden, mit der sowohl die Prototypen als auch die Funktionstabelle erzeugt werden. Die Funktionstabelle wird dabei in einer C-Datei abgelegt, so dass zur Anbindung der Funktionen die enthaltene Funktion *BindFunctions_<Name des Dialogs>* aufgerufen werden müsste.

In der Funktion **AppMain()** werden diese Funktionen, die dem Dialog Manager zur Verfügung gestellt werden sollen, in die „FuncMap“ eingetragen. Das ist eine Struktur, die die Adressen der Funktionen bereitstellt, damit im Dialog Manager auf diese Definitionen im C-Programm zugegriffen werden kann. Die definierte Variable „FuncCount“ gibt dabei die Anzahl der in der Tabelle definierten Funktionen an. Es ist darauf zu achten, dass nur die Funktionen in **AppMain()** eingetragen werden, die keine Records als Parameter besitzen, da diese über einen anderen Mechanismus an den Dialog angebunden werden.

```
#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 Funktionen mit Records als Parametern

Das Simulationsprogramm des IDM erzeugt eine Include-Datei mit den Prototypen der im Dialog Manager definierten Funktionen mit Records als Parametern und die für ihren Aufruf notwendigen Funktionen über eine Startoption. Die dabei entstehende C-Datei muss mit übersetzt und zu der Anwendung dazu gelinkt werden. Die entstehende Include-Datei sollte für die eigentliche Funktionsanbindung verwendet werden, denn darin sind auch die Strukturen definiert.

Dieses erfolgt durch die Startoption **+writetrampolin <Basis-Name> <Dialog-Name>**

Dabei wird mit Basis-Name der Name einer Datei angegeben, an den die Endungen „.h“ und „.c“ angehängt werden, um die für den Aufruf der C-Funktionen notwendigen Dateien zu generieren. In der **.h**-Datei werden unter anderem die Prototypen der Funktionen und die Abbildung der Records auf C-Strukturen definiert.

In der **.c**-Datei werden zum einen die Funktionen mit Records als Parameter in die *RecMap[]* eingetragen und so dem Dialog Manager bekannt gemacht.

Zum anderen werden dort die Funktionen generiert, die in der Funktion **AppMain()** aufgerufen werden müssen, um auch die hier definierten Funktionen an den Hauptdialog zu binden.

Für unser Beispiel sieht die Kommandozeile wie folgt aus:

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

Durch diesen Aufruf entstehen die Dateien

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

Diese Include-Datei **tram_bsp.h** hat für unser Beispiel folgendes Aussehen:

```
#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
```

Die generierten C- und Header-Dateien müssen mit übersetzt und zu der Anwendung hinzu gelinkt werden. Die Header-Dateien sollten benutzt werden, um auf die im Dialog getroffenen Definitionen der Übergabestruktur zugreifen zu können und die Definitionen der Funktionen in das C-Programm zu übernehmen.

2.5.5 Funktionen für das Tablefield

Funktionen, die den Inhalt von Objekten mit Listencharakter füllen oder abfragen, sind in Bezug auf das Schichtenmodell eine Ausnahme. In diesem Fall ist eine **DM_**-Funktion angebracht, damit die Anwendungen performant ablaufen können. Das Füllen eines Tablefields muss mit Hilfe eines Vektors erfolgen. Dieser Bereich wird dann von der Anwendung gefüllt und anschließend dem Objekt zugewiesen. Durch diese Vorgehensweise wird bei lokalen Anwendungen das ständige Flackern des Tablefields beim Füllen verhindert und bei verteilten Anwendungen zusätzlich eine sehr hohe Performanzsteigerung gegenüber der Einzelzuweisung erreicht. Im einzelnen sieht die Vorgehensweise wie folgt aus:

- Ist festgelegt, wie viele Daten geladen werden, kann man diesen Speicherbereich statisch allokieren. Andernfalls kann ein dynamischer Bereich angelegt werden; dieses erfolgt mit Hilfe der Funktion DM_Malloc.
- Nach dem Anlegen des Speicherbereiches werden die zu setzenden Attribute in diesem Bereich abgelegt.
- Die Daten werden dem Objekt des Dialog Managers in einem Parameter vom Typ **DM_VectorValue** im Element *vector* übergeben.
- Das DM-Objekt kann dann mit Hilfe der Funktionen DM_SetVectorValue oder DM_SetContent gefüllt werden. Dazu müssen die **DM_Value**-Strukturen den Typ *DT_index* für zweidimensionale Objekte oder *DT_integer* für eindimensionale Objekte und die Indizes der Elemente enthalten, die zuerst und zuletzt gesetzt werden sollen. Es ist darauf zu achten, dass die Funktion **DM_SetVectorValue()** nur rechteckige Bereiche setzen kann und daher auch die Anzahl der zu setzenden Objekte (*count* in **DM_VectorValue**) mit diesen Indizes übereinstimmen muss. Die eigentlichen Werte werden in der Struktur **DM_VectorValue** übergeben. Hier steht der Datentyp, die Anzahl der zu setzenden Attribute und der Vektor des zu setzenden Inhalts.

Das Auslesen von Werten läuft analog zum Abfragen der Werte im Objekt:

- Abholen der Werte aus dem Objekt mit Hilfe der Funktion DM_GetVectorValue.
- Zugriff auf die Struktur mit normalem C.

» Freigeben des temporären Bereiches über die Funktion DM_FreeVectorValue.

2.5.5.1 Funktion FILLTAB()

Diese Funktion übernimmt das initiale Füllen des Tablefields. Dazu wird dieser Funktion die ID des Tablefields und die Anzahl der zu füllenden Elemente übergeben.

```
/*
 * Funktion zum Fuellen des Tablefields bei Dialogstart
 * Parameter:
 *  table  ist die ID des Tablefields, das gefuellt wird
 *  number ist die Anzahl der Datensaetze, die geladen werden
 *  header ist die Anzahl der Header des Tablefields
 * Rueckgabewert:
 *  Boolescher Wert, der anzeigt, ob die Daten erfolgreich geladen wurden
 */
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);
  }

  /* erste Zelle des Tablefields, die gefuellt wird */
  startIdx.type = DT_index;
  startIdx.value.index.first = (ushort) header;
```

```
startIdx.value.index.second = 1;

/* letzte Zelle des Tablefields, die gefuellt wird */
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;
  }

  /* Freigabe nur mit Dialog Manager Funktionen! */
  while (--vpos >= 0)
  {
    DM_Free(strvec[vpos]);
  }
  DM_Free(buffer);
  return retval;
}
```

2.5.5.2 Funktion CONTENT()

Mit Hilfe dieser Funktion soll das Nachladen des Tablefields erreicht werden. Immer wenn der Benutzer in einen Bereich scrollt, der noch nicht gefüllt ist, wird vom Dialog Manager automatisch diese Funktion aufgerufen.

Die Parameter sind fest vom Dialog Manager definiert und können daher nicht geändert werden. Diese Nachlade-Funktionen erhalten als Parameter die Struktur **DM_ContentArgs**, in der die notwendigen Informationen gespeichert sind.

In *object* wird die ID des Tablefields übergeben. Die Felder *visfirst* und *vislast* enthalten die erste und letzte sichtbare Zeile, die Felder *loadfirst* und *loadlast* die erste und letzte zu ladende Zeile. Dies sind jedoch nur die Minimalwerte, die Anwendung darf jederzeit mehr als die angegebenen Zeilen laden. In diesem Beispiel werden immer fünf Zeilen geladen.

```
/*
 * Funktion zum dynamischen Auffuellen des Tablefields beim scrollevent,
 * wenn Zeilen im Tablefield erreicht werden, die noch nicht gefuellt sind.
 */
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;

  /* erste Zelle des Tablefields, die gefuellt wird */
  startIdx.type = DT_index;
  startIdx.value.index.first = args->loadfirst - 1;
  startIdx.value.index.second = 1;

  /* letzte Zelle des Tablefields, die gefuellt wird */
  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;

  /*
   * Einlesen des Inhalts und Aufbereiten der Daten
   * HIER: GELÖSCHT!
   */

  vector.count = (ushort) vpos;

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

  /*
   * Freigabe der durch den Dialog Manager allokierten Speicherbereiche
   * nur mit Dialog Manager Funktionen!
   */
  while (--vpos >=0 )
    DM_Free(strvec[vpos]);

  DM_Free(buffer);

  /*
   * Sollte das Ende der Datei erreicht werden, wird dieses dem Dialog
Manager
   * durch ein externes Event bekannt gegeben, damit er darauf reagieren
kann.
   * Hier wird eine Messagebox ausgegeben.
   * Da in Callback-Funktionen auf DM_- Funktionen verzichtet werden soll,
   * hier ein Beispiel, wie der Dialog Manager auf Ereignisse reagieren kann.
```

```
     * Die Funktion DM_QueueExtEvent ist die einzige Funktion,
     * die sicher abgearbeitet wird.
     */
    if(feof(f))
    {
      DM_Value *null = 0;
      DM_QueueExtEvent (args->object,9999,0,null, DMF_DontTrace);
    }
}
```

2.5.5.3 Funktion FILECLOSE()

Funktion zum Schließen der Datei, wenn der Dialog beendet wird.

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

2.5.6 Funktionen mit Records als Parameter

Im Gegensatz zu den Funktionen für das Tablefield können die nachfolgenden Funktionen ohne Wissen über den Dialog auskommen und sollten das auch. Diese Funktionen werden daher ohne jeglichen Aufruf an den Dialog Manager in Standard-C geschrieben. Lediglich der Eintrag in der „FuncMap“ zeigt, dass es sich um Funktionen handelt, die vom Dialog Manager aufgerufen werden.

2.5.6.1 Funktion GETADDR()

Diese Funktion soll zu einem gegebenen Namen den restlichen Datensatz an die Oberfläche liefern. Damit auf die Definitionen des Dialoges zugegriffen werden kann, wurde mit **+writetrampolin** eine entsprechende C-Struktur generiert.

```
/*
 * Funktion zum Holen der vollstaendigen Daten eines im Tablefield
 * angewaehlten Datensatzes. Hier muesste eine Funktion mit Dateiverarbeitung
 * geschrieben werden. Zur Vereinfachung wird der zu fuellende Record mit
 * einem festen Wert gefuellt.
 * Parameter:
 *  Address ist ein Record, der teilweise im Dialog Manager gefuellt wurde
 *              und im Programm um die fehlenden Eintraege ("Strasse")
 *              ergaenzt wird
 * Rueckgabewert:
 *  Boolescher Wert, der anzeigt, ob die Daten erfolgreich geholt wurden
```

```
 */
DM_Boolean DML_default DM_ENTRY GETADDR __1((RecAddress *, Address))
{
  Address->Street = "Neue Strasse";
  return (DM_TRUE);
}
```

2.5.6.2 Funktion PUTADDR()

Diese Funktion soll die vom Benutzer geänderten Daten wieder abspeichern. Damit auf die Definitionen des Dialoges zugegriffen werden kann, wurde mit dem Simulationsprogramm über die Option **+writetrampolin** eine entsprechende C-Struktur generiert.

```
/*
 * Funktion zum Schreiben der im Dialog geaenderten Daten in die Datei.
 * Diese Funktion ist hier nicht ausprogrammiert.
 */
DM_Boolean DML_default DM_ENTRY PUTADDR __1((RecAddress *, Address))
{
  Address->Street = "";
  return(DM_TRUE);
}
```

2.6 Übersetzen und Linken

Abschließend müssen nun die erstellten Sourcen übersetzt werden. Die einzelnen Compiler-Optionen sind dabei umgebungsabhängig. Es sollten die in den Beispielen ausgelieferten Makefiles genommen und an die örtlichen Gegebenheiten angepasst werden.

Prinzipiell sind zum Bauen einer ablauffähigen Anwendung folgende Schritte notwendig:

1. Mit Hilfe des Simulationsprogramms des Dialog Managers werden die notwendigen Dateien generiert.
   - » Für Dialoge mit Funktionen ohne Records als Parameter:

     ```
     idm  +writeproto  <Header-Name>  <Dialogname>
     ```

     hier

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

   - » Für Dialoge mit Funktionen mit Records als Parameter:

     ```
     idm  +writetrampolin  <Basisname>  <Dialogname>
     ```

hier

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

2. Die generierten Header und die C-Datei werden mit Hilfe des C-Compilers und den für den Dialog Manager üblichen Compiler-Optionen übersetzt.
3. Anschließend werden die übersetzten Dateien zu einem Executable zusammen gelinkt.

Für einen lauffähigen Dialog sind somit also mindestens ein C-Executable und eine Dialogdatei vom Dialog Manager (Binär- oder ASCII-File) notwendig.

3 Datentypen

Zur Kommunikation zwischen Dialog Manager und Anwendung sind verschiedene Datentypen definiert, die durch das Einbinden der Datei **IDMuser.h** verfügbar werden. Sie dienen dabei sowohl der Übergabe von Werten an den DM als auch der Rückgabe von Werten des DM an die Anwendung. Diese Datentypen werden dann innerhalb zusammengesetzter Strukturen und bei den Funktionsparametern verwendet. Notwendig sind diese Datentypen, da im Zusammenhang mit anderen Werkzeugen die Definitionen oft nicht einheitlich sind.

Im nächsten Kapitel werden zunächst die Grunddatentypen vorgestellt, im darauffolgenden Kapitel die davon abgeleiteten Dialog Manager Datentypen und abschließend die zusammengesetzten Strukturen.

3.1 Grunddatentypen

In den nachfolgenden Kapiteln werden alle vom Dialog Manager definierten Grunddatentypen und deren Bedeutung vorgestellt. Diese Grunddatentypen sind definiert, damit auf allen vom Dialog Manager unterstützten Plattformen dieselben Inhalte in den entsprechenden Dialog Manager Datentypen abgelegt werden können. Sie sorgen also z.B. dafür, dass eine Zahl immer im Bereich von -2147483648 bis 2147483647 liegen kann, unabhängig davon wie groß eigentlich eine Integerzahl auf der jeweiligen Architektur ist.

3.1.1 Grunddatentyp DM_Int

Über diesen Datentyp wird ein "int"-Datentyp definiert, wie er im normalem C verwendet wird. Dieser Datentyp ist von seiner Größe her abhängig vom verwendeten Basissystem; auf dem PC also 2 Byte und auf den meisten Unix-Anlagen 4 Byte.

Definition

```
typedef int DM_Int;
```

3.1.2 Grunddatentyp DM_UInt

Über diesen Datentyp wird ein "int"-Datentyp ohne Vorzeichen definiert. Dieser Datentyp ist von seiner Größe her abhängig von dem verwendeten Basissystem., auf dem PC also 2 Byte und auf den meisten Unix-Anlagen 4 Byte.

Definition

```
typedef unsigned int DM_UInt;
```

3.1.3 Grunddatentyp DM_Int1

Mit Hilfe dieses Datentyps wird eine 1 Byte große Zahl mit Vorzeichen definiert. Sie hat damit einen Wertebereich von -128 bis 127.

Definition

```
typedef char  DM_Int1;
```

3.1.4 Grunddatentyp DM_UInt1

Mit Hilfe dieses Datentyps wird eine 1 Byte große Zahl ohne Vorzeichen definiert. Sie hat damit einen Wertebereich von 0 bis 255.

Definition

```
typedef unsigned char  DM_UInt1;
```

3.1.5 Grunddatentyp DM_Int2

Mit Hilfe dieses Datentyps wird eine 2 Byte große Zahl mit Vorzeichen definiert. Sie hat damit einen Wertebereich von -32768 bis 32767.

Definition

```
typedef short DM_Int2;
```

3.1.6 Grunddatentyp DM_UInt2

Mit Hilfe dieses Datentyps wird eine 2 Byte große Zahl definiert. Sie hat damit einen Wertebereich von 0 bis 65535.

Definition

```
typedef short DM_UInt2;
```

3.1.7 Grunddatentyp DM_Int4

Mit Hilfe dieses Datentyps wird eine 4 Byte große Zahl mit Vorzeichen definiert. Sie hat damit einen Wertebereich von -2147483648 bis 2147483647.

Definition

Auf Hardware-Architekturen mit 2 oder 4 Byte Adressierung sieht die Definition wie folgt aus:

```
typedef long DM_Int4;
```

Auf Hardware-Architekturen mit 8 Byte Adressierung (z.B. DEC Alpha) sieht die Definition wie folgt aus:

```
typedef int DM_Int4;
```

3.1.8 Grunddatentyp DM_UInt4

Mit Hilfe dieses Datentyps wird eine 4 Byte große Zahl ohne Vorzeichen definiert. Sie hat damit einen Wertebereich von 0 bis 4294967295.

Definition

Auf Hardware-Architekturen mit 2 oder 4 Byte Adressierung sieht die Definition wie folgt aus:

```
typedef unsigned long DM_UInt4;
```

Auf Hardware-Architekuren mit 8 Byte Adressierung (z.B. DEC Alpha) sieht die Definition wie folgt aus:

```
typedef unsigned int DM_UInt4;
```

3.2 Dialog Manager Datentypen

In den nachfolgenden Kapiteln werden alle vom Dialog Manager definierten Datentypen und deren Bedeutung vorgestellt. Diese Datentypen kommen sowohl als Typen von Parametern der Interface- und Anwendungsfunktionen vor als auch als Elemente der vom Dialog Manager definierten zusammengesetzten Strukturen.

3.2.1 Datentyp DM_Attribute

Mit Hilfe dieses Datentyps wird definiert, wie ein Attribut abgespeichert wird.

Definition

```
typedef DM_UInt4  DM_Attribute;
```

3.2.2 Datentyp DM_Boolean

Mit Hilfe dieses Datentyps wird ein boolescher Wert definiert, der den Wert TRUE oder FALSE annehmen kann.

Definition

```
typedef DM_UInt  DM_Boolean;
```

3.2.3 Datentyp DM_Class

Mit Hilfe dieses Datentyps wird definiert, wie eine Klasse abgespeichert wird.

Definition

```
typedef DM_UInt  DM_Class;
```

3.2.4 Datentyp DM_Enum

Mit Hilfe dieses Datentyps wird definiert, wie der Aufzählungstyp abgespeichert wird.

Definition

```
typedef DM_UInt  DM_Enum;
```

3.2.5 Datentyp DM_ErrorCode

Mit Hilfe dieses Datentyps wird definiert, wie die Fehlercodes intern abgespeichert werden.

Definition

```
typedef DM_UInt4  DM_ErrorCode;
```

3.2.6 Datentyp DM_Event

Mit Hilfe dieses Datentyps wird definiert, wie die Nummer eines Ereignisses abgespeichert wird.

Definition

```
typedef DM_UInt  DM_Event;
```

3.2.7 Datentyp DM_Integer

Mit Hilfe dieses Datentyps wird eine 4 Byte große Zahl definiert.

Definition

```
typedef DM_Int4  DM_Integer;
```

3.2.8 Datentyp DM_ID

Mit Hilfe dieses Datentyps wird definiert, wie Objekte intern abgespeichert werden.

Definition

```
typedef DM_UInt4  DM_ID;
```

3.2.9 Datentyp DM_Method

Mit Hilfe dieses Datentyps wird definiert, wie eine Methode abgespeichert wird.

Definition

```
typedef DM_UInt4  DM_Method;
```

3.2.10 Datentyp DM_Options

Mit Hilfe dieses Datentyps wird definiert, wie die Optionen der Interface-Funktionen an den Dialog Manager übergeben werden.

Definition

```
typedef DM_UInt  DM_Options;
```

3.2.11 Datentyp DM_Pointer

Mit Hilfe dieses Datentyps wird ein Zeiger auf einen beliebigen Bereich definiert. Dieser Zeiger ist je nach System aufgebaut, ist aber mindestens eine 4 Byte Adresse. Seine Definition selber ist vom Basissystem abhängig.

Definition

```
typedef void * DM_Pointer;
```

oder

```
typedef char * DM_Pointer;
```

3.2.12 Datentyp DM_Scope

Mit Hilfe dieses Datentyps wird definiert, wie die Art eines Objektes (Instanz, Modell oder Default) abgespeichert wird.

Definition

```
typedef DM_UInt  DM_Scope;
```

3.2.13 Datentyp DM_String

Mit Hilfe dieses Datentyps wird ein Zeiger auf einen String definiert

Definition

```
typedef char *  DM_String;
```

Zur Behandlung von String Parametern in Dialog Manager Funktionen bitte unbedingt im Handbuch „C-Schnittstelle - Funktionen“ das Kapitel „Behandlung von String-Parametern“ und die jeweiligen Funktionsbeschreibungen beachten.

3.2.14 Datentyp DM_Type

Mit Hilfe dieses Datentyps wird definiert, wie der Datentyp eines Attributes abgespeichert wird.

Definition

```
typedef DM_UInt  DM_Type;
```

3.2.15 NULL- DM_ID

Mit Hilfe der DM_ID 0 wird das Zurücksetzen von Ressource-Attributen ermöglicht.

Diese DM_ID wird NULL-Objekt genannt. Dieses NULL-Objekt kann dazu benutzt werden, alle Ressource-Attribute zurückzusetzen. Es kann von **DM_SetValue** erhalten werden und kann für **DM_GetValue** oder für Funktionsargumente benutzt werden.

Das NULL-Objekt selbst ist typunabhängig, d.h. es gibt nur ein NULL-Objekt für alle Ressourcen und Objekte. Das NULL-Objekt kann auf alle Datentypen angewendet werden, die intern ein Objekt sind. Bei GetValue ist der Realtyp des NULL-Objekts abhängig vom Typ des Attributes.

Auswirkungen

- Schnittstellen-Funktionen können ein NULL-Objekt als Return-Wert haben.Das NULL-Objekt kann auch als („input“ und „output“) Parameter vorkommen.
- Wenn ein NULL-Objekt als Return-Wert von **DM_GetValue** erhalten wird, wird es immer den genauest möglichen Datentyp haben, z.B. *DT_text* anstatt *DT_instance*.
- Wenn ein Attribut von der Anwendung auf *null* gesetzt werden soll, muss der Datentyp so genau wie möglich angegeben werden, d.h. z.B. *DT_accelerator* anstatt *DT_instance*.

3.3 Zusammengesetzte Strukturen zum Setzen und Abfragen von Attributen

In den nachfolgenden Kapiteln werden alle vom Dialog Manager definierten Strukturen und deren Bedeutung vorgestellt. Diese Strukturen dienen der Kommunikation zwischen der Anwendung und dem Dialog Manager.

3.3.1 DM_Index

Mit Hilfe dieser Struktur wird ein zweidimensionaler Indexwert beschreiben. Er wird also z.B. benutzt, wenn auf Elemente im Inneren des Tablefields zugegriffen wird.

Definition

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

Bedeutung der Elemente

DM_UInt2 first

In diesem Element wird der erste Teilindex abgelegt.

DM_UInt2 second

In diesem Element wird der zweite Teilindex abgelegt.

3.3.2 Struktur DM_ValueUnion

Mit Hilfe dieser Struktur können Daten im Dialog abgefragt und gesetzt werden. Diese Struktur erlaubt es, genau einen Attributwert im Dialog zu erfragen bzw. genau einen Attributwert im Dialog zu setzen. Diese Struktur ist in C als Union definiert, d.h. immer genau ein Wert innerhalb dieser Struktur ist gültig, auf alle anderen kann zu diesem Zeitpunkt nicht zugegriffen werden.

```
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;
```

Bedeutung der Elemente

DM_UInt cardinal

Dieser Eintrag ist nur noch aus Kompatibilitätsgründen zur Version 2 des Dialog Managers vorhanden. Wenn dieser Eintrag angesprochen werden soll, muss neben den üblichen Definitionen beim Übersetzen der Sourcen mit dem C-Compiler auch noch das Symbol DM_COMPAT_CARDINAL definiert werden. Ist dieses Symbol definiert, kann über diesen Eintrag auf Werte vom Typ Aufzählung (DT_enum), Typ (DT_type) und Scope (DT_scope) zugegriffen werden.

DM_Scope scope

In diesem Element werden die Daten übergeben, wenn die Art eines Objektes (Instanz, Modell oder Default) erfragt werden soll. Dabei entspricht der Wert 1 einem Default, der Wert 2 einem Modell und der Wert 3 einer Instanz.

DM_Boolean boolean

In diesem Element werden die Werte von booleschen Attributen hinterlegt. Dabei kann dieses Element den Wert TRUE oder FALSE annehmen.

DM_String string

In diesem Element werden die Werte von Attributen mit Textcharakter hinterlegt. Dieses Element ist dann ein Zeiger auf den entsprechenden String.

DM_Integer integer

In diesem Element werden die Werte von Attributen mit Zahlencharakter hinterlegt.

DM_Pointer pointer

In diesem Element werden von der Anwendung definierte Zeiger übergeben. Der Inhalt dieser Zeiger ist dabei dem Dialog Manager unbekannt, er reicht lediglich den entsprechenden Zeiger weiter.

DM_Class classid

In diesem Element wird die Klasse eines Objektes übergeben. Es kann damit also alle Werte annehmen, die in der Include-Datei "IDMuser.h" mit "DM_Class" anfangen. Eine vollständige Tabelle der verfügbaren Klassen kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Event event

In diesem Element werden Ereignisse gespeichert. Diese Ereignisse sind in der Include-Datei "IDMuser.h" definiert. Für eine ausführliche Erklärung muss das Dialog Manager Manual "Regelsprache" herangezogen werden. Eine vollständige Tabelle der Ereignisse kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Type type

In diesem Element werden Datentypen gespeichert. Diese Datentypen sind ebenfalls in der Include-Datei "IDMuser.h" definiert. Alle diese Datentypen fangen mit dem Prefix "DT_" an. Eine vollständige Tabelle aller Datentypen kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Enum enumval

In diesem Element werden alle Datentypen übergeben, die intern als Aufzählungstyp gehandhabt werden. Diese Konstanten fangen alle mit dem Prefix "DM_" an. Eine vollständige Tabelle aller verfügbaren Aufzählungskonstanten kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Index index

In diesem Element werden zwei Werte für die betroffene Spalte und Zeile, also zwei Zahlen, übergeben.

DM_Method method

In diesem Element werden Methoden übergeben. Eine vollständige Tabelle aller verfügbaren Methoden kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_ID id

In diesem Element wird die Objekt-ID eines Dialog Manager Objektes übergeben.

3.3.3 Struktur DM_Value

Die Struktur dient dem eigentlichen Datenaustausch zwischen Anwendung und Dialog Manager. Sie beinhaltet die DM_ValueUnion zur Übergabe der eigentlichen Werte. Zusätzlich enthält sie weitere Informationen zu dem abgefragten bzw. gesetzten Attribut.

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

Bedeutung der Elemente

DM_Boolean inherit

Über das Element *inherit* kann beim Abfragen von Werten im DM die Information erfragt werden, ob das Objekt den Wert selbst weiß oder ob das Objekt diesen Wert von seinem Modell oder Default geerbt (*inherit*) hat. Ist dieses Element auf TRUE nach der Abfrage eines Attributes, so hat das Objekt den Wert nur geerbt, ist es FALSE, so hat das Objekt den Wert selbst.

DM_Boolean changed

Das Strukturelement *changed* dient vor allem dem internen Gebrauch im DM.

Dieses Element kann aber auch nach dem Setzen eines Attributwertes abgefragt werden, ob durch die Wertzuweisung das betroffene Attribut wirklich einen neuen Wert erhalten hat. In diesem Fall steht *changed* auf TRUE. Falls das Attribut den neu zugewiesenen Wert schon vorher hatte, steht *changed* auf FALSE.

DM_UInt1 type

Wird über DM_Value beim DM ein Wert erfragt, so setzt der DM entsprechend dem erfragten Attribut das Strukturelement *type*. Wird hingegen im Dialog Manager von der Anwendung aus ein Wert gesetzt, so muss die Anwendung den richtigen Typ in diesem Element hinterlegen. Abhängig von dem hier gespeicherten Wert ist die nachfolgende Struktur DM_ValueUnion belegt bzw. zu belegen.

Dabei gibt es folgende mögliche Belegungen:

Name	Belegtes Element	Bedeutung
DT_accel	id	Acceleratorreferenz.
DT_attribute	attribute	Dieser Datentyp enthält die Bezeichnung eines Attributes.
DT_boolean	boolean	Logischer Wert TRUE oder FALSE.
DT_class	classid	Dieser Datentyp beschreibt die Klasse der unterschiedlichen Objekte.
DT_color	id	Farbreferenz.
DT_cursor	id	Cursorreferenz.
DT_datatype	type	Zur Angabe von Datentypen als Wert von Variablen.
DT_enum	enum	Wird z.B. für die Definition von Icons und Buttons der ***messagebox***, für ***tablefield*** und ***setup***-Objekt benötigt.
DT_event	event	Variablen oder Attribute können als Werte verschiedene Ereignisse annehmen, die in der Ereigniszeile der Regel angegeben werden können. (Immer nur ein Ereignis kann als Wert angegeben werden!)
DT_font	id	Zeichensatzreferenz.
DT_func	id	Funktionsreferenz
DT_hash	*pointer*	Assoziatives Feld
DT_index	index	Mit diesem Datentyp können zweidimensionale Indices als ein Wert abgelegt werden (z.B. für Tablefield -> z.T. Attribute, die Koordinaten von Tablefield-Element beinhalten. Werte vom Typ DT_index werden mit [y, x] definiert.
DT_instance	id	Instanzreferenz.
DT_integer	integer	Dieser Datentyp dient zur Beschreibung von Zahlen.
DT_list	*pointer*	Werteliste
DT_matrix	*pointer*	Zweidimensionales Feld
DT_method	method	Methode.
DT_object	id	Objektreferenz.

Name	Belegtes Element	Bedeutung
DT_pointer	pointer	beliebiger Zeiger/Pointer.
DT_refvec	*pointer*	Referenzliste mit DM_ID-Werten, bei der dieselbe ID nur einmal vorkommen kann
DT_rule	id	Regelreferenz.
DT_scope	scope	Mit Hilfe dieses Datentyps wird beschrieben, ob es sich um ein Defaultobjekt, ein Modell oder ein Objekt handelt.
DT_string	string	Mit diesem Datentyp wird eine Zeichenkette übergeben.
DT_text	id	Textreferenz.
DT_tile	id	Musterreferenz.
DT_var	id	Variablenreferenz.
DT_vector	*pointer*	Feld mit gleichem Wertetyp

Die möglichen Werte für *DT_Scope* sind

- 1: Defaultobjekt
- 2: Modell
- 3: Instanz

3.3.4 Struktur DM_VectorValue

Diese Struktur dient dem effizienten Setzen und Abfragen von vektoriellen Attributen. Diese Struktur hat einen ähnlichen Aufbau wie die DM_Value Struktur. In der äußeren Struktur wird angegeben, welchen Datentyp die zu setzenden bzw. abgefragten Werte haben und wie viele Werte übergeben werden. In der inneren Struktur werden dann die eigentlichen Werte übergeben.

```
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;
```

Bedeutung der Elemente

DM_UInt1 type

Wird über DM_VectorValue beim DM ein Wert erfragt, so setzt der DM entsprechend den erfragten Attributen das Strukturelement *type*. Wird hingegen im Dialog Manager von der Anwendung aus ein Wert gesetzt, so muss die Anwendung den richtigen Typ in diesem Element hinterlegen. Abhängig von dem hier gespeicherten Wert, ist die nachfolgende Struktur "vector" belegt bzw. zu belegen.

DM_UInt count

In diesem Element wird beim Abfragen angegeben, wie viele Elemente der Dialog Manager in der Unterstruktur "vector" abgespeichert hat. Beim Setzen von Attributen von der Anwendung aus muss hier die Anwendung die Anzahl der zu setzenden Attribute hinterlegt haben.

DM_UInt *cardinalPtr

Dieser Eintrag ist nur noch aus Kompatibilitätsgründen zur Version 2 des Dialog Managers vorhanden. Wenn dieser Eintrag angesprochen werden soll, muss neben den üblichen Definitionen beim Übersetzen der Sourcen mit dem C-Compiler auch noch das Symbol DM_COMPAT_ CARDINAL definiert werden. Ist dieses Symbol definiert, kann über diesen Eintrag auf Werte vom Typ Aufzählung (DT_enum), Typ (DT_type) und Scope (DT_scope) zugegriffen werden.

DM_Scope *scopePtr

In diesem Element werden die Daten übergeben, wenn die Art eines Objektes (Instanz, Modell oder Default) erfragt werden soll. Dabei entspricht der Wert 1 einem Default, der Wert 2 einem Modell und der Wert 3 einer Instanz.

DM_Boolean *booleanPtr

In diesem Element werden die Werte von booleschen Attributen hinterlegt. Dabei kann dieses Element den Wert TRUE oder FALSE annehmen.

DM_String *stringPtr

In diesem Element werden die Werte von Attributen mit Textcharakter hinterlegt. Dieses Element ist dann ein Array von Zeigern auf Strings.

DM_Integer *integerPtr

In diesem Element werden die Werte von Attributen mit Zahlencharakter hinterlegt. Dieses Element ist also ein Array von Zahlen.

DM_Pointer *pointerPtr

In diesem Element werden von der Anwendung definierte Zeiger übergeben. Der Inhalt dieser Zeiger ist dabei dem Dialog Manager unbekannt, er reicht lediglich den entsprechenden Zeiger weiter.

DM_Class *classidPtr

In diesem Element wird die Klasse eines Objektes übergeben. Es kann damit also alle Werte annehmen, die in der Include-Datei "IDMuser.h" mit "DM_Class" anfangen. Eine vollständige Tabelle der verfügbaren Klassen kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Event *eventPtr

In diesem Element werden Ereignisse gespeichert. Diese Ereignisse sind in der Include-Datei "IDMuser.h" definiert. Für eine ausführliche Erklärung muss das Dialog Manager Manual "Regelsprache" herangezogen werden. Eine vollständige Tabelle der Ereignisse kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Type *typePtr

In diesem Element werden Datentypen gespeichert. Diese Datentypen sind ebenfalls in der Include-Datei "IDMuser.h" definiert. Alle diese Datentypen fangen mit dem Prefix "DT_" an. Eine vollständige Tabelle aller Datentypen kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Enum *enumPtr

In diesem Element werden alle Datentypen übergeben, die intern als Aufzählungstyp gehandhabt werden. Diese Konstanten fangen alle mit dem Prefix "DM_" an. Eine vollständige Tabelle aller verfügbaren Aufzählungskonstanten kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Method *methodPtr

In diesem Element werden Methoden übergeben. Eine vollständige Tabelle aller verfügbaren Methoden kann dem Manual "Objekte, Attribute und Methoden" entnommen werden.

DM_Index *indexPtr

In diesem Element werden jeweils zwei Werte für die betroffene Spalte und Zeile, also zwei Zahlen, übergeben.

DM_ID *idPtr

In diesem Element wird die Objekt-IDs von Dialog Manager Objekten übergeben.

DM_Value *valuePtr

In diesem Element werden Werte in einer DM_Value-Struktur übergeben.

Anmerkung

Werden mit Hilfe dieser Struktur Werte vom Dialog in die Anwendung geholt, so allokiert der Dialog Manager den notwendigen Speicher und gibt in später auch wieder frei. Werden jedoch mit Hilfe dieser Struktur Werte von der Anwendung im Dialog gesetzt, so muss die Anwendung die Vektoren in der geeigneten Größe allokieren und diesen Speicherplatz wieder freigeben.

3.3.5 Struktur DM_MultiValue

Diese Struktur dient dem effizienten Setzen und Abfragen von mehreren Attributen bei mehreren Objekt in einem einzigen Aufruf an den Dialog Manager. In dieser Struktur wird für jedes hinterlegte Attribut angegeben, bei welchem Objekt es gesetzt bzw. abgefragt werden soll.

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

Bedeutung der Elemente

DM_ID object

Dieses Element enthält das Objekt, bei dem das Attribut erfragt bzw. ersetzt werden soll. Dabei darf dieser Wert 0 sein; in diesem Fall wird der Identifikator des beim Funktionsaufruf angegebenen Objekts verwendet.

DM_Attribute attribute

Dieses Element enthält das Attribut, das erfragt bzw. ersetzt werden soll.

DM_Value index

Dieses Element enthält in geeigneter Form den Index des Attributs, das erfragt bzw. ersetzt werden soll.

DM_Value data

Dieses Element enthält den abgefragten bzw. den neuen Wert des Attributes.

DM_Options options

Dieses Element enthält die für dieses Attribut bei Setzen bzw. beim Abfragen geltenden Optionen (DMF_...).

3.3.6 Struktur DM_Content

Diese Struktur dient dem effizienten Setzen und Abfragen von kombinierten vektoriellen Attributen bei Tablefields und Listboxen. Mit Hilfe dieser Struktur können mit einem Zugriff für alle betroffenen Felder der Aktivierungszustand, die Selektierbarkeit, der eigentliche String und die zugehörigen Userdata gesetzt bzw. abgefragt werden.

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

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

Bedeutung der Elemente

DM_String string

In diesem Element wird der eigentlich anzuzeigende Inhalt angegeben.

DM_Boolean active

Dieses Element gibt an, ob der entsprechende Eintrag in dem Objekt bereits vorselektiert dargestellt werden soll (true) oder nicht (false).

DM_Boolean sensitive

Dieses Element gibt an, ob der entsprechende Eintrag vom Benutzer selektierbar ist oder nicht.

DM_UInt1 udtype

In diesem Element wird der Datentyp des Attributs userdata angegeben.

DM_Value udvalue

Dieses Element enthält den eigentlichen Wert des Attributes userdata.

3.4 Objekt-Callback-Struktur DM_CallBackArgs

Diese Struktur wird immer als Parameter an Callback-Funktionen übergeben.

```
typedef struct {
  DM_UInt4  evbits
  DM_ID     object;
  DM_ID     accel;
  DM_Int2   index;
  DM_UInt2  skeyno;
  DM_Int2   index2;
} DM_CallBackArgs;
```

Bedeutung der Elemente

DM_UInt4 evbits

In diesem Element wird die Eventmaske des auslösenden Ereignisses übergeben. Da jedem Ereignis genau ein Bit in dieser Maske gesetzt ist, können diese Ereignisse auch in kombinierter Form auftreten.

DM_ID object

In diesem Element wird dabei das Objekt übergeben, an das diese Funktion gebunden ist.

DM_ID accel

Dieses Element definiert den DM-Identifikator der gedrückten Taste.

DM_UInt2 index

In diesem Element wird die betroffene Zeile oder Zelle des Objekts übergeben, falls es sich bei dem Ereignis um ein mit Index versehenes Objekt handelt (z.B. Selektion in einer Listbox, einem Poptext).

DM_UInt2 skeyno

Dieses Element definiert die Nummer der Taste in X-Notation der gedrückten Taste.

DM_UInt2 index2

Dieser Eintrag der Struktur enthält den zweiten Index. Beim Tablefield gilt also:

- index gibt die Zeile an
- index2 gibt die Spalte an

Bei listenartigen Objekten wie etwa Listbox, Poptext, etc. ist der Eintrag index2 ungültig.Hier gilt also: index gibt den Nummer des Eintrages in der Liste an. index2 ist ungültig, Wert hat keine Bedeutung

3.5 Objekt-Nachlade-Struktur DM_ContentArgs

Diese Struktur wird immer als Parameter an Nachlade-Funktionen übergeben.

```
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;
```

Bedeutung der Elemente

DM_ID object

In diesem Element wird dabei das Objekt übergeben, an das diese Funktion gebunden ist.

DM_UInt2 reason

In diesem Element wird der Grund des Aufrufs angegeben. Hier ist zur Zeit nur der Wert CFR_ load möglich. Dieser Wert soll angeben, dass der Inhalt in das Tablefield nachgeladen werden soll.

DM_UInt2 visfirst

DM_UInt2 vislast

In diesen Elementen werden - in Abhängigkeit des Attributwerts von .direction - die erste und die letzte sichtbare Zeile oder Spalte angegeben: Ist das Attribut .direction mit 1 definiert, erfolgt das Nachladen zeilenweise; folglich beschreiben "visfirst" und "vislast" die erste und letzte sichtbare Zeile. Ist das Attribut .direction mit 2 definiert, erfolgt das Nachladen spaltenweise; folglich beschreiben "visfirst" und "vislast" die erste und letzte sichtbare Spalte.

DM_UInt2 loadfirst

DM_UInt2 loadlast

In diesen Elementen werden - in Abhängigkeit des Attributwerts von .direction - die erste und die letzte zu ladende Zeile oder Spalte angegeben: Ist das Attribut .direction mit 1 definiert, erfolgt das Nachladen zeilenweise; folglich beschreiben "loadfirst" und "loadlast" die erste und letzte zu ladende Zeile. Ist das Attribut .direction mit 2 definiert, erfolgt das Nachladen spaltenweise; folglich beschreiben "loadfirst" und "loadlast" die erste und letzte zu ladende Spalte.

DM_UInt2 count

In diesem Element wird die beim Tablefield definierte Zeilenanzahl = .rowcount (.direction = 1) bzw. Spaltenanzahl = .colcount (.direction = 2) übergeben.

DM_UInt2 header

In diesem Element wird die beim Tablefield definierte Überschriftenanzahl übergeben.

3.6 Datenfunktions-Struktur DM_DataArgs

Diese Struktur wird immer als Parameter an Datenfunktionen übergeben.

```
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;
```

Bedeutung der Elemente

DM_ID object

In diesem Element wird die Objekt-ID des Datenmodells an die Datenfunktion übergeben.

DM_Method task

In diesem Element wird der Grund des Aufrufs angegeben. Hier sind zur Zeit die folgende Werte möglich: *MT_get*, *MT_set* oder *MT_call*. Bei *MT_get* sollte die Datenfunktion für ein Modell-Attribut die nötigen Daten zurückliefern. Über *MT_set* werden Änderungen von der View an die Datenfunktion weitergeleitet. *MT_call* dient dazu, Daten-Aktionen zu implementieren, also Aufrufe welche zu einer „vielfältigen" Manipulation der Daten führen. Sinnvollerweise sollten Änderungen an Daten immer über **DM_DataChanged()** gemeldet werden.

DM_Attribute attribute

Für die Aufrufgründe *MT_get* und *MT_set* steht in diesem Element das Attribut auf das die Funktion anzuwenden ist.

DM_Attribute method

Für den Aufrufgrund *MT_call* steht in diesem Element die Methode welche die Datenfunktion ausführen soll. Diese wird durch einen Aufruf einer Daten-Aktion über die Methode `:calldata (<Method>, <Arg1>, ...<Arg15>)` ausgelöst.

DM_String identifier

In diesem Element steht der Attributbezeichner (aus dem *attribute*-Element) oder der Methodenbezeichner (aus dem *method*-Element) in String-Form um die Implementierung bei benutzerdefinierten Attributen zu erleichtern bzw. einen Modul-unabhängigen Vergleich von Attributen bzw. Methoden zu ermöglichen.

int argc

In diesem Element ist für den Aufrufgrund *MT_call* (Daten-Aktion die über **:calldata()** aufgerufen wird) die Anzahl der Argumente gesetzt.

DM_Value *argv

In diesem Element ist für den Aufrufgrund *MT_call* (Daten-Aktion die über **:calldata()** aufgerufen wird) die Argumentliste mit der Anzahl *argc* hinterlegt. Eine Rückgabe oder Veränderung von Argumenten ist damit nicht vorgesehen bzw. ohne Wirkung.

DM_Value index

In diesem Element steht der Indexwert des Attributzugriffs auf den sich der Aufrufgrund *MT_get* bzw. *MT_set* bezieht. Hat das *index*-Element den Datentyp *DT_void*, ist damit der Gesamtwert gemeint. Für indizierte Datenwerte sollten die Relationsarten und auch die Verarbeitung von unvollständigen Indexwerten (z.B. *[0]*, *[?,0]* oder *[0,?]*) beachtet werden.

DM_Value data

In diesem Element steht für den Aufrufgrund *MT_set* der Datenwert der von der Datenfunktion zu verarbeiten ist.

DM_Value retval

In diesem Element ist für den Aufrufgrund *MT_get* der Datenwert für das angegebene Attribut und den gesetzten Index zurückzuliefern.

3.7 ToolkitDaten-Struktur DM_ToolkitDataArgs

Diese Struktur kann als Daten-Parameter an die Funktion DM_GetToolkitDataEx übergeben werden. Sie dient dabei zur Kommunikation in beide Richungen und erlaubt die Übergabe und Rückgabe verschiedener Wertetypen. Als Markierung für einen gesetzten Wert dienen dabei mit logischem ODER verknüpfte DM_TKAM_*-Defines im *argmask*-Feld der Struktur.

Das „argmask“ Feld muss zwingend initialisiert werden bevor DM_GetToolkitDataEx aufgerufen wird.

Die DM_Rectangle-Struktur ist eine Hilfsstruktur für die Koordinaten und Größe einer rechteckigen „Objektform“.

```
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;
```

Bedeutung der Elemente

DM_UInt4 argmask

Ein-/Ausgabe: Muss vor dem Aufruf initialisiert werden. Wenn Felder als Eingabe verwendet werden, dann müssen die entsprechenden Bits gesetzt sein. DM_GetToolkitDataEx füllt je nach abgefragtem Attribut Felder dieser Struktur aus und fügt die entsprechenden Bits der „armask“ hinzu. In folgender Liste wird das zugehörige Bit in Klammern hinter dem Feld aufgeführt.

DM_Value index

Eingabe: Index einer Tabellenzelle

Bits: DM_TKAM_index

DM_Value data

Ausgabe: DM_ID und Datentyp eines IDM-Objekts.

Bits: DM_TKAM_widget, DM_TKAM_widget

DM_Rectangle rectangle

Ausgabe: Koordinaten und Größe einer rechteckigen „Objektform“ oder Breite und Höhe eines Bildes - je nach gesetzter argmask.

Bits: DM_TKAM_rectangle, DM_TKAM_tile

DM_UInt2 tile.gfxtype

Ausgabe: Wird auf den GFX Type gesetzt, der am besten passt.

Bits: DM_TKAM_tile

DM_UInt tile.dpi

Ausgabe: Der DPI Wert für den die vom IDM geladenen Bilder entworfen wurden.

Bits: DM_TKAM_tile

DM_UInt dpi

Ausgabe: Standard DPI Wert des Systems (meist 96). (MICROSOFT WINDOWS: siehe unten)

Bits: DM_TKAM_dpi, DM_TKAM_tile

int scale.dpi

Ausgabe: Der vom System entsprechend der eingestellten Skalierung gesetzte DPI-Wert. (MICROSOFT WINDOWS: siehe unten)

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

int scale.factor

Ausgabe: Systemskalierungfaktor (MICROSOFT WINDOWS: siehe unten)

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

DM_UInt2 tile_req

Eingabe: Gewünschter Datentyp bei Abfrage einer tile Ressource.

Bit: DM_TKAM_tile_reqtype

3.7.1 Spezifische Strukturelemente für Microsoft Windows

Die für das Fenstersystem MICROSOFT WINDOWS spezifischen Parameter sind wie folgt definiert:

HANDLE handle

Eingabe: Je nach abgefragtem Attribut kann hier ein Microsoft Windows Handle als zusätzlicher Eingabeparameter gesetzt werden.

Bits: DM_TKAM_handle

DM_UInt2 tile.gfxtype

Ausgabe: Wird auf den GFX-Type gesetzt, der zurückgeliefert wird. Mögliche Werte sind:

- 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

Ausgabe: Wird auf den "DMF_TlkData*"" Type gesetzt, der zurückgeliefert wird. Mögliche Werte sind:

- DMF_TlkDataIsIcon: Microsoft Windows Icon Handle in "tile.data"
- DMF_TlkDataIsWMF: Microsoft Windows Metafile Handle in "tile.data"
- DMF_TlkDataIsEMF: Microsoft Windows Enhanced Metafile Handle in "tile.data"
- DMF_TlkDataIsD2D1Bmp: Microsoft Direct2D Bitmap (ID2D1Bitmap *) in "tile.iunk"
- DMF_TlkDataIsD2D1SVG: 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

Ausgabe: Wird auf die Daten gesetzt, der genaue Typ ist abhängig von **tile.gfxtype** bzw. **tile.datatype**:

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

Bits: DM_TKAM_tile

HBITMAP tile.mask

Ausgabe: Wird auf die monochrome Maske gesetzt, falls eine vorhanden ist. Dies kann nur bei *DM_GFX_BMP* vorkommen.

Bits: DM_TKAM_tile

HPALETTE tile.palette

Ausgabe: Wird auf die Farbpalette gesetzt, falls eine vorhanden ist.

Bits: DM_TKAM_tile

LPUNKNOWN tile.iunk

Ausgabe: In diesem Element wird gegebenenfalls der MICROSOFT DIRECT2D Interface-Zeiger abgelegt.

Bits: DM_TKAM_tile

DM_UInt dpi

Ausgabe: Wird auf den Microsoft Windows Standard DPI Wert gesetzt. Dies ist der Wert, den Anwendungen die DPI Unaware sind verwenden. Dieser Wert wird von Microsoft Windows als

Konstante *USER_DEFAULT_SCREEN_DPI* (Wert: 96) definiert.

Diesen Wert verwendet der IDM auch für seine Pixelkoordinaten.

Bits: DM_TKAM_dpi, DM_TKAM_tile

int scale.dpi

Ausgabe: Wird auf den aktuellen DPI Wert des Objektes gesetzt. Wenn das Attribut AT_DPI abgefragt wird, ist der Rückgabewert von DM_GetToolkitDataEx und dieser Wert identisch. Dieser Wert ist beim IDM für Windows 11 dynamisch, wenn die Anwendung DPI Aware ist. Sonst, DPI Unaware oder IDM für Windows 10, ist der Wert fest auf 96 definiert.

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

int scale.factor

Ausgabe: Ganzzahliger Prozentwert, der aus **scale.dpi** und **dpi** berechnet wird.)

Bits: DM_TKAM_scaledpi, DM_TKAM_tile

DM_UInt2 tile_req

Eingabe: Dient dazu, um einen bestimmten Datentyp bei der Abfrage einer tile-Ressource zu erhalten.
Es stehen folgende Werte zur Verfügung:

- 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 *)

Diese Datentypen können mit "bitwise or" verknüpft angegeben werden. Der Wert DM_GFX_BMP ist als Fallback vorselektiert.

Bits: DM_TKAM_tile_req

3.7.2 Spezifische Strukturelemente für Qt

Für das Fenstersystem QT spezifischen Parameter sind wie folgt definiert:

DM_Pointer widget

Eingabe: Je nach abgefragtem Attribut kann hier ein QWidget als zusätzlicher Eingabeparameter gesetzt werden.

Bits: DM_TKAM_widget

DM_Pointer tile.pixmap

Ausgabe: Wird auf die QPixmap der Tile gesetzt. Datentyp in **tile.gfxtype** ist immer *DM_GFX_QPIXMAP*.

Bits: DM_TKAM_tile

3.7.3 Spezifische Strukturelemente für Motif

Für das Fenstersystem MOTIF spezifischen Parameter sind wie folgt definiert:

Widget/ DM_Pointer widget

Eingabe: Je nach abgefragtem Attribut kann hier ein XWidget als zusätzlicher Eingabeparameter gesetzt werden.

Bits: DM_TKAM_widget

XImage*/ DM_Pointer tile.ximage

Ausgabe: vollständige Bildinformationen als XImage, wenn **tile.gfxtype** = *DM_GFX_XIMAGE*

Bits: DM_TKAM_tile

Pixmap/DM_Pointer tile.pixmap

Ausgabe: Bildinformationen als Pixmap, wenn **tile.gfxtype** = *DM_GFX_PIXMAP*

Bits: DM_TKAM_tile

Pixmap/DM_Pointer tile.trans_mask

Ausgabe: die Transparenz-Clipmaske zugehörig zu den Bildinformationen in tile.pixmap, wenn **tile.gfxtype** = *DM_GFX_PIXMAP*

Bits: DM_TKAM_tile

3.8 Strukturen und Definitionen für die Funktionsanbindung

In den nachfolgenden Kapiteln werden alle vom Dialog Manager definierten Strukturen und deren Bedeutung vorgestellt. Diese Strukturen dienen der Kommunikation zwischen der Anwendung und dem Dialog Manager.

3.8.1 Definition DML_c, DML_pascal und DML_default

Mit Hilfe dieser Definitionen werden die Aufrufkonventionen der Anwendungsfunktionen auf den PC-basierten Systemen (Windows) definiert. Diese Definitionen sind dabei vom Fenstersystem und Compiler abhängig.

```
#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 Definition DM_ENTRY

Mit Hilfe dieser Definition können die Anwendungsfunktionen so definiert werden, dass die Definition auf alle vom Dialog Manager unterstützten Systemen ohne Änderung übernommen werden kann. Daher müssen alle vom Dialog Manager aufgerufenen Funktionen vom Typ DM_ENTRY sein.

Die Definition selber ist Plattform abhängig und sieht wie folgt aus:

```
#  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

Mit Hilfe dieser Definition können die Callback-Funktionen so definiert werden, dass die Definition auf alle vom Dialog Manager unterstützten Systemen ohne Änderung übernommen werden kann. Daher müssen alle vom Dialog Manager aufgerufenen Callback-Funktionen vom Typ DM_CALLBACK sein.

Die Definition selber ist Plattform abhängig und sieht wie folgt aus:

```
#  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

Mit Hilfe dieser Funktionen werden alle im Interface des Dialog Managers liegenden Funktionen definiert.

Die Definition selber ist Plattform abhängig und sieht wie folgt aus:

```
#  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

Diese Typdefinition dient dazu, dass unterschiedliche Funktionsarten in einer einzigen Funktionstabelle an den Dialog Manager übergeben werden können.

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

3.8.6 Struktur DM_FuncMap

Mit Hilfe dieser Struktur werden dem Dialog Manager die Adressen der im Dialog definierten und in der Anwendung realisierten Funktionen bekannt gemacht. Danach ist der Dialog Manager in der Lage, diese Anwendungsfunktionen aufzurufen.

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

Bedeutung der Elemente

DM_String symbol

In diesem Element wird der Name der Funktion, wie sie im Dialogskript definiert wurde, an den Dialog Manager übergeben.

DM_EntryFunc address

In diesem Element wird die Adresse der realen Funktion an den Dialog Manager übergeben.

3.9 Strukturen für Canvas-Funktionen

Die in den nachfolgenden Kapiteln vorgestellte Struktur dient als Parameter für alle Canvas-Funktionen. Diese Struktur sowie die dazu gehörende Funktion sind Fenstersystem abhängig.

3.9.1 Struktur DM_CanvasUserArgs für Microsoft Windows

Für das Fenstersystem Microsoft Windows sieht die Definition der Canvas-Struktur wie folgt aus:

```
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;

  DM_Pointer  userdata;
} DM_CanvasUserArgs;
```

Bedeutung der Elemente

DM_ID object

In diesem Element wird die DM-ID des Canvas-Objektes übergeben.

int reason

In diesem Element wird der Grund für den Aufruf der Canvas-Funktion übergeben. Dabei gibt es zur Zeit folgende Möglichkeiten:

Reason	Bedeutung
CCR_expose	Es liegt ein Redraw-Ereignis für die Canvas vor.
CCR_input	Der Benutzer hat eine Eingabe mit der Maus oder der Tastatur vorgenommen.
CCR_reschg	Eine Canvas-Ressource wurde verändert.
CCR_resize	Die Canvas wurde in ihrer Größe verändert.

Reason	Bedeutung
CCR_start	Die Canvas ist auf dem Bildschirm sichtbar gemacht worden.
CCR_stop	Die Canvas ist nicht mehr auf dem Bildschirm sichtbar.
CCR_winmsg	Eine Microsoft Windows-Nachricht ist eingetroffen. Bei dieser Task ist der Rückgabewert der Canvas-Funktion von entscheidender Bedeutung. Ein TRUE heißt, dass die Message verarbeitet wurde; "mresult" wird an Microsoft Windows zurückgegeben. Ein FALSE heißt, dass die Message nicht verarbeitet wurde, der Dialog Manager führt die Default-Aktion aus.

HWND hwnd

HandleID des Microsoft Windows Fensters.

UINT message
WPARAM wParam
LPARAM lParam
LRESULT mresult

Dies sind die originalen Message-Parameter von Microsoft Windows. Wenn keine Message vorhanden ist, hat das Element "message" den Wert 0. Die Elemente *wParam*, *lParam* und *mresult* sind nur gültig, wenn das Element "message" ungleich 0 ist.

int x

In diesem Element wird die X-Koordinate der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

int y

In diesem Element wird die Y-Koordinate der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

int width

In diesem Element wird die Breite der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

int height

In diesem Element wird die Höhe der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

HANDLE hdc

Dieses Element ist der **device-context-handle**, der zum Zeichnen in die Canvas benötigt wird.

DM_Pointer userdata

In diesem Element kann die Callback-Funktion beliebige anwendungsbezogene Daten speichern. Die gespeicherten Werte werden bei jedem Aufruf an die Canvas-Funktion mit übergeben. Für

weitere Erläuterungen kann das Manual "Objekte, Attribute und Methoden" für das Attribut "AT_CanvasData" herangezogen werden.

Anmerkung

Die Belegung dieser Struktur ist immer abhängig von der aktuellen Task. Die Elemente *object*, *reason* und *hwnd* sind immer belegt, die Belegung der anderen Strukturelemente kann der nachfolgenden Tabelle entnommen werden:

Task	Belegtes Element	Bedeutung
CCR_expose	x, y, width, height	Dimension des Update-Rechtecks
	hdc	Device-Context-Handle zum Zeichnen in die Canvas.
CCR_input	x, y	Diese Felder sind belegt, falls das ursprüngliche Ereignis ein "ButtonPress"-Ereignis gewesen ist.
CCR_reschange	x, y	Koordinate der linken oberen Ecke der Nutzfläche der Canvas bezüglich des eigentlichen Canvas-Fensters.
	width, height	Dimension der Canvas-Nutzfläche.
CCR_resize	x, y	Koordinate der linken oberen Ecke der Nutzfläche bezüglich des Canvas-Fensters.
	width, height	Dimension der Canvas-Nutzfläche.
CCR_start	x, y	Koordinate der linken oberen Ecke der Nutzfläche der Canvas bezüglich des eigentlichen Canvas-Fensters.
	width, height	Dimension der Canvas-Nutzfläche.
CCR_winmsg	message, wParam, lParam, mresult	In diesen Elementen werden die original Windows Messages weitergereicht. Die möglichen Werte und Bedeutung dieser Elemente können dem Microsoft Windows Manual entnommen werden.

3.9.2 Struktur DM_CanvasUserArgs für Motif

Für das Fenstersystem Motif sieht die Definition der Canvas-Struktur wie folgt aus:

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

#ifdef _XLIB_H_
  Xevent      *xevent;
  Window      window;
```

```
#else
  DM_Pointer  xevent;
  long        window;
#endif

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

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

  DM_Pointer  userdata;
} DM_CanvasUserArgs;
```

Bedeutung der Elemente

DM_ID object

In diesem Element wird die DM-ID des Objekts übergeben, das diesen Funktionsablauf ausgelöst hat, also die Canvas, zu der die aufgerufene Funktion gehört.

int reason

In diesem Element teilt der DM der Anwendung mit, warum die Canvas-Funktion aufgerufen wurde.

Dabei gibt es folgende als Konstanten definierte Möglichkeiten:

Reason	Bedeutung
CCR_expose	Es liegt ein Redraw-Ereignis für die Canvas vor.
CCR_focus_in	Die Canvas hat den Eingabefokus bekommen.
CCR_focus_out	Die Canvas hat den Fokus verloren.
CCR_input	Der Benutzer hat eine Eingabe mit der Maus oder der Tastatur vorgenommen.
CCR_reschg	Eine Canvas-Ressource wurde verändert.
CCR_resize	Die Canvas wurde in ihrer Größe verändert.
CCR_start	Die Canvas ist auf dem Bildschirm sichtbar gemacht worden.

Reason	Bedeutung
CCR_stop	Die Canvas ist nicht mehr auf dem Bildschirm sichtbar.
CCR_xevent	Es liegt ein beliebiges anderes X-Event für die Canvas vor.

Xevent *xevent

In diesem Parameter wird das Orginal-X-Event an die Anwendung übergeben. Falls kein X-Event vorhanden ist, ist dieser Parameter ein *NULL*-Pointer. Diese Ereignisse werden in den entsprechenden X-Manuals erklärt.

Window window

In diesem Element wird das zugehörige X-Windows-Fenster an die Anwendung übergeben. Für eine Erklärung sei hier auf die entsprechenden X-Windows-Manuals hingewiesen.

Widget widget

In diesem Element wird die Xt-Widget Information dem Objekt übergeben.

DM_Int2 x

In diesem Element wird die X-Koordinate der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

DM_Int2 y

In diesem Element wird die Y-Koordinate der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

DM_UInt2 width

In diesem Element wird die Breite der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

DM_UInt2 height

In diesem Element wird die Höhe der Nutzfläche der Canvas bzgl. des Canvas-Fensters übergeben.

DM_Pointer userdata

In diesem Element kann die Callback-Funktion beliebige anwendungsbezogene Daten speichern. Die gespeicherten Werte werden bei jedem Aufruf an die CanvasFunktion mit übergeben. Für weitere Erläuterungen kann das Manual "Objekte, Attribute und Methoden" für das Attribut "AT_CanvasData" herangezogen werden.

Anmerkung

Die Belegung dieser Struktur ist immer abhängig von der aktuellen Task. Die Elemente "object" und "reason" sind immer belegt, die Elemente "window", "widget" und "xevent" sind soweit verfügbar immer belegt, die Belegung der anderen Strukturelemente kann der nachfolgenden Tabelle entnommen werden:

Task	Belegtes Element	Bedeutung
CCR_expose	x, y, width, height	Dimension des Update-Rechtecks
CCR_input	x, y	Diese Felder sind belegt, falls das ursprüngliche Ereignis ein "ButtonPress"-Ereignis gewesen ist.
CCR_reschange	x, y	Koordinate der linken oberen Ecke der Nutzfläche der Canvas bezüglich des eigentlichen Canvas-Fensters.
	width, height	Dimension der Canvas-Nutzfläche.
CCR_resize	x, y	Koordinate der linken oberen Ecke der Nutzfläche bezüglich des Canvas-Fensters, falls die Werte schon verfügbar sind, sonst 0.
	width, height	Dimension der Canvas-Nutzfläche, falls die Werte schon verfügbar sind sonst 0.
CCR_start	x, y	Koordinate der linken oberen Ecke der Nutzfläche der Canvas bezüglich des eigentlichen Canvas-Fensters.
	width, height	Dimension der Canvas-Nutzfläche.

3.10 Strukturen für Input-Handler-Funktionen

Mit Hilfe von sogenannten Input-Handler ist die Anwendung in der Lage, auf nicht vom Dialog Manager bereitgestellte Ereignisse zu reagieren. Diese Reaktion erfolgt dadurch, dass dem Dialog Manager von der Anwendung definierte Input-Handler Funktionen installiert werden, die dann zu gegebener Zeit vom Dialog Manager aufgerufen werden. Die Strukturen sind in Fenstersystem abhängig.

3.10.1 Struktur DM_InputHandlerArgs für Microsoft Windows

Für das Fenstersystem Microsoft Windows sieht die Definition der Input-Handler-Struktur wie folgt aus:

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

Bedeutung der Elemente

HWND hwnd

In diesem Element wird der Windows-Handle zu dem Objekt übergeben, an das die Message gesendet worden ist.

UINT message

In diesem Element wird die Message übergeben, die an das Objekt geschickt worden ist. Die möglichen Belegungen entnehmen Sie bitte den entsprechenden Handbüchern von Microsoft Windows.

WPARAM wParam

In diesem Element werden Informationen zu der Message übergeben, wie sie vom Fenstersystem erhalten worden sind.

LPARAM lParam

In diesem Element werden Informationen zu der Message übergeben, wie sie vom Fenstersystem erhalten worden sind.

LRESULT mresult

In diesem Element wird der Rückgabewert der Input-Handler-Funktion übergeben. Dieses Element wird nur bei einem speziellen Aufruf-Modus ausgewertet.

DM_Pointer userdata

In diesem Element wird der Zeiger übergeben, der beim Installieren des Input-Handler angegeben worden ist. Seine Bedeutung ist anwendungsspezifisch, d.h. hier kann die Anwendung beliebige von ihr benötigte Informationen hinterlegen.

3.10.2 Struktur DM_InputHandlerArgs für Motif

Für dieses System ist keine Struktur definiert, da hier keine fenstersystem-spezifische Information an die Anwendung übergeben werden muss.

3.11 Strukturen und Definitionen für die Formatierung von Eingaben

Für die Formatierung von Eingabe in Eingabefelder sind eine Reihe von Strukturen definiert, die in den nachfolgenden Kapiteln vorgestellt werden. Diese Strukturen sind alle direkt oder indirekt Parameter einer in der Anwendung definierten Formatfunktion.

3.11.1 Definitionen für die Formatierung

Die verschiedenen Aufgaben der Formatfunktion sind über Definitionen in der Include-Datei **IDMuser.h** verfügbar. Im einzelnen bedeuten die einzelnen Aufgaben folgendes:

Task-Definition	Aufgabe der Task
FMTK_parseformat	Bei dieser Task soll der angegebene Formatstring geparst und die entsprechenden Strukturen aufgebaut werden.
FMTK_cleanformat	Bei dieser Task sollen alle zu dem Format gehörenden allokierten Strukturen freigegeben werden.
FMTK_create	Diese Task soll die Formatfunktion informieren, dass das Format bei einem gültigen Objekt verwendet wird. Falls es noch notwendig ist, sollen jetzt endgültig die zu dem Format gehörenden Strukturen allokiert und initialisiert werden.
FMTK_destroy	Das zum Format gehörende Objekt wird zerstört. Daher können auch die formatspezifischen Strukturen zum Format wieder freigegeben werden.
FMTK_setcontent	Diese Task soll die Formatfunktion informieren, dass im Objekt per Programm ein neuer Inhalt gesetzt worden ist. Die Formatfunktion muss daher den Inhalt gegen das Format prüfen und die entsprechenden Formatinformationen korrigieren.
FMTK_setselection	Die Formatfunktion soll die neue Positionen des Cursors und des Anfangs des Selektionsbereichs bezüglich des Inhaltsstrings setzen.
FMTK_setmaxchars	Diese Task informiert die Formatfunktion, dass dem Objekt das Attribut *.maxchars* gesetzt worden ist. Die Funktion muss die dazu gehörenden Aktionen durchführen.
FMTK_formatcontent	Diese Task informiert die Fomatfunktion dahin, dass der Inhalt formatiert werden soll, damit der Inhalt im Fenstersystem gesetzt werden kann.
FMTK_enter	Diese Task wird aufgerufen, wenn das zu diesem Format gehörende Eingabefeld den Eingabefocus erhalten hat. Die Formatfunktion kann dann also die beim Betreten des Eingabefeldes notwendigen Aktionen durchführen.
FMTK_leave	Diese Task wird aufgerufen, wenn das zu diesem Format gehörende Eingabefeld den Eingabefokus verloren hat. Die Formatfunktion kann dann also die beim Verlassen des Eingabefeldes notwendigen Aktionen durchführen.

Task-Definition	Aufgabe der Task
FMTK_modify	Diese Task wird aufgerufen, wenn der Benutzer eine Eingabe in den Text vorgenommen hat. Die Funktion muss die Eingabe verarbeiten und die entsprechende Aktion im dargestellten Text durchführen.
FMTK_keynavigate	Diese Task wird aufgerufen, wenn der Benutzer eine Cursorbewegung im Text vorgenommen hat. Die Funktion muss die Navigation verarbeiten und die entsprechende Aktion im dargestellten Text durchführen.
FMTK_setcursorabs	Diese Task wird aufgerufen, wenn der Benutzer den Cursor absolut im dargestellten Text positioniert hat. Die Funktion muss dann die entsprechenden Aktionen auf den internen Strukturen durchführen.

3.11.2 Struktur DM_FmtContent

Mit Hilfe dieser Struktur wird der eigentliche Inhalt gespeichert. Diese Struktur muss also bei den entsprechenden Aufrufen an die Formatfunktion mitaktualisiert werden.

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

Bedeutung der Elemente

DM_String string

In diesem Element ist der aktuelle Inhalt gespeichert.

DM_UInt2 size

In diesem Element wird die Größe des in dem Element *string* allokierten Speichers abgelegt.

DM_UInt2 length

In diesem Element wird die aktuelle Länge des Inhaltstrings abgelegt.

DM_UInt2 curpos

In diesem Element wird die Cursorposition im Inhaltstring abgelegt.

DM_UInt2 selpos

In diesem Element wird die Selektionsposition im Inhaltstring abgelegt.

DM_UInt2 maxchars

In diesem Element wird die maximale Länge des Inhaltstrings abgelegt.

3.11.3 Struktur DM_FmtRequest

Mit Hilfe dieser Struktur wird der eigentliche Datenaustausch zwischen Dialog Manager und Formatfunktion vollzogen. Die in der Struktur enthaltene Union ist je nach Aufgabe belegt.

```
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;
```

Bedeutung der Elemente

DM_UInt1 task

In diesem Element wird die Aufgabe abgelegt, die von der Formatfunktion erfüllt werden soll. Der hier gespeicherte Wert steuert die Belegung der nachfolgenden Union.

Struktur parseformat bei Task FMTK_parseformat

DM_UInt1 codepage

In diesem Element wird die Codepage übergeben, in der die Ausgabe des formatierten Strings erfolgen soll. Die dazu notwendigen Konstanten sind in der Include-Datei "IDMuser.h" definiert und beginnen alle mit dem Prefix "CP_".

DM_String formatstr

In diesem Element wird der eigentliche Formatstring übergeben.

Struktur setselection bei Task FMTK_setcontent

DM_UInt1 codepage

In diesem Element wird die Codepage übergeben, in der die Ausgabe des formatierten Strings erfolgen soll. Die dazu notwendigen Konstanten sind in der Include-Datei "IDMuser.h" definiert und beginnen alle mit dem Prefix "CP_".

DM_String string

In diesem Element wird der neue Inhalt des Objektes übergeben, der formatiert werden soll.

Struktur setselection bei Task FMTK_setselection

DM_UInt2 contcurpos

In diesem Element wird die aktuelle Cursorposition bezogen auf den Inhalt übergeben.

DM_UInt2 contselpos

In diesem Element wird die aktuelle Selektionsposition bezogen auf den realen Inhalt übergeben.

Struktur setmaxchars bei Task FMTK_setmaxchars

DM_UInt2 maxchars

In diesem Element wird die neue maximale Zeichenanzahl übergeben.

Struktur modify bei Task FMTK_modify

DM_String string

In diesem Element wird der String ohne terminierendes Null-Byte übergeben, der an der aktuellen Cursorposition in den vorhandenen Inhalt eingefügt werden soll.

DM_UInt2 strlength

In diesem Element wird die Länge des einzufügenden Strings übergeben.

DM_UInt2 dpycurpos

In diesem Element wird die aktuelle Cursorposition im angezeigten String übergeben. An dieser Stelle soll der angegebene String eingefügt werden.

DM_UInt2 dpyselpos

In diesem Element wird die aktuelle Selektionsposition im angezeigten String übergeben. Ist dieser Wert gleich dem Wert in dpycurpos, so wird der angegebene String an der Position eingefügt; ist dieser Wert ungleich dem Wert in dpycurpos, so wird der durch dpycurpos und dpyselpos eingeschlossene String durch den angegebenen String ersetzt.

Struktur keynavigate bei Task FMTK_keynavigate

DM_Int2 yoffset

In diesem Element wird die Navigation in Y-Richtung angegeben. Ein positiver Wert hierbei bedeutet, eine Navigation nach unten, ein negativer Wert eine Navigation nach oben.

DM_Int2 xoffset

In diesem Element wird die Navigation in X-Richtung angegeben. Ein positiver Wert hierbei bedeutet, eine Navigation nach rechts, ein negativer Wert eine Navigation nach links.

DM_Boolean movecur

In diesem Element wird hinterlegt, ob durch die Cursorbewegung die Cursorposition verändert werden soll (TRUE) oder nicht (FALSE).

DM_Boolean movesel

In diesem Element wird hinterlegt, ob durch die Cursorbewegung die Selektionsposition verändert werden soll (TRUE) oder nicht (FALSE).

Struktur setcursorabs bei Task FMTK_setcursorabs

DM_UInt2 dpycurpos

In diesem Element wird die neue absolute Cursorposition bezogen auf den angezeigten String übergeben.

DM_UInt2 dpyselpos

In diesem Element wird die neue absolute Selektionsposition bezogen auf den angezeigten String übergeben.

3.11.4 Struktur DM_FmtFormat

Diese Struktur enthält Daten, die nur der Dialog Manager internen Format-Funktion zugänglich sind und die die internen Informationen zum aktuellen Format - unabhängig vom Inhaltsstring darstellen.

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

Bedeutung der Elemente

DM_UInt2 maxchars

In diesem Element wird die durch den Formatstring festgelegte Maximallänge des Inhaltstrings angegeben.

char secretChar

In diesem Element wird das bei der verdeckten Formatierung benutzte Zeichen abgespeichert.

3.11.5 Struktur DM_FmtDisplay

Diese Struktur wird zum Anzeigen der formatierten Strings benötigt. Aus dieser Struktur bezieht das Fenstersystem die Information, welcher Text angezeigt werden soll. Zusätzlich wird in dieser Struktur hinterlegt, an welcher Stelle im String der Cursor stehen soll und an welcher Stelle die Selektionsmarke sitzen soll.

```
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;
```

Bedeutung der Elemente

DM_Boolean overwritemode

In diesem Element wird angegeben, ob das Fenstersystem im Overwrite- oder Insert-Modus ist.

DM_UInt1 codepage

In diesem Element wird die Codepage des Darstellungsstring angegeben.

DM_UInt2 curpos

In diesem Element wird die aktuelle Cursorposition im Darstellungsstring abgespeichert.

DM_UInt2 selpos

In diesem Element wird die aktuelle Selektionsposition im Darstellungsstring abgespeichert.

DM_UInt2 length

In diesem Element wird die Länge des aktuellen Darstellungsstrings abgespeichert.

DM_UInt2 size

In diesem Element wird die Länge des allokierten Speichers des Elements "string" abgespeichert.

DM_String string

In diesem Element wird der aktuelle Darstellungsstring abgespeichert.

3.12 Abbildung der DM-Datentypen

Die Abbildung, der im Dialogskript verwendeten Datentypen erfolgt nach einem festen Schema.

Sind diese Datentypen nur als **Input-Parameter** einer Funktion deklariert, so müssen sie entsprechend der folgenden Tabelle in der C-Funktion deklariert werden.

DM-Datentyp	C-Datentyp
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

Werden Sie aber auch bzw. nur als **Output-Parameter** einer Funktion deklariert, so müssen diese in der C-Funktion in der Regel als Pointer auf diese Datenstrukturen deklariert werden.

DM-Datentyp	C-Datentyp
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 *

Beispiel

Dialogskript:

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

void function2(boolean input, integer output);

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

C-Deklaration:

```
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
)
```

Bei der Deklaration der Parameter einer Funktion ist folgendes zu beachten:

Wird ein Parameter nur als Output-Parameter deklariert, wird er vom DM nicht mit sinnvollen Werten vorbelegt, dann sollte in der C-Funktion nicht lesend auf diesen Parameter zugegriffen werden. Ist er sowohl für Input als auch für Output deklariert, wird er vom DM mit dem entsprechenden Wert vorbelegt.

Bei der Verwendung von Output-Parametern können durch einen Funktionsaufruf mehrere Elemente (z.B. Objektattribute) verändert werden.

4 Sammlungen und Stringbehandlung

Sammlungen können als Argumente oder Rückgabewerte von C-Funktionen oder für Datenmodell-Callbacks genutzt werden. Als Rückgabetyp für C-Funktionen ist auch der Datentyp *index* (*DT_index*) erlaubt.

Grundsätzlich dürfen **DM_Value**-Strukturen, die Sammlungen beinhalten können bzw. sollen, **nur** über spezielle DM-Funktionen manipuliert und erfragt werden, da der IDM hierfür die Speicherallokierung und Verwaltung von Strings durchführt.

Siehe auch

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

Um den Umgang mit der **DM_Value**-Struktur sowie mit Strings (*DM_String*) in C-Funktionen zu erleichtern wurde das folgende Konzept eingeführt.

Die C-Schnittstelle des IDM kann Wertereferenzen (*DM_Value** und *DM_String**) nach ihrer Verwendung unterscheiden bzw. für eine spezielle Verwendung initialisieren:

1. Wertereferenz ist ein Argument des Funktionsaufrufs und noch ungemanagt.
2. Wertereferenz ist ein statischer bzw. globaler Wert und wird durch den IDM gemanagt.
3. Wertereferenz ist ein lokaler Wert der vom IDM gemanagt wird. Die Rückgabe an den IDM ist erlaubt. Nach Beendigung des Funktionsaufrufs wird er freigeben.
4. Wertereferenz verweist auf einen vom IDM gemanagten Wert der ungültig ist, zum Beispiel weil er fehlerhaft durch den Anwender manipuliert wurde.
5. Wertereferenz verweist auf ungemanagten Wert (quasi wie bisher).

Dieses Konzept wird von den Funktionen der Art **DM_Value*()** und **DM_*Return()** unterstützt. Es ermöglicht die Manipulation von Werten (z.B. Änderung eines Strings, Hinzufügen von Werten zu einer Liste bzw. einem Hash), den Zugriff auf Elemente bzw. Indizes von Wertelisten sowie die Rückgabe von Wertelisten oder Strings ohne dass man sich explizit um die Verwaltung und Allokierung kümmern muss.

Regeln für die Verwendung

Für Werte und **DM_Value**-Strukturen der Art 1–3 muss Folgendes berücksichtigt werden:

- Eine direkte Manipulation der Werte in der Struktur muss unterbleiben. Lediglich der lesende Zugriff auf skalare Werte ist erlaubt.
- Das direkte Kopieren dieser Struktur in einen anderen Speicherbereich oder an eine andere Adresse sowie die Verwendung einer solchen Kopie sind verboten! Hierfür kann keine korrekte Verwaltung und Zugriff gewährleistet werden!

- Erlaubt ist lediglich das Kopieren eines gemanagten Wertes in einen noch ungemanagten Rückgabeparameter.
- Eine einmal gesetzte lokale oder statische Verwendung kann nicht mehr geändert werden.
- Die Rückgabe eines lokalen Wertes, Strings oder Index sollte über die Funktionen **DM_ValueReturn()**, **DM_StringReturn()** und **DM_IndexReturn()** erfolgen, sodass vom IDM gewährleistet werden kann, dass auch von einer lokalen Variablen der Wert korrekt zurückgegeben wird. Bei statischen Werten und Strings ist dies nicht nötig.

Da die vom IDM gemanagten Werte auch Strings beinhalten können, ist zu beachten, dass diese Strings in der Applikations-Codepage gehalten und zurückgeliefert werden. Eine Änderung dieser Codepage durch einen **DM_Control()**-Aufruf führt nicht zu einer Codepage-Änderung der gemanagten Strings. Die Applikations-Codepage sollte also so früh wie möglich korrekt gesetzt werden.

Die Nutzung der Funktionen **DM_ValueInit()**, **DM_ValueChange()**, **DM_ValueGet()**, **DM_ValueCount()**, **DM_ValueIndex()**, **DM_ValueReturn()**, **DM_StringInit()**, **DM_StringChange()**, **DM_StringReturn()** und **DM_IndexReturn()** ist nur für lokale und verteilte (DDM) C-Funktionen möglich. Eine Nutzung in Formatfunktionen ist nicht erlaubt.

Eine fehlerhafte Manipulation eines vom IDM gemanagten Wertes oder einer Zustandsänderung durch den Anwender wird in der C-Schnittstelle, wenn erkannt, durch die Fehlercodes *„DME_InvalidContext“* oder *„DME_BadContext“* gemeldet. Typischerweise sollten auch Ein- und Ausgabeparameter nicht auf die gleiche Wertereferenz zeigen, was wiederum mit einem Fehler `DME_InvalidArg` angezeigt wird.

Lokal gemanagte Wertereferenzen werden nach dem zugehörigen Funktionsaufruf aufgeräumt und falls nötig Unterstrukturen und Strings freigegeben.

Für besonders zeitkritische Funktionen sollte unter Umständen von der Nutzung gemanagter Werte und Strings Abstand genommen bzw. das Zeitverhalten geprüft werden, da deren Nutzung auf jeden Fall einen gesteigerten Verwaltungsaufwand im IDM bedeutet.

5 Anbindung des C-Programms

5.1 Hauptprogramm

In diesem Kapitel wird beschrieben, wie die im Dialog Manager definierten Funktionen in C realisiert und an den Dialog Manager angebunden werden müssen.

Prinzipiell hat jedes C-Programm, das in der nicht verteilten Lösung den Namen "AppMain" besitzt, in der verteilten Lösung den Namen "AppInit".

Dieses Hauptprogramm wird vom Dialog Manager aufgerufen und kann dann die notwendigen Initialisierungsschritte durchführen.

5.1.1 Lokale C-Programme

In der lokalen Lösung des Dialog Managers wird das Programm "AppMain" als das eigentliche Programm betrachtet. Es muss dann folgende Schritte durchführen:

- Initialisierung des Dialog Managers (DM_Initialize)
- Laden eines Dialoges (DM_LoadDialog)
- Initialisieren der Anwendung
- Übergeben der Funktionsadressen (DM_BindFunctions)
- Einlesen des Benutzerprofiles (DM_LoadProfile)
- Starten des Dialoges (DM_StartDialog)
- Starten der Verarbeitung (DM_EventLoop)

Nach der erfolgreichen Ausführung obiger Schritte kann der Anwender mit dem Programm arbeiten.

5.1.2 Verteilte C-Programme

In der verteilten Lösung des Dialog Managers muss das C-Hauptprogramm "AppInit" folgende Schritte durchführen:

- Übergabe der Funktionsadressen (DM_BindFunctions)
- Initialisierung der Anwendung

5.2 Normale Anwendungsfunktionen

Da vor allem auf den PC-basierten Systemen die Art der Parameter und die Art des Funktionsaufrufes entscheidend sind für die erfolgreiche Ausführung des Programms, sollte man unbedingt die Fähigkeit des Dialog Managers ausnutzen, Prototypen aus den im Dialog definierten Funktionen zu generieren. Wenn man die erzeugten Prototypen in Form einer Include-Datei in sein

Programm einbaut, so überprüft der C-Compiler, ob die Realisierung der C-Funktion mit der Deklaration übereinstimmt. Ist dies nicht der Fall, so gibt der C-Compiler je nach Schwere des Fehlers einen Fehler oder eine Warnung beim Übersetzen aus. Auf diese Art und Weise können solche fatalen Fehler frühzeitig erkannt werden. Neben den Prototypen erstellt diese Option auch eine C-Datei, die eine Funktion zum Anbinden der Funktionen an den Dialog enthält, so dass die Funktionstabelle nicht separat editiert werden muss.

Die Option zum Generieren der Prototypen und C-Datei mit Hilfe des Simulationsprogramms ist:

+writefuncmap

Damit ergibt sich folgende Kommandozeile für den Simulator:

```
idm +writefuncmap <Basis-Name der Datei> <Name der Dialogdatei>
```

Der Namen der generierten C-Funktion wird dabei nach folgendem Schema gebildet:

```
BindFunctions_<Name des Dialogs, des Moduls oder der Applikation>
```

Beispiel

Aus einer Dialogdefinition, die folgende Funktionen definiert hat, sollen die notwendigen Funktionsprototypen generiert werden.

Eingegebenes Kommando:

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

Dabei hat das Dialogskript list.dlg folgendes Aussehen:

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

Die generierte C-Datei sieht wie folgt aus:

```
#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))
}
```

Die generierte H-Datei sieht wie folgt aus:

```
#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_ */
```

Wenn nur eine Datei mit Funktionsprototypen generiert werden soll, kann auch die Option

+writeproto <Name der Include-Datei>

verwendet werden.

Damit ergibt sich folgende Kommandozeile für den Simulator:

```
idm +writeproto <Name der Include-Datei> <Name der Dialogdatei>
```

5.3 Funktionen mit Records als Parametern

Um Funktionen für Dialoge mit Records mit einer Applikation versehen zu können, müssen vom Dialog Manager generierte C-Module übersetzt und dazu gelinkt werden. Diese Funktionen dürfen auf gar keinen Fall in einer Funktionstabelle auftauchen, die in der Anwendung definiert ist, da diese Funktionen über das generierte Modul hinzugebunden werden. Diese Generierung ist für jedes im Dialog vorhandene Modul notwendig, das Definitionen für Funktionen beinhaltet. Die Generierung dieser Module erfolgt durch Aufruf der Simulation mit der Option **+writetrampolin**:

```
idm +writetrampolin <outfile> <dialogskript>
```

Dieses Statement generiert aus einem Dialogskript die notwendigen Module zum Aufruf von Funktionen. Dabei kann das angegebene Dialogskript ein Modul sein. Je nach Art der Funktionen, die solche Records verwenden, werden die entsprechenden Header-Dateien (C und/oder COBOL) erzeugt.

Bei der Implementierung dieser Funktionen muss in das entsprechende Modul die erzeugte Header-Datei eingebunden werden. Dies geschieht durch das Statement:

```
include
```

Wenn solche Funktionen und Records in einer Applikation verwendet werden, müssen diese Records durch den Aufruf der Funktion

```
RecInit<Dialogname>
```

oder

```
RecMInit<Modulname>
```

initialisiert werden.

Werden solche Strukturen in Dialog Manager-Applikationen verwendet, wird also der verteilte Dialog Manager eingesetzt, muss noch zusätzlich die Applikation angegeben werden, für die die Dateien generiert werden sollen.

Der Name der Initialisierungsfunktion wird dann aus dem Namen der Applikation gebildet.

Also muss

```
RecMInit<ApplicationName>
```

aufgerufen werden.

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

Beispiel

Aus dem 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;
}
```

wird folgende Include-Datei generiert:

```
#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 Dynamische Anbindung von Record-Funktionen

Verfügbarkeit

Ab IDM-Version A.06.01.d

Mit „dynamischer Anbindung“ ist die Bindungsart von Anwendungsfunktionen gemeint, welche kein explizites Setzen des Funktionspointers der Anwendungsfunktionen über die Funktionen DM_BindFunctions oder DM_BindCallBacks benötigt. Dazu gehören z.B. die Anbindung von Funktionen aus dynamischen Bibliotheken (Transport *"dynlib"*) oder der Aufruf von COBOL-Funktionen über ihren Namen durch die „BindThruLoader-Funktionalität“. Da allerdings Anwendungsfunktionen mit **Record-Parametern** eine zusätzliche Stub-Funktion für das Umwandeln (Marshalling) der Record-Struktur benötigen, wurde bisher beim Aufruf dieser Record-Funktionen keine Struktur übertragen.

Ab IDM-Version A.06.01.d wird für dynamisch angebundene Record-Funktionen die Struktur der Record-Parameter ohne Stub-Funktion übertragen. Eine Nutzung solcher Record-Funktionen ist möglich, ohne C- oder COBOL-Code zu generieren. Es ist aber sinnvoll sich die Funktionsprototypen wie auch die Strukturdefinition über **+writeheader <basefilename>** generieren zu lassen.

Beispiel

```
dialog D
record RecAdr {
  string[80] Name := "";
  string[120] Strasse := "";
  string[40] Ort := "";
  integer PLZ := 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 "Applikation(en) nicht korrekt angebunden!";
  elseif not Search("Udo", RecAdr) then
    print "Udo nicht gefunden, f\374ge ihn hinzu";
    RecAdr.Name := "Udo";
    RecAdr.Strasse := "Meisenweg 7";
    RecAdr.Ort := "Wolkenstein";
    RecAdr.PLZ := 71723;
    New(RecAdr);
  else
    print "Udo gefunden, wohnt in: "+RecAdr.Ort;
  endif
  exit(();
}
```

Die Generierung der Prototypen kann nun anstatt mit **+writeproto**, **+writefuncmap** und **+/-writetrampolin** einfach mit **+writeheader** erfolgen:

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

Die eigentliche Implementierung in C und COBOL kann nun die Funktionsprototypen bzw. Record-Definitionen für „RecRecAdr“ in **adr.h** (C-Headerdatei) und **adr.cpy** (COBOL Copy-Strecke) finden.

5.3.2 Hinweis bei Verwendung von DM-Funktionen

Innerhalb einer C-Funktion, welche einen ***Record*** als Parameter verwendet, muss diese DM-Funktion immer mit der Option *DMF_DontFreeLastStrings* aufgerufen werden, da ansonsten im ***Record*** enthaltenen Strings ebenfalls freigegeben werden; diese Strings also nicht mehr zur Verfügung stehen würden.

Siehe auch

Kapitel „Behandlung von String-Parametern“ im Handbuch „C-Schnittstelle - Funktionen“ und die jeweiligen Funktionsbeschreibungen.

5.4 Objektcallback-Funktionen

Funktionen, die in der Dialogbeschreibung als Objektcallback-Funktionen deklariert sind, müssen in C wie folgt deklariert werden:

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

wenn die Funktion im Dialogskript als

```
{export | reexport } function callback Funktionsname() for Ereignisse;
```

definiert ist, oder als

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

wenn die Funktion im Dialogskript als

```
{export | reexport} function c callback Funktionsname() for Ereignisse;
```

definiert ist.

Der Parameter dieser Funktion ist fest vom Dialog Manager vorgegeben und kann nicht verändert werden. Diese Funktion wird immer dann aufgerufen, wenn ein bei der Funktion angegebenes Ereignis beim entsprechenden Objekt eingetreten ist.

Die Rückgabe des Wertes TRUE bedeutet dabei, dass nach dem Aufruf dieser Funktion die normale Regelbearbeitung ausgeführt werden soll; die Rückgabe des Wertes FALSE unterbindet eine solche Regelausführung.

Beispiel

Die Definition einer Callback-Funktion im Dialog-Skript sieht wie folgt aus:

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

Die Zuordnung zu einem Objekt erfolgt bei dem betroffenen Objekt:

```
child edittext File
{
  .xleft 14;
  .ytop 0;
  .width 20;
  .function CheckFilename;
```

```
  .content "list.dlg";
}
```

Die nachfolgende Funktion soll bei erfolgter Eingabe überprüfen, ob die Eingabe des Benutzers einen gültigen Dateinamen darstellt.

```
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 can not 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 Nachlade-Funktionen

Funktionen, die in der Dialogbeschreibung als Nachlade-Funktionen deklariert sind, müssen in C wie folgt deklariert werden:

```
DM_Boolean DML_default DM_CALLBACK Funktionsname
(
  DM_ContentArgs *data
)
```

wenn die Funktion als

```
{export | reexport} function contentfunc Funktionsname();
```

im Dialogskript definiert ist, oder als

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

wenn die Funktion als

```
{export | reexport} function c contentfunc Funktionsname ();
```

im Dialogskript definiert ist.

Der Parameter dieser Funktion ist fest vom Dialog Manager vorgegeben und kann nicht verändert werden. Diese Funktion wird immer dann aufgerufen, wenn bei dem Tablefield, das diese Funktion als Nachlade-Funktion definiert hat, in einen Bereich gescrollt worden ist, in dem kein Inhalt mehr im Dialog Manager vorhanden ist und dieser neu von der Anwendung geladen werden soll.

Zu beachten

Der Dialog Manager unterscheidet nicht, ob Attribute mit sichtbarer oder unsichtbarer Ausprägung (z.B. *.content* oder *.userdata*) in einer Funktion gesetzt werden. Bei jeder Änderungen eines Feldes (entspricht einer beliebigen Attributsetzung) wird dieses Feld als initialisiert betrachtet.

Werden beispielsweise mit **DM_SetVectorValue** nur die *.userdata* eines Tablefield-Bereichs gesetzt, so gilt dieser Bereich für den Dialog Manager dennoch als initialisiert und es wird für diesen Bereich keine Nachlade-Funktion mehr aufgerufen.

Beispiel

Im Dialog ist eine Nachlade-Funktion wie folgt definiert:

```
!! Füllt das Tablefield beim Scrollen auf,
!! wenn es notwendig wird
function contentfunc CONTENT();
```

Diese Funktion wird dann an ein Tablefield wie folgt gebunden:

```
child tablefield T1
{
  .visible true;
  .xauto 0;
  .xleft 1;
  .xright 1;
  .yauto 0;
  .ytop 0;
  .ybottom 1;
  .posraster true;
  .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;
}
```

Die Realisierung der Funktion in C sieht dann wie folgt aus:

```
/*
** Funktion zum dynamischen Auffuellen des Tablefields
** beim Scrollen, wenn Zeilen im Tablefield erreicht werden,
** die noch nicht gefuellt sind.
*/

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;

/*
** Öffnen der Datei, aus der der Inhalt des Tablefields
** gelesen werden soll
*/
if (!(f = fopen("bsp.dat","r")))
{
  DM_TraceMessage("Cannot open In-File", DMF_LogFile);
  return;
}
/*
** Allokierung von Speicher, damit die gelesenen Zeilen
** vor der Zuweisung an das Objekt zwischengespeichert
** werden können. Dazu werden die DM_-Funktionen genutzt.
*/
strvec  = (char **) DM_Malloc ( (CONT_ROWS * COLUMNS)
      * sizeof (char*) );
buffer  = (char *) DM_Malloc (STR_LEN * sizeof (char) );

/*
** erste Zelle des Tablefields, die gefuellt wird
*/
startIdx.type = DT_index;
startIdx.value.index.first = args->loadfirst - 1;
startIdx.value.index.second = 1;
/*
** letzte Zelle des Tablefields, die gefuellt wird
*/
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;

/*
** Einlesen der Datei in den Zwischenspeicher.
** Dabei wird nach jedem Leerzeichen ein neues Element
** im Tablefield angefangen
*/
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";
    }
```

```
    }
  }
  /*
  ** Setzen der Anzahl der gültigen Einträge in dem Vektor
  */
  vector.count = (ushort) vpos;

  /*
  ** Zuweisen des aufgebauten Vektors an das Tablefield
  */
  DM_SetVectorValue(args->object, AT_field,
        &startIdx, &endIdx, &vector, 0);

  /*
  ** Freigeben des Speichers, der in dieser Funktion
  ** allokiert worden ist.
  */
  while (--vpos >= 0 )
    DM_Free(strvec[vpos]);
  DM_Free(strvec);
  DM_Free(buffer);

}
```

5.6 Datenfunktionen

Funktionen, die in der Dialogbeschreibung als Datenfunktionen deklariert sind, müssen in C wie folgt deklariert werden:

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

wenn die Funktion als

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

im Dialogskript definiert ist

oder als

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

wenn die Funktion als

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

im Dialogskript definiert ist.

Der Parameter dieser Funktion ist fest vom ISA Dialog Manager vorgegeben und kann nicht verändert werden. Sie repräsentiert ein Datenmodell (Model-Komponente) welches die Präsentationsobjekte (View-Komponente) mit Datenwerten versorgt oder diese speichert und verwaltet.

Diese Funktion wird aufgerufen wenn eine Synchronisation zwischen View- und Model-Komponente erforderlich wird. Dies kann entweder automatisch geschehen, entsprechend den Steuerungsoptionen im *.dataoptions[]*-Attribut an den beteiligten Komponenten. Der Aufruf kann aber auch ausgelöst sein durch explizite Nutzung der Methoden **:collect()**, **:propagate()**, **:apply()** oder **:represent()**. Der Aufruf erfolgt einzeln für jedes Model-Attribut.

Beispiel

Dialogdatei

```
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-Datei

```
#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);
}
```

Siehe auch

Funktion DM_DataChanged

5.7 Canvas-Funktionen

Funktionen, die in der Dialogbeschreibung als Canvas-Funktionen deklariert sind, müssen in "C" fenstersystemabhängig programmiert werden. Dabei wird in der Regel auf das zugrundeliegende Fenstersystem zugegriffen, um die gewünschten Aktionen auszuführen.

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

wenn die Funktion im Dialogskript als

```
{export | reexport} function canvasfunc Funktionsname();
```

definiert ist, oder als

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

wenn die Funktion im Dialogskript als

```
{export | reexport} function c canvasfunc Funktionsname();
```

definiert ist.

Beispiel

Im Dialog ist eine Canvas-Funktion wie folgt definiert:

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

Die Zuordnung zur Canvas sieht wie folgt aus:

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

Eine Realisierung für das Fenstersystem Motif sieht dann wie folgt aus:

```
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-Funktionen

Funktionen, die als Input-Handler-Funktionen installiert werden, müssen in "C" programmiert werden. Ihre Aufgabe besteht darin, zusätzliche Eingabequellen zu verarbeiten und die Eingabe in geeigneter Art und Weise in den Dialog einzuschleusen.

Je nach Art des zugrundeliegenden Fenstersystems sind die Quellen solcher externer Nachrichten unterschiedlich. Unter Motif auf Unix kann mit Hilfe solcher Funktionen auf zusätzlichen File-deskriptoren auf Nachrichten gewartet werden, die von anderen Prozessen kommen können. Unter Windows wird in diesem Input-Handler auf spezielle Messages gewartet, die ihrerseits von anderen Prozessen kommen können. Diese Funktion ist Fenstersystem und Betriebssystem abhängig, daher gelten je nach System verschiedenen Definitionen für diese Funktion:

Definition für Motif

```
DM_Pointer Funktionsname
(
  DM_Pointer userdata,
  int fdescr,
  DM_UInt iomode
)
```

Definition für Windows

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

Beispiel

Aufgabe

Wie kann das Programm, wenn der Anwender auf 'Windows beenden' klickt , nochmals kurz dazwischen kommen, um z.B. eine ***messagebox*** aufzumachen, die ihn fragt, ob er seine Änderungen speichern will oder nicht.

Lösung

Vom ISA Dialog Manager direkt wird dafür kein Event verschickt. Es kann aber ein DM_InputHandler auf die 'WM_ENDSESSION' Message installiert werden. In diesem Handler darf **keine** DM-Funktion gerufen werden, außer **DM_QueueExtEvent**. Dieses macht allerdings keinen Sinn, da die Anwendung keine Chance mehr erhält weiterzuarbeiten.

Dialogdatei

```
dialog WinExit {}

messagebox M
{
  .title "Meldung";
  .text "Wollen sie speichern ?";
  .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
      !! Daten Speichern
    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, "Wollen sie speichern ?",
      "Meldung", 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-Funktionen

Die Format-Funktionen müssen wie folgt in der Anwendung definiert werden:

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

Die Format-Funktion wird im Dialogskript wie folgt definiert:

```
{export} function formatfunc Funktionsname ();
```

Parameter

DM_FmtRequest far * req

In diesem Parameter wird die eigentliche Aufgabe, die von der Format-Funktion übernommen werden soll, übergeben. Dazu wird im Element *task* dieser Struktur die Aufgabe übergeben.

Task	Bedeutung
FMTK_parseformat	Bei dieser Task soll der angegebene Formatstring geparst und eine Initialisierung der zu dem Format gehörenden Strukturen durchgeführt werden.
FMTK_cleanformat	Diese Task soll die abgelegten privaten Format-Informationen wieder freigeben, da das Format nicht mehr benötigt wird.
FMTK_create	Diese Task soll die Initialisierung der privaten Daten für den eigentlichen Inhaltstring durchführen, sodass nach dieser Task auf den Inhaltstring zugegriffen werden kann.
FMTK_destroy	Diese Task soll die in *contPriv* abgelegten privaten Daten wieder freigeben.
FMTK_setcontent	Es wird ein neuer Inhaltstring gesetzt. Dieser muss nach *cont* übertragen werden. Gegebenenfalls müssen die Daten in *contPriv* aktualisiert werden. Falls *eine Display-Struktur* vorhanden ist, muss der neue Inhaltstring formatiert werden und das Ergebnis *in der Display-Struktur* abgelegt werden.

Task	Bedeutung
FMTK_setselection	Es werden neue Positionen des Cursors und des Anfangs des Selektionsbereichs bezüglich des Inhaltsstrings gesetzt. Diese Werte müssen nach *cont* übertragen werden. Gegebenenfalls müssen die Daten in *contPriv* aktualisiert werden. Falls *dpy* vorhanden ist, müssen die entsprechenden Positionen bezüglich des formatierten Darstellungsstrings aktualisiert werden.
FMTK_setmaxchars	Es wird eine neue Maximallänge des Inhaltstrings gesetzt. Dieser muss nach *cont* übertragen werden. Gegebenenfalls müssen der Inhaltstring und die Daten in *contPriv* aktualisiert werden. Falls *dpy* vorhanden ist, muss der neue Inhaltstring formatiert werden und das Ergebnis in *dpy* abgelegt werden.
FMTK_formatcontent	Falls die Display-Struktur vorhanden ist, muss der neue Inhaltstring formatiert werden und das Ergebnis in dieser Struktur abgelegt werden.
FMTK_enter	Diese Task wird aufgerufen, wenn der editierbare Text, der dieses Format benutzt, den Fokus erhält. Sie muss keine speziellen Aufgaben durchführen.
FMTK_leave	Diese Task wird aufgerufen, wenn der editierbare Text, der dieses Format benutzt, den Fokus verliert. Sie muss keine speziellen Aufgaben durchführen.
FMTK_modify	Diese Task wird aufgerufen, wenn der formatierte Darstellungsstring in der Display-Struktur modifiziert werden soll. Die gewünschte Modifizierung muss so im Inhaltstring durchgeführt werden, dass dessen erneute Formatierung den modifizierten Darstellungsstring ergibt. Der Darstellungsstring muss ebenfalls entsprechend modifiziert werden.
FMTK_keynavigate	Diese Task wird aufgerufen, wenn die Positionen im Darstellungsstring relativ zu den bisherigen Positionen verändert werden sollen. Die jeweiligen Positionen im Inhaltsstring in "cont" sind so zu berechnen, dass sie den neuen Positionen im Darstellungsstring entsprechen. Die neuen Positionen im Darstellungsstring müssen in neuen Positionen im Inhaltstring umgerechnet und in der Display-Struktur abgelegt werden.

Task	Bedeutung
FMTK_setcursorabs	Diese Task wird aufgerufen, wenn die Positionen im Darstellungsstring auf einen neuen Wert gesetzt werden sollen. Die jeweiligen Positionen im Inhaltsstring in cont sind so zu berechnen, dass sie den neuen Positionen im Darstellungsstring entsprechen. Die neuen Positionen im Darstellungsstring müssen aus den neuen Positionen im Inhaltsstring umgerechnet und in der Display-Struktur abgelegt werden.

DM_FmtFormat far * fmt

In diesem Parameter wird ein Zeiger auf formatspezifische Informationen übergeben.

DM_Pointer *fmtPriv

Im diesem Parameter können von der Format-Funktion formatspezifische private Daten abgelegt werden. Diese Daten sind nur der Anwendung bekannt und werden daher vom Dialog Manager nicht ausgewertet.

DM_FmtContent far * cont

In diesem Parameter wird ein Zeiger auf eine Struktur übergeben, in der der eigentliche Inhalt des Feldes gespeichert werden soll. Dazu müssen hier die entsprechenden Elemente immer aktualisiert werden.

DM_Pointer *contPriv

In diesem Parameter können von der Format-Funktion inhaltsspezifische private Daten abgelegt werden. Der Inhalt dieser Daten ist nur der Anwendung bekannt und wird daher nicht vom Dialog Manager ausgewertet.

DM_FmtDisplay far * dpy

In diesem Parameter wird ein Zeiger auf die Struktur übergeben, in der die eigentliche Anzeigeinformation gespeichert wird. Diese Struktur muss immer, wenn es notwendig ist, aktualisiert werden, damit das zugehörige Eingabefeld im Fenstersystem richtig angezeigt werden kann.

Beispiel

```
/* 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;
```

Die Zuordnung dieser Funktion zu einem Eingabefeld erfolgt dann wie folgt:

```
/* 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;
}
```

Diese Funktion wird dann wie folgt in C realisiert:

```
DM_Boolean DML_default DM_CALLBACK My_Formatter __6(
(DM_FmtRequest far *, req),
(DM_FmtFormat far *, fmt),
(DM_Pointer *, fmtPriv),
(DM_FmtContent far *, cont),
(DM_Pointer *, 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 **) (DM_Pointer) fmtPriv, cont,
            (DM_FmtContentDef **) (DM_Pointer) contPriv, dpy);
      }
      else
        retval = FALSE;
    }
    break;
```

```
    default:
    /*
    ** if no special handling is neccessary
    ** use standard formatter
    */
    retval = DM_FmtDefaultProc(req, fmt,
      (DM_FmtFormatDef **) (DM_Pointer) fmtPriv,
      cont, (DM_FmtContentDef **) (DM_Pointer) contPriv, dpy);
    break;
    }
  return (retval);
}
```

Siehe auch

Attribut formatfunc in der „Attributreferenz“

Ressource format in der „Ressourcenreferenz“

6 Attribute und Definitionen

Um Zugriffe von der Anwendung auf den Dialog Manager zu ermöglichen, stehen in der Include-Datei IDMuser.h eine Reihe von symbolischen Namen zur Verfügung, die hier kurz erläutert werden sollen.

6.1 Attributdefinitionen und ihre Datentypen

Die für die einzelnen Objektarten zulässigen Attribute entnehmen Sie bitte der „Objektreferenz“. Prinzipiell werden sie aber aus dem Namen abgeleitet, der in der Regelsprache verwendet wird, indem ein „AT_“ dem Namen vorangestellt wird.

Beispiel

Regelsprache *.visible* => C-Schnittstelle *AT_visible*

6.2 Definitionen für die Datentypen der Attribute

Die für die einzelnen Attribute zulässigen Datentypen entnehmen Sie bitte der „Attributreferenz“. Prinzipiell werden sie aber aus dem Namen abgeleitet, der in der Regelsprache verwendet wird, indem ein „DT_“ dem Namen vorangestellt wird.

Beispiel

Regelsprache *string* => C-Schnittstelle *DT_string*

6.3 Zugriff auf benutzerdefinierte Attribute

Da die Zuordnung von Attributnamen zur internen Attributkodierung dynamisch ist, ist die Ermittlung der Attributkodierung erst zur Laufzeit möglich.

Die Attributkodierung ist nur innerhalb ihres Dialoges gültig, d.h. gleiche benutzerdefinierte Attribute haben in verschiedenen Dialogen unterschiedliche interne Kennungen.

Um nun aus einem im Programm bekannten Namen eines Attributs die Attributkodierung selbst ermitteln zu können, muss wie folgt vorgegangen werden:

Beispiel

Dialogskript:

```
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";
    /* der Attributname (vgl.Beispiel oben)*/

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

Dieses Schema kann für jedes benutzerdefinierte Attribut verwendet werden.

Nun kann auf das Attribut (.Position) zugegriffen werden, z.B. mit Hilfe der Funktion DM_GetValue:

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

Dieser Aufruf liest den fünften Wert aus dem benutzerdefinierten Attribut .Position aus.

6.4 Klassendefinition

Folgende Klassendefinitionen sind in der Include-Datei "IDMuser.h" enthalten:

Klassenidentifikator	Bedeutung
DM_ClassAccel	Accelerator
DM_ClassApplication	Application
DM_ClassCanvas	Canvas
DM_ClassCheck	Checkbox
DM_ClassColor	Farbe
DM_ClassCursor	Cursor
DM_ClassDialog	Dialog
DM_ClassEdittext	editierbarer Text
DM_ClassGroupbox	Groupbox
DM_ClassListbox	Listbox
DM_ClassFont	Zeichensatz
DM_ClassFunc	Funktion

Klassenidentifikator	Bedeutung
DM_ClassImage	Bild
DM_ClassImport	Bild
DM_ClassMenubox	Menübox
DM_ClassMenuitem	Menüeintrag
DM_ClassMenusep	Menüseparator
DM_ClassMessagebox	Messagebox
DM_ClassModule	Module
DM_ClassNotebook	Notebook
DM_ClassNotepage	Notepage
DM_ClassPoptext	Poptext (Combobox)
DM_ClassPush	Pushbutton
DM_ClassRadio	Radiobutton
DM_ClassRect	Rechteck
DM_ClassRule	Regel
DM_ClassScroll	Scrollbar
DM_ClassSetup	Setup
DM_ClassSpinbox	Spinbox
DM_ClassStatext	statischer Text
DM_ClassStatusbar	Statusbar
DM_ClassTablefield	Tablefield
DM_ClassText	Text
DM_ClassTile	Muster
DM_ClassTreeview	Treeview
DM_ClassVar	Variable
DM_ClassWindow	Fenster

7 Übersetzen und Linken von DM-Programmen

In diesem Kapitel wird beschrieben, wie mit DM entwickelte Programme übersetzt und gelinkt werden müssen.

7.1 Include-Dateien

Alle Source-Dateien der Anwendung, die irgendwelche Beziehungen zum DM haben, müssen die vom DM bereitgestellte Include-Datei **IDMuser.h** beinhalten. Abhängig davon, wo diese Datei installiert wurde, kann diese auf die folgende Art und Weise eingebunden werden:

1. `# include <IDMuser.h>`
   Diese Methode kann dann angewendet werden, wenn die Datei im Suchpfad des C-Compilers installiert worden ist. Dieser Suchpfad kann dabei durch die Anweisung -I<directory> erweitert werden.
2. `# include "pfad/IDMuser.h"`
   Auf diese Art und Weise kann die Datei eingebunden werden, wenn sie nicht im Suchpfad des C-Compilers installiert worden ist. Diese Methode ist dabei weniger flexibel und sollte deshalb nicht verwendet werden.

7.2 Compilerflags

Damit der DM richtig arbeiten kann und die Anwendung die richtige Definition aus der Include-Datei **IDMuser.h** verwendet, müssen beim Aufruf des Compilers verschiedene Definitionen gesetzt sein. Diese Definitionen beinhalten dabei die Art des Betriebssystems, die Version des Betriebssystems, den Namen des verwendeten Compilers und die Art des eingesetzten Toolkits.

Wenn zum Beispiel eine Datei für eine HP 9000-800 mit Motif und der Betriebssystemversion 10.0 übersetzt werden soll, sehen die notwendigen Symbole wie folgt aus:

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

Als Alternative zu dieser Möglichkeit kann vor der Verwendung der Datei **IDMuser.h** eine weitere Datei eingebunden werden, die obige Definitionen enthält.

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

7.3 Spezielle C-Compiler

Da auf den unterschiedlichen Systemen unterschiedliche C-Compiler existieren, gibt es in der Include-Datei **IDMuser.h** compilerabhängige Definitionen.

Beispiel

```
--DMSC
```

ist das Symbol, das definiert sein muss, wenn der Microsoft-C-Compiler benutzt wird.

Falls Ihr C-Compiler Funktionsprototypen versteht, sollten die in **IDMuser.h** vorhandenen Funktionsdefinitionen für die DM-Funktionen verwendet werden.

7.4 Übersicht

Die Angabe der richtigen Optionen und Libraries müssen der ausgelieferten Datei **MakeDefs** entnommen werden.

Wenn Sie Ihr Programm linken wollen, müssen Sie es mit einer der oben spezifizierten Libraries und einer toolkit-spezifischen Dialog Manager-Library linken. Damit sollten Sie unbedingt die im Makefile angegebene Reihenfolge der Libraries einhalten, um unerwünschte Effekte zu vermeiden.

Wenn Ihre Anwendung unter MOTIF laufen soll, müssen Sie die Library **libIDM.a** zu Ihrem Programm linken.

Toolkit	Notwendige Dialog Manager Libraries
MOTIF	-lIDM bzw. libIDM.a und -lIDMinit.a
MSW / WIN32	dm.lib und dminit.lib

Für jedes System werden zwei Bibliotheken unterschieden:

- eine Bibliothek, die während der Programmentwicklung zum Einsatz kommt:
  libIDM.a (für MOTIF)
  dm.lib bzw. **dmdll.lib** (für MICROSOFT WINDOWS)
- eine Runtime-Library, die zur Programmauslieferung gebraucht wird:
  libIDMrt.a (für MOTIF)
  dmrt.lib bzw. **dmrtdll.lib** (für MICROSOFT WINDOWS)

Der Unterschied zwischen den beiden Bibliotheken besteht darin, dass in der Entwicklungsbibliothek mehrere Sicherheitsabfragen ("checks") gemacht werden und ASCII-Dateien eingelesen und verarbeitet werden.

In der Auslieferungsbibliothek werden nur wenige Checks gemacht und es gibt keine Möglichkeit, ASCII-Dateien einzulesen oder zu verarbeiten. Dadurch wird der Speicherplatz in dieser Bibliothek deutlich minimiert.

Aus diesem Grund muss für Programme, die ausgeliefert werden, die Runtime-Bibliothek verwendet werden!

7.5 Übersetzen auf dem PC

Auf dem PC müssen abhängig vom verwendeten Betriebssystem, Fenstersystem und eingesetzten Compiler unterschiedliche Kommandozeilenoptionen beim Übersetzen angegeben werden:

Im Verzeichnis "demos" im Makefile können Sie die aktuell gültigen Optionen erhalten.

Diese Optionen sind für alle direkt vom Dialog Manager aufzurufenden Funktionen verbindlich. Ebenso sollten Sie die im Makefile angegebene Reihenfolge der Libraries einhalten.

Index

E

F

G

H

I

S

T

U

V

W

Z