ISA Dialog Manager

DISTRIBUTED DIALOG MANAGER (DDM)

A.06.03.b

In this manual the network option of the ISA Dialog Manager (Distributed Dialog Manager, DDM) is depicted. With this option distributed applications can be developed where the user interface and the application logic reside on different computers within a network.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Germany

Notation Conventions

DM will be used as a synonym for Dialog Manager.

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

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

Description Mode

All keywords are bold and underlined, e.g.

variable **integer** **function**

Indexing of Attributes

Syntax for indexed attributes:

[I]

[I,J] meaning [row, column]

Identifiers

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

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

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

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

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

<digit> ::= 1 | 2 | 3 | … 9 | 0

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

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

Table of Contents

1 Distribution of Application to Various Hosts

The Distributed Dialog Manager ("DDM") enables you to distribute an application to various processes. The distribution should take place in useful units in order to keep the communication effort between processes as low as possible. Otherwise there might occur a high load on the net or on the host which will necessarily diminish the performance of the application.

One process handles the user interface and processes user interactions; the other processes handle all the algorithmic part of the application.

These processes can run on different machines with different operating systems, for instance, the user interface runs on Microsoft Windows under DOS and the application runs on an UNIX-based system. Communication is handled by the DDM using basic communication services, and by the network that links the machines.

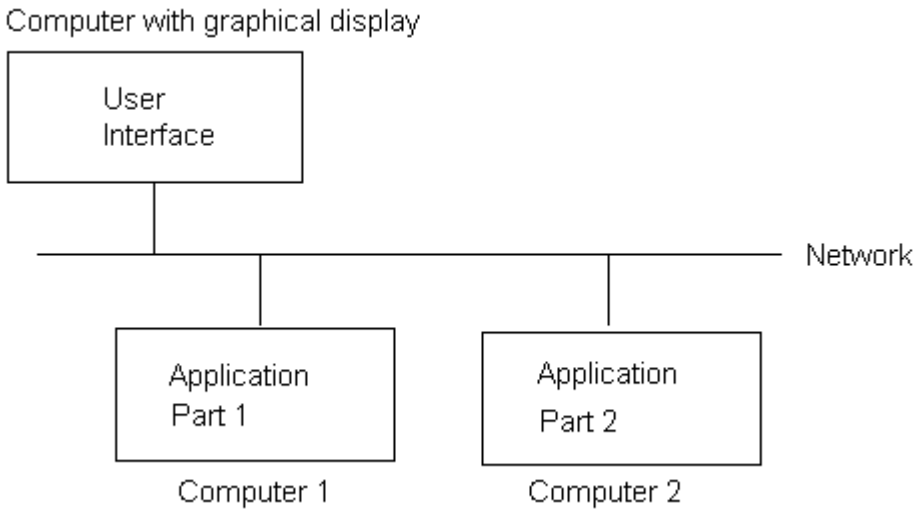

Figure 1: *Distributed Dialog Manager*

A part of the application can of course remain on the display machine when the application is distributed in order to maximize the utilization of system resources, as well as to realize output of free graphics or to request external interfaces.

2 General Architecture

The DDM is based on the basic communication services that have to be implemented on the machine. In reference to the ISO OSI reference model of communication, the DDM realizes layer 5 ("session layer") and layer 6 ("presentation layer").

For the DDM to run properly, the host has to provide layers 1 through 4 ("physical", "datalink", "network", and "transport").

The DM sends a request to the application part if the user interface wants to call the application. The DM packs and unpacks the information between processes. The communication mechanism between the processes is equivalent to "rpc" (remote procedure call), i.e. individual processes get stubs that are necessary for communication. These stubs are part of the DM.

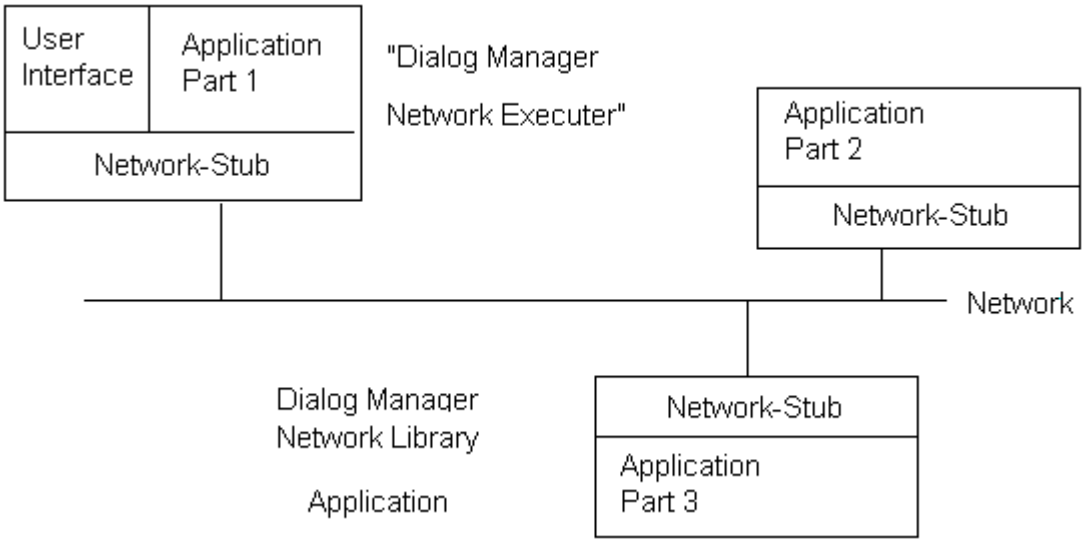

Figure 2: *General Architecture*

A DM Network Executer is necessary to let the user interface run. This program handles communication between the various parts of the application. The DM Network Executer can either be provided (simulation program **idmndx**), or can be created by linking specific DM libraries.

Networkable applications are needed on the other side. Applications can be made networkable by linking the application with the DM network library (**libIDMnet.a**) and adding networking functions to the source code (**AppInit**, **AppFinish**). In order to be able to realize additional communication between various parts of the application, the DM provides mechanisms that are used to call functions transparently.

3 Adaption of Non-distributed to Distributed Applications

Adapting a regular, non-distributed application to a distributed application requires changes in both dialog file and source code.

Please follow this process in the included sample program **list**.

3.1 Definition of Application Parts - Subdivision

The existing dialog file has to be subdivided into useful units, so-called applications (***application***). The changes are limited to the function definitions and can be described as follows:

- Functions which are to be executed on the display machine (user interface), i.e. locally, have to be defined globally for the dialog (as usual).
- Functions which are to be executed on other hosts and in other parts of the application, have to be defined in the application.

Functions may be defined only once, i.e. a function name may be used only once throughout a distributed application.

The application definition also includes the specifications of the program name, the host name, the protocol to be used, and the connection establishment to the application. These specifications can be changed during runtime by setting specific attributes (with limitations, see chapter "Object application").

3.2 Definition of a Fallback Strategy

System-dependent errors can occur during the program runtime when a network is used, which is not the case in local applications. As a consequence of such errors, not all functions necessary for the application may be useable since other hosts or the network have failed. In distributed applications, such errors can be recognized and at least partially be avoided or remedied. There are various possibilities of reacting to errors:

- Dynamically change the host name or program name to the name of a host or program that can be accessed during runtime. Then restart the relevant application parts.
- Start or link application parts locally, which can be achieved by setting the attribute .local to true. This method only makes sense if the local application contains the necessary function calls, but does not use them in a normal situation because of performance considerations. If an application is started locally, the corresponding start rule *on <application> start* is executed. The function that transfers the locally provided functions to the DM can be called in this start rule. Then proceed as usual.

Example

In the example **list**, variant b is to be used, i.e. local linking. The start rule is changed so that the local application is switched to local if starting the remote application does not succeed. The names of the now locally available functions are transferred to the DM with the additional function `InitTestAppl`. The function is called in the start rule of the application: *on TestAppl start*.

Those are all necessary changes to the dialog file. Additional changes are necessary in the application source code.

3.3 Source Code Changes

At least two applications have to be linked for a complete distributed application:

- the local part that is linked to the display (user interface),
- the part that serves as remote application.

No changes are required for the local part. The program is started as usual with **AppMain**. The regular initialization has to be carried out, as in the non-distributed DM.

No **AppMain** may be contained in the remote application. However, the remote application has to contain the functions

- **AppInit** and
- **AppFinish**

to initialize and finish the application. The task of the **AppInit** function is to initialize the actual application and transfer the functions contained in it to the DM. The task of the **AppFinish** function is to correctly finish the corresponding application part.

Example "list"

In order to have only one source code file for both local and remote application, the source contains conditional statements (`ifdef`). If XXX_NETWORK is defined, the four functions as well as **AppInit** and **AppFinish** are called. If XXX_NETWORK is undefined, **AppMain** is defined as usual, plus the function `InitTestAppl`.

For the example, the local linking of functions was implemented as a fallback, i.e. the local source file contains the four functions required by the example in addition. However, they are used only if the remote application can not be started.

3.4 Makefile Changes

Two programs are to be created instead of one. To achieve this, the remote application merely has to be linked with the DM network library (**libIDMnet.a**).

In addition to the traditionally used

- "libIDM.a" (Motif)
- "dm.lib" (MS-Windows),

the local application has to be linked with the library for the DM Network Executer

- "libIDMndx" (Motif)
- "dmndx.lib" (MS-Windows)

in which the library for the DM Network Executer has to be given first.

The result are two application parts that can communicate with each other.

To adapt the application to your system environment, the host name given in the dialog file has to be changed to a name available in your network.

4 Functionality and Syntax in the Dialog Script

In order to process distributed applications, new constructs have to be implemented in the dialog description language. The object application is responsible for this implementation. The application object contains information about the application, application functions, and type of communication. The resulting applications can be started and ended individually.

The subdivision into application informs the DM how functions are assigned to each other and from an application for the user.

4.1 Object Application

Basically, the application object is to be treated like a regular object, except that it contains no display information at all. It has the following attributes.

Attribute	Access	Significance
.active	RW	Defines and requests whether the application is currently active. Changing this attribute from *false* to *true* starts the application; changing it from *true* to *false* ends the application.
.connect	RW	Defines that the application should connect to a running server process which was started on the host (defined by the host name or IP address and portnumber).
.exec	RW	Defines that the application should start a process given by the path on the host (defined by the host name or IP address). This attribute's value contains the program name, path, host name, and miscellaneous information.
.label	RW	Internal application name, identifier.
.local	RW	Defines whether the application runs locally or on a network.
.transport	RW	Defines the internal transport mechanism to be used by the application. The following is possible: *tcpip*.

The attributes *.transport*, *.connect*, *.local*, and *.exec* can only be changed if .active is set to false.

The attributes *.connect* and *.exec* depend on the used transport, i.e. future new types of the transport layer may imply different types of connection establishment.

4.1.1 TCP/IP

If the TCP/IP protocol is used, the syntax used for *.connect* is this:

```
"<host>:<portnumber>"
```

The <*host*> parameter contains the host name, the <*port*> parameter contains the port number.

If the TCP/IP protocol is used, the syntax used for *.exec* is this:

```
"<host>[%<username>[%<password>]]:<path>"
```

The <*host*> parameter contains the host name on which the program is to be started. The <*username*> parameter may contain a user name existent on the host. The <*password*> parameter may contain the valid password of that user. (These two parameters are optional, i.e. they are compelling dependent on the kind of installation.)

The <*path*> parameter contains the path of the program to be started.

4.1.2 IPv6 Support

As of IDM version A.05.02.i, the DISTRIBUTED DIALOG MANAGER (DDM) supports the IPv6 protocol on all architectures that **natively** support IPv6.

When an IPv6 address is defined in the dialog script, it has to be written in brackets ('[' and ']'), as is the case with URLs (e.g. "[::1]").

4.2 Assignment of Functions

The functions contained in a dialog have to be distributed over various applications. The same function may not be assigned to more than one application.

The assignment to applications is carried out when the functions are declared. The functions for a specific application are declared within the declaration of that application, i.e. they are defined like the children of an object. If functions are defined in a dialog outside of an application, they are assigned to the dialog and have to be linked to the dialog.

4.3 Example

This example shows how the application can be split into two parts; one part in the dialog, one part as a remote application.

```
dialog Example

application Remote
{
  .exec   "boole:/usr/bin/example1";
  .active false;
```

```
  function c integer LoadDB;   /* remote function */
}

function callback CheckValue;  /* local function */

on dialog start
{
  Remote.active := true;
  if (not Remote.active) then
    exit();
  endif
}
```

5 Application Interface

In order to realize a networkable application, the application has to be equipped with specific functions for starting and ending the application. Additionally, a DM utility function can be used for communication between application parts. This function can be used to call network-independent functions, no matter whether the function is realized locally or remotely.

5.1 C Interface

The main function **AppMain** has to be replaced by the functions

- **AppInit**
- **AppFinish**

if an application which is not linked to the dialog is to be started or ended in a distributed environment.

With help of the function DMCallFunction, functions of any application parts can be called.

Example

```
int DML_c AppInit __4(
(DM_ID, appl),
(DM_ID, dialog),
(int, argc),
(char **, argv))
{
  if (!DM_BindCallBacks(ApplFuncMap, ApplFuncCount, appl, DMF_Silent))
    DM_TraceMessage("There are some functions missing", 0);
  return 0;
}

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

5.2 COBOL Interface

The COBOL main program **COBOLMAIN** has to be replaced by the functions

- **COBOLAPPINIT**
- **COBOLAPPFINISH**

Example

```
* SET OSVS

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOLMAIN.

DATA DIVISION.
WORKING STORAGE SECTION.
COPY "IDMcobws.cob".

77    NULL-OBJECT        PIC 9(9) BINARY VALUE 0.
77    FUNC_NAME            PIC X(32) VALUE SPACES.
77    BUFFER            PIC X(80) VALUE SPACES.

LINKAGE SECTION.
01    EXIT-STATUS PIC 9(4) BINARY.
01    APPLIC-ID PIC 9(9) BINARY.
01    DIALOG-ID PIC 9(9) BINARY.

PROCEDURE DIVISION USING EXIT-STATUS.
MAIN SECTION.

WAKING-UP.
    DISPLAY "COBOLMAIN CALLED".
    GOBACK.

    ENTRY "COBOLAPPINIT" USING EXIT-STATUS APPLIC-ID DIALOG-ID.
INIT SECTION.

WELCOME.
    DISPLAY "COBOLAPPINIT CALLED".

INITIALIZE -IDM.
    MOVE "@" TO DM-SETSEP.
    MOVE LOW-VALUE TO DM-GETSEP.
    MOVE "DMcob_Initialize" TO FUNC-NAME.
    CALL "DMcob_Initialize" USING DM-STDARGS DM-COMMON-DATA.

BIND-FUNCTIONS.
    CALL "BindFuncs" USING DM-STDARGS APPLIC-ID.
    PERFORM ERROR CHECK.
    GOBACK.

    ENTRY "COBOLAPPFINISH" USING EXIT-STATUS APPLIC-ID
        DIALOG-ID.
FINISH SECTION.
```

```
GOOD-BYE.
    DISPLAY "COBOLAPPFINISH CALLED".
GOBACK.
```

5.3 Records and Function Declaration

If you want to use records in an application, you have to create the necessary file with help of the simulator with the options

-application

and

+writetrampolin.

Call

```
idm +application <applicationname> +writetrampolin <filename> <dialogfile>
```

This created C-file has to be compiled as usual.

If you want to define C-function prototypes for an application, the simulator has to be started also with the options **+application** and **+writetrampolin**.

Call

```
idm +application <applicationname> +writeproto <filename> <dialogfile>
```

6 Principal Work in Distributed Environment

The following conditions have to be noticed when programming distributed applications.

- The function call has usually a very large overhead, i.e. function calls over the network are relatively expensive.
- The amount of data given at a function call has only minor significance for the computers' overall load.

Thus, for programming with Dialog Manager, the following basic rules can be derived from this facts:

- The functions should be distributed in the way that very often needed functions are to be realized on the display machine.
- All informations needed by the functions should be given when calling application functions. These information can be packed in arbitrarily structured records and can be given to the application function. This is much more efficient than later query attributes with the interface function DM_GetValue.
- Objects should be principally processed with the mightiest available function. Thus, when processing listboxes and tablefields, the functions DM_GetContent and DM_SetContent should be used.

7 Command Line Options and Modes

The Dialog Manager supports various protocol types that can be used for the basic communication services. Depending on the used protocol, there are various command line options.

7.1 General Command Line Options

The application part which builds the display part, understands all options which the non-distributed program would understand, too (please refer to chapter “Command Line Options” in manual “Development Environment”).

The server part does understand only the following options:

- **-IDMversion**
- **-IDMtracefile**
- **-IDMerrfile**

I.e. all options are ignored which describe any display information.

7.2 Protocol TCP/IP

TCP/IP is available for most systems. To use this protocol, start the DM with the option **-IDMtransport tcpip**.

The complete application consists of various executables. One application can be the “DM Network Executer” (**idmndx**), if no part of the application is local. If any part of the application is local, that part is a program linked with the DM. The other applications are your application parts that are linked with the DM network library (**libIDMnet.a**) and the required system libraries for network functions.

This chapter lists the command line options that are needed when applications are started with the DDM. These options are normally used only if

- one or more parts of the applications are started as server;
- the command line options are not completely defined in the relevant dialog description file, or are to be overwritten.

Various attributes can be specified in the dialog description file to define the connection establishment, e.g. *.transport*, *.connect, .exec* (see object Application).

7.2.1 Option -IDMlisten

Starts the application in **listen mode**, i.e. the application waits to accept a connection to the other application part on a remote machine. If the other part requests a connection, the connection is accepted, and initialization was successful.

Syntax

-IDMlisten <port>

The port with number <*port*> has to be available and unoccupied on your machine. Usually, <*port*> should be larger than 1024, since most ports below 1024 are used by system services. Ports between 5800 and 7000 are used by the X Window System. To be sure that you use an available port, ask your system management, or specify a port above 10000.

After the actual application has terminated, the application started in **listen mode** is also terminated. If the application is to be restarted, the application in **listen mode** has to be started, too. Therefore, this option is usually used while a distributed application is developed, or if an application is only used infrequently.

This option is available on UNIX systems.

7.2.2 Option -IDMserve

Start the application in **server mode**, i.e. the application waits to accept a connection two one or more clients. If a client requests a connection, the server accepts the connection, searches for a free port, switches the connection part to the free port, and forks itself. Thus, the application runs twice after the application has accepted a connection. One application still waits for a connection, and the other application runs as usual.

Syntax

-IDMserve <port>

The port with number <*port*> has to be available and unoccupied on your machine (cf. above). During or after the execution of an application, the server is able to accept new connections. If the user wants to restart the application, only the other part of the application has to be restarted. The server can only be stopped by interrupting or canceling the relevant process. Therefore, this option is usually used if the application is started frequently or by more than one user at the same time.

This option is available only on systems that have implemented a fork process.

8 Requirements

Various requirements have to be met in order to utilize the DDM on different hardware platforms. Generally, the machines have to be part of a working network with a functioning communication mechanism. Additionally, the system requirements for the regular Dialog Manager have to be met.

On UNIX systems “BSD sockets” in form of libraries have to exist on the target machine.

Other Installation Requirements

Motif:

- UNIX
  C-Compiler, Motif 2.0 or CDE (based on Motif 1.2) including development environment (libraries, include files)

with COBOL interface for:

- MICRO FOCUS COBOL

with networking capabilities:

- Network requires development environment (libraries, include files)

9 Compiling and Linking

The program has to be compiled as usually with the correct options for Dialog Manager.

Subsequently, the application parts have to be linked as follows:

» Local part:

1. local application parts
2. DM library (**libIDM.a** or **libIDMaw.a** and **dm.lib**)
3. DM Network Executer library (**libIDMnx.a** and **ndx.lib**)
4. window system-specific libraries plus libraries for communication services

» Remote part:

1. remote application parts
2. DM Network library (**libIDMnet.a**)

See Also

Chapter “Compiling and Linking DM Programs” in manual “C Interface - Basics”

Chapter “Compiling and Linking Dialog Manager Programs” in manual “COBOL Interface”

10 Changes from Version A.05.01.d

The previous behavior of the distributed Dialog Manager (DDM) to react to network errors with an immediate termination of client or server has been abolished with version A.05.01.d.

The application object as a link and abstraction concept to define application parts (which are available e.g. locally, C/COBOL, via a dynamic library or via network) for the rule language gets the ability to detect errors and to react to them appropriately.

The following brief set of rules is intended to describe the IDM's expected behavior in this regard:

- An error is given if either a technical network error occurs (errors reported by the network system functions, e.g. hardware defects, connection loss due to "aborted" server process), the IDM versions between client and server side are incompatible or the IDM network protocol is violated. The latter case should certainly be reported to IDM Support.
  If the application is a dynamically linked library, it may just as well be a file that is not found.
  Another error is the missing link for the application type (e.g. idmndx library was not linked).
  An abort of the program e.g. by a "KILL" signal or by terminating a process via the "Task Manager" cannot be recognized as an application error. The other side will only get this reported as a network error.
  The .active attribute on the application object reflects the activation state, as before.
- As soon as an error is detected, the associated application is deactivated. In addition, an error code is placed on the error stack.
- A start event is triggered for a successfully activated application (initially, as well as when the .active attribute is converted). For an application that is deactivated (e.g. by exit(), DM_StopDialog, as well as by converting the .active attribute) a finish event is triggered.
  start/finish events occur initially or at the end of the application before or after a dialog start/finish, but otherwise asynchronously!
- The .errorcode and .systemerror attributes contain more detailed information about the error that occurred.
- If an error occurs during the execution of an application function, the application is deactivated and either the simulation rule is called or a Fail is generated.

If the DDM server detects an error, the network service loop is exited after processing the current server application function and AppFinish() is called, but with 0-IDs.

10.1 Consequences for previous dialogs

There are some implications that should be considered as an IDM application developer.

The IDM application (client) or server part does **not** terminate anymore, or not immediately!

The start/finish event is fired not only for a local application, but also for network applications and for local applications with dynlib transport.

From the rule language side, an error in a previously existing active application can be detected by a finish event indicating that the application has been deactivated; also by a sudden failure of an application function or by calling the simulation rule.

On the server side, an AppFinish() is called when an error is detected, thus giving the application developer the opportunity to "clean up". Errors during the processing of an application function should be handled correctly, since the server function is still being processed.

10.2 Application example

The following client/server example shows the use of simulation rules and how to react to start/finish events.

10.2.1 Client

```
!! sample.dlg
dialog D

default edittext
{ .borderwidth 0; }

default statictext
{ .sensitive false; }

default window
{ on close { exit(); } }

application Appl
{
  .active false;

  integer SimCount := 0;  // Counter for calls of
                          // simulation rules
  integer FailCount := 0;  // Counter for occurred fails

  function string FuncSimple(integer I)
  {
    !! Simulation rule in case of error
    this.SimCount := this.SimCount + 1;
    return "SIM-" + this.SimCount;
  }

  !! Server function that calls rule on the client side.
  !! Now without simulation rule, error is caught by fail()
  function string FuncComplex(object Rule, integer I);
```

```
  on start
  {
    StStatus.text := "CONNECTED";
    PbSimple.sensitive := this.active;
    PbComplex.sensitive := this.active;
    CbConnect.active := this.active;
  }
  on finish
  {
    if this.errorcode <> error_none then
      StStatus.text := "FAIL: " + this.errorcode + " - " +
                       this.systemerror;
    else
      StStatus.text := "";
    endif

    PbSimple.sensitive := this.active;
    PbComplex.sensitive := this.active;
    CbConnect.active := this.active;
  }
}

rule string Convert(integer I)
{
  return "CONV-" + I;
}

window Wi
{
  .title "DDM NetErrors";
  .width 400;
  .height 200;

  integer SimCount := 0;

  checkbox CbConnect
  {
    .width 80;
    .active false;
    .text "Connect";
    on activate
    {
      Appl.connect := EtConnect.content;
      Appl.active := true;
      EtConnect.sensitive := false;
    }
```

```
  on deactivate
  {
    EtConnect.sensitive := true;
    Appl.active := false;
  }
}

edittext EtConnect
{
  .xauto 0;
  .xleft 80;
  .content "localhost:4711";
}

statictext StStatus
{
  .ytop 30;
  .xauto 0;
}

pushbutton PbSimple
{
  .ytop 60;
  .text "Call Simple-Func";
  .sensitive false;

  on select
  {
    variable integer I;

    Appl.SimCount := 0;
    Appl.FailCount := 0;

    if fail(I := atoi(EtCount.content)) then
      I := 0;
    endif
    while(I>0) do
      if fail(StCount.text := FuncSimple(I)) then
       !! Fail occurs only if function is not connected
       !! from server was connected.
       !! Normally simulation rule is called in case of
       !! error
       Appl.FailCount := Appl.FailCount + 1;
       StCount.text := "FAILED-" + Appl.FailCount;
      endif
      updatescreen();
```

```
      I := I-1;
    endwhile
  }
}

edittext EtCount
{
  .ytop 60;
  .xleft 200;
  .content "2000";
}

pushbutton PbComplex
{
  .ytop 90;
  .text "Call Complex-Func";
  .sensitive false;

  on select
  {
    variable integer I, FailCount:=0;
    this.window.SimCount := 0;
    Appl.SimCount := 0;
    Appl.FailCount := 0;
    if fail(I := atoi(EtCount.content)) then
      I := 0;
    endif
    !! Trigger loop via extevent so that
    !! processing of finish is possible
    sendevent(this,1,I);
  }

  on extevent 1(integer I)
  {
    !! Calling the network function which in turn
    !! calls Convert rule.
    if (I>0) then
      if fail(StCount.text := FuncComplex(Convert,I)) then
       !! Network errors are caught via fail()
       Appl.FailCount := Appl.FailCount + 1;
       StCount.text := "FAILED-" + Appl.FailCount;
      else
        I := I-1;
        sendevent(this,1,I);
      endif
      updatescreen();
```

```
      endif
    }
  }

  statictext StCount
  {
    .ytop 90;
    .xleft 200;
    .xauto 0;
  }
}
```

10.2.2 Server

```
// sample.c
#include <stdio.h>
#include <IDMuser.h>
#include <samplefm.h>

static trace_errors(char *txt)
{
    DM_ErrorCode errbuf[100];
    uint nerrors, i;

    nerrors = DM_QueryError(errbuf, sizeof(errbuf), 0);
    for (i=0; i<nerrors; i++)
    DM_TraceMessage("%s - error #%d", DMF_LogFile|DMF_Printf,
                    txt, errbuf[i]);
}

DM_String DML_default DM_ENTRY FuncSimple __1((DM_Integer, I))
{
    static char buf[100];

    sprintf(buf, "FUNC-%d", I);

    return buf;
}

DM_String DML_default DM_ENTRY FuncComplex __2((DM_ID, ID),
                                               (DM_Integer, I))
{
  DM_Value args[1], retval;
  DM_String str = NULL;

  args[0].type = DT_integer;
```

```
  args[0].value.integer = I;

  if (DM_CallRule(ID, ID, 1, args, &retval, 0)
      && retval.type == DT_string)
  {
    str = retval.value.string;
  }
  else
    trace_errors("callrule");
  return str;
}

int DML_c AppInit __4((DM_ID, appl), (DM_ID, dialog),
                   (int, argc), (char **, argv))
{
    DM_TraceMessage("AppInit called", DMF_LogFile);

    BindFunctions_Appl (appl, dialog, 0);

    return 0;
}

int DML_c AppFinish __2((DM_ID, appl), (DM_ID, dialog))
{
    DM_ErrorCode errbuf[100];
    uint nerrors, i;

    DM_TraceMessage("AppFinish (%d,%d) called",
                    DMF_LogFile|DMF_Printf, appl, dialog);
    trace_errors("appfinish");

    return 0;
}
```

10.3 Notes

In principle, the IDM does not perform an automatic/periodic check of the network connection. If this is desired by the application, this can be realized e.g. via the Timer object.

10.4 Changes to the application object

The attributes .errorcode and .systemerror have been newly introduced. For more details see application object.

10.4.1 start/finish-Event

The start event is triggered at the application object when the application has been successfully activated (without errors).

The finish event is triggered when the application was previously active and deactivated, this can happen due to an error, initially when the activation fails as well as by terminating the application or changing the .active attribute.

If an error occurs during the event loop, the start/finish events of the application are processed according to the event sequence. However, this also means that the then current .active state does not necessarily match the event.

The activation of an application object, whose .active is initially set to true, takes place when the dialog is started via DM_StartDialog or start() from the rule language. The start/end rule is executed before the dialog start rule is processed, thus ensuring that the local applications can connect their functions.

If you end a dialog, e.g. via exit(), with a still active application, the finish rule is executed not until after the finish rule of the dialog.

10.4.2 Error behavior for application functions

If an error occurs during the execution of an application function, the application is deactivated and an existing simulation rule is called. If no simulation rule is available, a fail is returned to the rule interpreter.

10.5 Network application side

If an error (network error or protocol error) is detected on the network side during the call of a DM-function, the function returns with a status expressing an error.

Der Netzwerk-Stub wird nach Beendigung der Server-Funktion bei einem Fehler verlassen. Die AppFinish()-Funktion wird auch aufgerufen wenn dies Aufgrund eines Netzwerk- oder Protokollfehlers passiert solang vorher ein AppInit() aufgerufen wurde. Allerdings wird keine Applikation-ID oder Module-ID mitgegeben. Auch in diesem Fall liegt ein entsprechender Fehler auf dem Fehlerstack.

The network stub is exited after the server function has finished in case of an error. The AppFinish() function is also called if this happens due to a network or protocol error as long as an AppInit() was called before. However, no application ID or module ID is passed. Also in this case there is a corresponding error on the error stack.

Index