ISA Dialog Manager

Rule Language

A.06.03.b

This manual explains the Rule Language of the ISA Dialog Manager that is used to program the dynamic behavior of the user interface. Some of the topics are event and rule processing, data types, extent of the Rule Language, syntax of statements, and built-in functions.

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

Notation Conventions ... 3

Table of Contents ... 5

1 Introduction ... 11

2 Events ... 13

2.1 User Events ... 13

2.1.1 Objects and their User Events ... 15

2.1.2 Drag and Drop Events ... 23

2.1.2.1 Cut Event ... 24

2.1.2.2 Paste Event ... 24

2.1.2.3 Examples ... 24

2.1.2.4 Tips & Tricks for Drag & Drop and Clipboard ... 25

2.1.3 User Events with Special Inheritance ... 26

2.2 Internal Events ... 27

2.3 System Events ... 28

2.4 External Events ... 29

3 Rule Processing ... 31

3.1 Normal Rules ... 31

3.2 before Rules ... 31

3.3 after Rules ... 31

3.4 Processing Order ... 32

4 Named Rules (Subprograms) ... 35

5 Special Objects and Object Referencing ... 37

5.1 this ... 37

5.2 Object Referencing ... 37

5.3 null Object ... 40

5.4 Event Object thisevent ... 41

6 Data Types ... 43

6.1 Collection Data Types ... 44

6.1.1 Syntax of Expressions ... 45

6.1.2 List Size and Default Values 45
6.1.3 Access and Assignment of Collections 45
6.1.4 Automatic Conversion 46
6.1.5 Performance 47
6.1.6 Local and Global Variable 47
6.1.7 Relational Operators 48
6.1.8 Uninitialized variables 49

7 Functions 50

7.1 Function Called in Rules 51
7.1.1 Alias Name with Optional Code Page Definition 52
7.1.2 Calling Parameters of a Function during Runtime 53
7.2 Callback Function 55
7.3 Canvas Function 55
7.4 Data Function 56
7.5 Format Function 57
7.6 Reloading Function 57
7.7 Simulation of Functions 57
7.8 Functions in modularized dialogs 60

8 Global Variables 61

8.1 Variable Definition 61
8.2 Variable Definition with Initialization 62
8.3 Configurable Variables 62
8.4 Protecting Variables against Changes 63
8.5 Attributes of Global Variables that Can Be Changed Dynamically 63

9 Validity Range for Better Type Checking 65

9.1 Restrictions 66
9.2 Access and Value Tests 67
9.3 Derivation of the Validity Range 68
9.4 Enhancement of the IDM Syntax 69
9.5 Attributes in Relation to Validity Ranges 70

10 Actions in the Program Block 71

10.1 General Structure of a Statement 71
10.2 Commenting Instructions 72
10.3 Comments 73
10.4 Operators in the Rule Language 74
10.4.1 Assignment Operators 74

10.4.2 Comparison Operators ... 74
10.4.3 Arithmetical Operators ... 75
10.4.4 Logical Operators ... 75
10.5 Brackets in the Rule Language ... 76
10.6 Changing Attribute Values ... 77
10.7 Control of the Program Flow ... 77
10.7.1 if-then-else ... 78
10.7.2 if-elseif-else ... 78
10.7.3 case Statement ... 79
10.8 Loop Constructs ... 81
10.8.1 for Loop ... 81
10.8.2 foreach Loop ... 82
10.8.3 while Loop ... 83
10.9 Calling Named Rules ... 83
10.10 Local Variables ... 83
10.10.1 Normal Local Variables ... 84
10.10.2 Static Variables ... 85
10.11 Return of Values ... 86
10.12 Call of Application Functions ... 87

11 Built-in Functions ... 88

11.1 append() ... 88
11.2 applyformat() ... 91
11.3 atoi() ... 92
11.4 beep() ... 93
11.5 closequery() ... 95
11.6 concat() ... 97
11.7 countof() ... 99
11.8 create() ... 101
11.9 delete() ... 104
11.10 destroy() ... 107
11.11 dumpstate() ... 109
11.12 exchange() ... 113
11.13 execute() ... 116
11.14 exit() ... 120
11.15 fail() ... 122
11.16 find() ... 124
11.17 first() ... 127
11.18 getvalue() ... 128

11.19 getvector() 130
11.20 indexat() 132
11.21 inherited() 134
11.22 insert() 135
11.23 itemcount() 139
11.24 itoa() 141
11.25 join() 142
11.26 keys() 145
11.27 length() 146
11.28 load() 147
11.29 loadprofile() 148
11.30 max() 149
11.31 min() 150
11.32 parsepath() 151
11.33 print() 154
11.34 querybox() 155
11.35 queryhelp() 157
11.36 random() 158
11.37 regex() 159
11.38 run() 165
11.39 save() 166
11.40 saveprofile() 167
11.41 second() 169
11.42 sendevent() 170
11.43 sendmethod() 172
11.44 setinherit() 173
11.45 setvalue() 174
11.46 setvector() 176
11.47 sort() 179
11.48 split() 181
11.49 sprintf() 182
11.50 stop() 188
11.51 strcmp() 189
11.52 stringpos() 191
11.53 strreplace() 192
11.54 substring() 196
11.55 tolower() 197

11.56 toupper() 198
11.57 trace() 199
11.58 trimstr() 200
11.59 typeof() 201
11.60 updatescreen() 202
11.61 valueat() 203
11.62 values() 205

12 Formal Syntax of Rules and Statements 207

12.1 Operators 207
12.2 Expression 208
12.3 Statements 217

Index 221

1 Introduction

The behavior of the generated dialog objects is described in the IDM rule base.

A rule consists of the

- event and
- action part.

Basically, the IDM rules consist of the so-called "event line" and a "program block" (in braces) which will be executed as soon as the event described in the event line occurs.

The rule processing is triggered by an event that is directly or indirectly initiated by the user. This event is described in the "event line".

An event is an action made by the user or by the IDM, referring to a IDM object or its attributes (e.g. window relocating, resizing, clicking of buttons, text input, etc.).

There are the following event types:

- user events
- keyboard and help events
- internal events
- object class events
- system events
- external events.

The functionality of the rules is described by the program block, set in braces, following the event line. With this rule component the entire dialog execution including its functionality can be described by the IDM.

Actions are for example:

- changing attributes or attribute values
- calling application functions
- calling built-in functions.

```
<event line>
{
  <program block>
}
```

In this definition, the program block is executed each time the event is occurring.

Precondition

In the following definition, the program block will be executed only, if an additional precondition is fulfilled.

The keyword for the additional precondition is **if**.

The program block consists of a sequence of assignments and statements concerning conditional program execution.

```
<event line>
if( <additional precondition> )
{
  <program block>
}
```

There are further control structures to influence the program flow, e.g. the possible sub-rules "if then else endif".

Note

The precondition ("if") has principally the same structure as the "if" in the conditional program flow. However, there is one difference:

- the preconditional "if" is checked before each rule execution, whereas
- the conditional "if" is only checked when the rule is processed.

Thus the rule succession is not relevant for the preconditional "if" since a rule cannot block another rule, if the preconditions are fulfilled.

2 Events

The event line is started with the keyword **on**.

Every rule may have a maximum of one event line which, however, can consist of several events for one object

The rule-triggering events can be divided into the following groups:

- User events
- Keyboard and help events
- Internal events
- System events
- External events.

2.1 User Events

Events referring to a certain object, model or object class are called "user events". It is decisive for this kind of events that the user is the responsible cause for the event, i.e. the relevant event has been triggered as a result of user actions.

The syntax for this kind of event line is as follows:

```
on <object ID> <event> { , <event> }
```

This definition shows that a rule can be bound to only one object, but it can be bound to several events separated by a comma.

Object IDs are the identifiers for existing IDM objects such as “OK_Button”, “TestWindow”, etc.

Event types describe user-initiated events such as “selected”, “moved”, “closed”.

Example

```
on Pushbutton select
{
  Action;
}
```

The following **event types** are provided by the IDM.

Event Type	Description
activate	The indicated object has been activated. This event can be triggered for all objects whose active states can be clearly recognized by the user.
changed	Value of indicated variable or object attribute has been changed.
charinput	A character was input into an edittext. It can e.g. be checked now.
close	Close mechanism (e.g. closebox) of a window was used. A sub-tree of a treeview has been closed.
cut	The user has cut an object by Drag and Drop.
dbselect	The user has selected the object with a double click, i.e. has edited an object.
deactivate	The user has deactivated an object, e.g. if he selects a checkbox which has already been activated, so that it is now deactivated.
deiconify	The icon for a window has been re-opened.
deselect	An object has been deselected (by selecting another object or by ending the input in an edittext with the `Return` key).
deselect_enter	A text array input has been canceled by pressing the `Return` key.
extevent	An external event has occurred.
finish	End of a dialog or a module. An application has been disconnected.
focus	The input focus has been set on the indicated object.
help	The user has asked for help for the current object. Calling the help function is carried out according to the requirements of the relevant window system (e.g. on Motif & Windows: `F1` key, for Qt shortcut `SHIFT+F1`).
hscroll	The user has scrolled horizontally.
iconify	The iconifybox of a window has been activated -> window becomes an icon.
key	The user has pressed the accelerator given after the keyword **key** for the object for which this rule has been written (see also "Keyboard- and Help-events").

Event Type	Description
modified	The user has finished editing a text after actively changing it.
move	The user has moved the window. A toolbar has been moved. With a docked toolbar the move event occurs only if the movement changes the attributes *.dock_offset* or *.dock_line*.
open	A menubox has been opened. A sub-tree of a treeview has been opened.
paste	The user moved an object by Drag and Drop and has dropped it on another object.
resize	The size of a window has been changed. With a splitbox the size of a split area has been changed.
scroll	The user has scrolled.
select	An object has been selected by user.
start	Start of a dialog or a module. A connection to an application has been established.
vscroll	The user has scrolled vertically.

2.1.1 Objects and their User Events

All events which can be triggered by the user for an object are listed in the following chart.

Object	Event	Description
Application	extevent	
	finish	End of dialog (execution only if .local = true).
	start	Beginning of dialog (execution only if .local = true).
Canvas	extevent	
	focus	
	help	
Checkbox	extevent	

Object	Event	Description
	focus	
	help	
	key	
	select, activate	Checkbox "on".
	select, deactivate	Checkbox "off".
Dialog	deactivate	A timeout occurred.
	extevent	
	finish	
	help	
	key	
	start	Dialog has been started.
Edittext	charinput	
	deselect	Edittext has been exited by a mouse click or by an unallowed key.
	deselect_enter	Edittext has been exited by pressing the `Return` key.
	extevent	
	focus	
	help	
	key	
	modified	Contents of edittext has been changed.
	select, activate	Clicking on edittext.
Groupbox	extevent	
	help	
	key	

Object	Event	Description
	scroll, hscroll	Scrolling has been carried out horizontally.
	scroll, vscroll	Scrolling has been carried out vertically.
	select	Clicking into groupbox without hitting anything else.
Import	none	
Image	dbselect	
	focus	
	help	
	key	
	select	
Listbox	dbselect	Double clicking on item.
	extevent	
	focus	
	help	
	key	
	scroll, hscroll	Horizontal scrolling
	scroll, vscroll	Vertical scrolling
	scroll, activate	Item has been selected.
	select, deactivate	Item has been deactivated (only in multiselection).
Menubox	extevent	
Menuitem	extevent	
	help	
	select	Button as menuitem has been selected.

Object	Event	Description
	select, activate	Checkbox or radiobutton as menu-items were selected.
	select, deactivate	Checkbox as menuitem has been deselected.
Menuseparator	extevent	
Messagebox	extevent	
Module	extevent	
	finish	Module is de-loaded.
	start	Module has been loaded and started.
Notebook	extevent	
	focus	
	help	
	key	
Notepage	activate	
	deactivate	
	extevent	
	focus	
	help	
	hscroll	
	key	
	scroll	
	select	
	vscroll	
Poptext	activate	
	charinput	
	deselect	

Object	Event	Description
	deselect_enter	
	extevent	
	focus	
	help	
	key	
	modified	
	select	User has selected a new item.
Pushbutton	extevent	
	focus	
	help	
	key	
	select	Clicking.
Radiobutton	activate	Clicking on a non-active radiobutton.
	deactivate	
	extevent	
	focus	
	help	
	key	
	select	
Rectangle	dbselect	
	extevent	
	focus	
	select	
	help	
	key	
Scrollbar	extevent	

Object	Event	Description
	focus	
	help	
	key	
	scroll	
Setup	none	
Statictext	extevent	
	focus	
	help	
	key	
	select	
Statusbar	extevent	
	focus	
	help	
	key	
	paste	
	select	
Spinbox	charinput	
	cut	
	extevent	
	focus	
	help	
	key	
	paste	
	select	
	scroll	

Object	Event	Description
Tablefield	activate	Object in tablefield has been activated.
	charinput	The user has entered a valid character in the edittext attached to the tablefield. It can now e.g. be checked and the application can react on a key input in the tablefield.
	deactivate	Object in tablefield has been deactivated.
	dbselect	Double-click on object in tablefield.
	extevent	
	focus	Focus has been changed in a tablefield or set on a tablefield.
	help	Help event has been sent to the tablefield.
	hscroll	Tablefield has been scrolled horizontally.
	key	A key not needed by the tablefield has been pressed.
	modified	Object in tablefield has been changed.
	scroll	Tablefield has been scrolled.
	select	Tablefield has been selected.
	vscroll	Tablefield has been scrolled vertically.
Timer	select	
Treeview	close	An item has been closed.
	cut	
	dbselect	
	extevent	
	focus	

Object	Event	Description
	help	
	key	
	open	An item has been opened.
	paste	
	scroll, hscroll	Treeview has been scrolled horizontally.
	scroll, vscroll	Treeview has been scrolled vertically.
	select, activate	An item has been selected.
	select, deactivate	Item has been deactivated (only in case of multiple selection).
Window	activate	
	close	
	deactivate	
	deiconify	
	extevent	
	help	
	iconify	
	key	
	move	
	resize	
	scroll + hscroll	Horizontal scrolling
	scroll + vscroll	Vertical scrolling
	select	Clicking into window without hitting anything else.

Note for IDM on Microsoft Windows

The event "dbselect" is always carried out without "activate" and "deactivate", because they have already been created in the former "select"-event.

Examples for Event Lines

```
on dialog start
{
  program block
  /* execution on DM program start */
}
```

```
on WINDOW close
{
  program block
  /* on closing any window, i.e. on
     activation of close button */
}
```

```
on OK_Button select
{
    program block
    /* on activation of OK_Button */
}
```

```
on Window.title changed
{
  program block
  /* on change of window title */
}
```

```
on Window.title changed
if ( Variable_1 = 12 )
{
  program block
  /* on change of window title
     AND if Variable_1 contains the value 12 */
}
```

2.1.2 Drag and Drop Events

In this chapter the events which are triggered by Drag&Drop operations are described. As with clipboard actions the relevant events are sent to the object. A Drag&Drop operation triggers a cut event at the source object and a paste event at the target object.

Note: The order of the events is arbitrary!

The Drag&Drop here described currently works on Microsoft Windows systems only.

2.1.2.1 Cut Event

On cutting the IDM does not automatically carry out a delete action, but reacts to the cut event. The attributes thisevent.type and thisevent.value are invalid and thus not used.

2.1.2.2 Paste Event

This event triggers a paste reaction.

thisevent.type contains a type_enum and shows the assumed format.

thisevent.value contains the data (DM data type).

When processing, note that .cut_pending = true, if the source object and target object are the same in a move operation (Cut + Paste).

2.1.2.3 Examples

The examples for these events are in directory *…\examples\dragdrop*.

Module with Standard Rules

The module **dnd_defa.mod** contains rules for the standard behavior of several objects.

Currently there are rules for Cut and Paste with the format *type_text* available for the objects ***edittext***, ***listbox*** and ***tablefield***.

Plain Rules

- `rule boolean DnD_default_cut (object Obj input);`
  Cut rule calling the corresponding rule for an object
- `rule boolean DnD_default_paste (object Obj input, anyvalue Event input);`
  Paste rule calling the corresponding rule for an object

Special Rules

- `rule boolean Edittext_cut (object Obj);`
- `rule boolean Edittext_paste (object Obj, anyvalue Type, anyvalue Value, anyvalue Index);`
- `rule boolean Listbox_cut (object Obj);`
- `rule boolean Listbox_paste (object Obj, anyvalue Type, anyvalue Value, anyvalue Index );`
- `rule boolean Tablefield_cut (object Obj);`
- `rule boolean Tablefield_paste (object Obj, anyvalue Type, anyvalue Value, anyvalue Index );`

The return value of all rules indicates whether the processing has been executed successfully.

Dialog “dnd_et.dlg”

This dialog is a Drag&Drop example for an ***edittext***. The rules of the module **dnd_defa.mod** are used.

Dialog “dnd_lb.dlg”

This dialog is a Drag&Drop example for a ***listbox***. The rules of the module **dnd_defa.mod** are used.

Dialog “dnd_tb.dlg”

This dialog is a Drag&Drop example for a ***tablefield***. The rules of the module **dnd_defa.mod** are used.

Dialog “solitair.dlg”

This stand-alone dialog illustrates how to use the Drag&Drop data type *type_object*.

2.1.2.4 Tips & Tricks for Drag & Drop and Clipboard

Cut rules and paste rules should always be able to operate independently, because of the following aspects:

- The source and the target are often different objects which do not necessarily occur in one and the same application.
- Depending on the user input the order of the events may be arbitrary. It is e.g. possible to cut or copy once via keyboard at one and the same object, carry out a Drag&Drop operation and two hours later insert 10x.

As soon as "Cut" is allowed at an object, this operation may be triggered at any time. Before the rule "on Cut" is executed other rules might access the object. To avoid that important information is inadvertently falsified, the attribute *.cut_pending* is set to *true*.

This is why you should note the following when programming:

As long as *.cut_pending = true* no attributes can be changed at the object, i.e. rules, which might be called between the "Cut" triggering and the cut event (e.g. by on focus), cannot change the object per default.

Attention

In a Drag&Drop move operation on one and the same object, the paste event occurs before the cut event, i.e. with *.cut_pending = true*.

If, nevertheless, a rule is to reset attributes you may set *.cut_pending := false*. To inform the following cut rule you should set *.cut_pending_changed := true*.

For the processing of the format DM_Object in the paste rule you should check if the object really is available at that moment.

2.1.3 User Events with Special Inheritance

To process also key events in the ISA Dialog Manager Rule Language, these key events can be queried as follows:

```
on <object ID> key <accelerator>
```

<object ID>
Name of an object, a Model or a Default defined in the dialog.

<accelerator>
Name of an accelerator defined in the dialog.

This event is internally inherited in the IDM according to the following scheme, so that a rule has not to be defined for each object (if, e.g., the key `F1` was pressed).

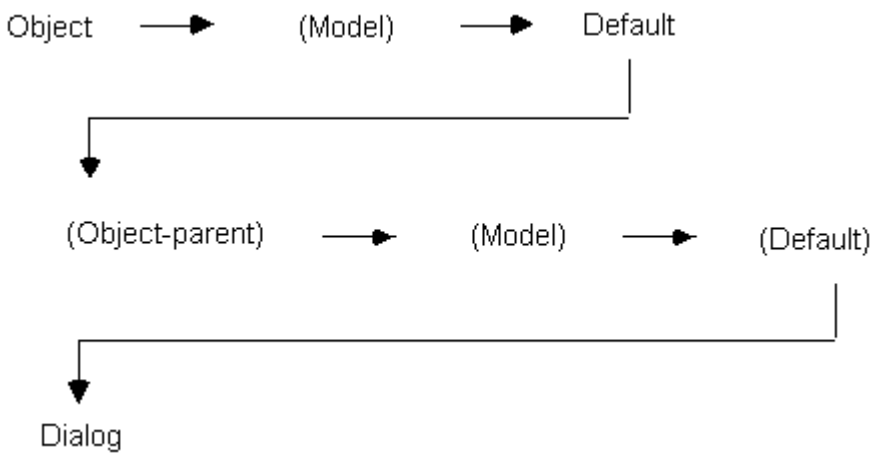

Figure 1: *Inheritance at key/help-rules*

This inheritance is interrupted as soon as one of these objects has defined a rule for this key.

Example

```
dialog Example
accelerator FK5
{
  0: F5;
}

window Window
{
  .xleft  10;
  .width  100;
  .ytop   10;
  .height 200;
```

```
  child pushbutton
  {
    .xleft 10;
    .ytop  10;
    .text  "pushbutton";
  }
}

on Example key FK5
{
  print "Key F5 was pressed";
}
```

The rule is executed as soon as `F5` is pressed in this window.

The same mechanism of hierarchical inheritance is valid for the *help* event, i.e. only one rule is responsible for *help*. This rule is attached to the dialog.

2.2 Internal Events

Events referring to the change of an object attribute or of a variable are called “internal events”.

```
on <object ID><attribute name> changed
```

Please refer to the descriptions in the “Attribute Reference” for the attributes which can be used here.

The following is valid for variables:

```
on <variable name>.value changed
```

See Also

Attribute value

Examples

```
on Variable_A.value changed
{
}
```

```
on Pushbutton.xleft changed
{
}
```

2.3 System Events

Events which refer to the entire IDM program system are called “system events”. These events are used to formulate rules for dialog or module starts and dialog or module ends.

The same applies to the object application.

Start Rule

```
on dialog start
  Event that indicates the start of the IDM program processing.

on module start
  Event that indicates the start of the module processing.

on application start
  Event that indicates the start of an application object processing.
```

Finish Rule

```
on dialog finish
  Event that indicates the end of the IDM program processing.

on module finish
  Event that indicates the end of the module processing.

on application finish
  Event that indicates the end of an application object processing.
```

The following system events are possible:

Event Type	Description
application start	Event which indicates the start of the processing of the "application" object.
application finish	Event which indicates the end of the processing of the "application" object.
dialog start	Event which indicates the start of the dialog processing.
dialog finish	Event which indicates the end of the dialog processing.
module start	Event which indicates the start of the dialog processing.
module finish	Event which indicates the end of the dialog processing.

Examples

```
on dialog start
{
}
```

```
on dialog finish
{
}
```

Typically, the start rule contains statements for initializing data, the finish rule contains statements for a controlled exiting of the IDM (e.g. closing all windows).

The start rule should always be defined; at least one rule containing the keyword **exit** must be defined in addition.

Execution of the rule containing the keyword **exit** calls the finish rule.

2.4 External Events

External events are events that are to be treated equally to the dialog events in the ISA Dialog Manager as concerns their assignment and their processing. However, their source is not controlled by the IDM. One possible source of such events can e.g. be a signal handler which generates this event so that the dialog can respond to it.

An external event is a mixture of dialog event and parametrized rule.

As with parametrized rules, the number of parameters is limited to *16*.

It has to be noted though, that only *14* parameters are usable when used from Rule Language. Therefore it is reasonable to define external events with a maximum of *14* parameters (see also built-in function **sendevent()**).

```
on <object ID> extevent <event ID> ([<data type> <parameter name> <type>])
```

anyvalue <event ID>
: Indicates an identifier that is unique for the respective object and refers to the external event. For instance, a message resource may be used here.

External events are sent to an object and then passed to the object's Default until an object processes the event.

Example

```
on Object1 extevent 4711 (integer ErrCode, string ErrText);
on Object2 extevent EvSave (boolean Success, string Text);
```

The execution of a rule attached to an external event is registered by the application with the interface functions **DM_QueueExtEvent** or **DM_SendEvent**.“Registered” means that the rule is not executed

immediately, but that the external event is queued and then processed according to the dialog event mechanisms.

See Also

C functions DM_QueueExtEvent() and DM_SendEvent() in manual “C Interface - Functions”

3 Rule Processing

3.1 Normal Rules

Normal rules are rules which do not have a further keyword in the event line. These rules have a fixedly defined kind of processing that is described in the following.

These objects to which rules can be defined have an internal hierarchy. A visible object (instance) can be based on a model which itself can be derived directly from the corresponding default. This object hierarchy plays an important role when inheriting attributes or rules. Please note that these hierarchy steps/levels may not be mixed up with the parent-child-relation that usually does not have any influence on the inheritance of attributes and rules.

If normal rules are defined for different hierarchy levels, rules that are defined on a lower hierarchy level will apply, but the rest will be ignored (as is the case with attributes). If rules are defined for an object, the corresponding model or default, these rules will only apply to the object. Those rules which were defined for the model or the default do not apply because they are minor to the object rules. In this case the rules behave identically with the attributes.

Since this behavior is too rigid for the rule processing, additional rules can be executed by using further keywords. Such keywords are described in the following chapters.

3.2 before Rules

With the keyword **before** rules can be marked so that they are carried out in any case before the normal rules. Such a keyword is given after the event line.

Syntax

```
on { <object ID> } <event> { , <event> } before
```

or

```
on { <object ID> } <attribute> changed before
```

<object ID> can **only** be omitted if the rule is defined within an object definition.

When searching for these **before** rules, the IDM starts at the relevant default of the object and then follows the model hierarchy to the actual object. All found **before** rules will be carried out.

3.3 after Rules

With the keyword **after** rules can be marked so that they are carried out in any case after the normal rules. This keyword is given after the event line.

Syntax

```
on { <object ID> } <event> { , <event> } after
```

or

```
on { <object ID> } <attribute> changed after
```

<object ID> can **only** be omitted if the rule is defined within an object definition.

When searching for these **after** rules, the IDM starts at the actual object and then follows the model hierarchy to the relevant default. All found **after** rules will be carried out.

3.4 Processing Order

The different kind of rules are processed in a fixedly defined order which can be described as follows:

If several rules are defined for an object and an event, the order in which the rules have been defined is decisive for the processing. However, this kind of order should only be used if the rules are independent of each other.

If an event occurs, the IDM begins searching gradually for suitable events. In doing so, the IDM searches first for rules beginning with **before** in the corresponding object default. Then the IDM looks for rules belonging to the object models. Finally, the IDM checks at the object itself whether there is a defined **before** rule. In the meantime all found rules are being processed.

Afterward the IDM searches for a normal rule with the object and, if no objects can be found, the IDM advances to the corresponding model. If, however, a rule has been defined for the occurred event, the IDM will not search for a model. At the model the IDM checks whether a suitable rule has been defined. If this is the case, the rule will be processed and the search will be broken up.

Finally the IDM restarts searching for event rules at the object; this time, however, it searches for events marked by the keyword **after**. From the model the IDM advances to the corresponding model or default to search for rules. In doing so, all found rules will be processed.

There is a deviation from this scheme for the rules which are bound to key events (*key*) and help events (*help*). For these events the IDM searches for rules also at the object parent, if no suitable rule could be found during the whole searching procedure. In doing so, a help system in form of a rule can be bound very easily by defining a corresponding rule in the dialog.

The following diagram illustrates the search for rules. In this picture a model directly derived from the default is assigned to the object.

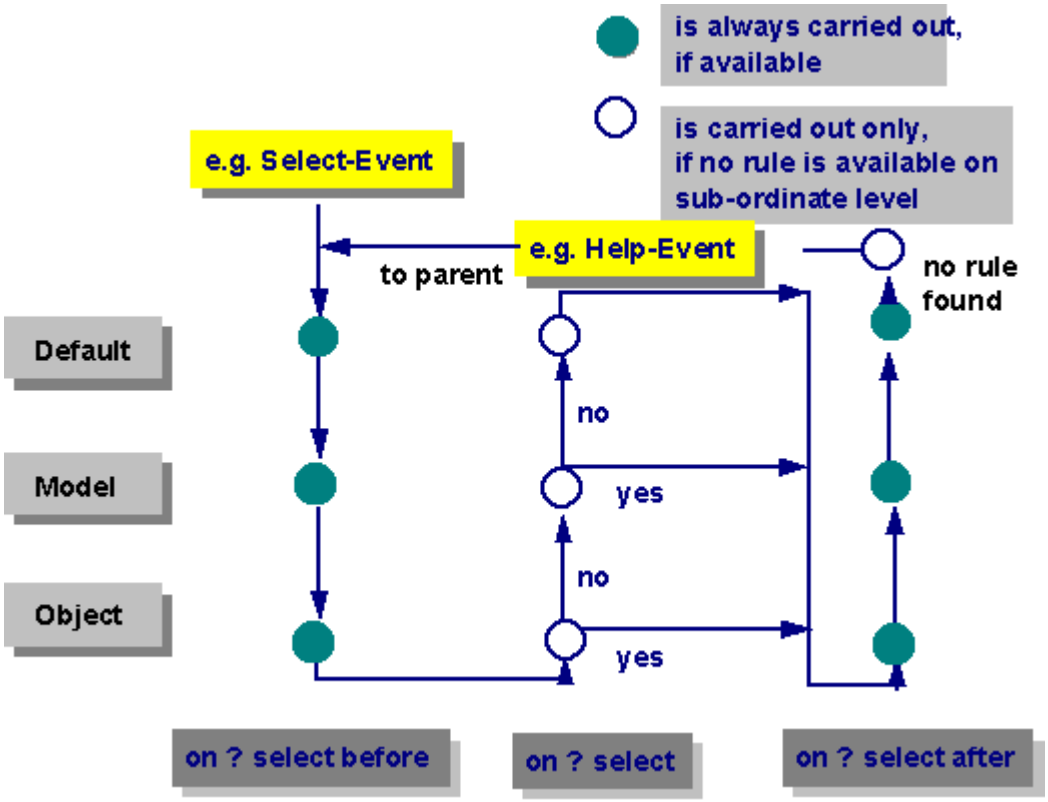

Figure 2: Order of Rule Processing

Example

For a window object the following four rules are defined:

1. `on WINDOW close before`
2. `on Window1 close`
3. `on WINDOW close`
4. `on Window1 close after`

The order thus is: 1-2-4.

Rule 3 will never be processed due to rule 1.

Supposing that for a pushbutton the following five rules have been defined:

1. `on PUSHBUTTON select`
2. `on OKButton select`
3. `on PUSHBUTTON select after`
4. `on OKButton select before`
5. `on OKButton select after`

The order then is: 4-2-5-3.

Rule 1 will never be processed due to rule 2.

As an example we can mention here a system for the data input. In this system many windows are defined which process the contents through an OK button and which ignore the contents by a Cancel button. After pressing either of the two buttons the window is closed.

The rules can be defined so that a rule which closes the corresponding window is bound to the model of the Cancel button. Doing the same for the OK button is more difficult, since a window-specific processing has to come first. The following method can be used:

Each instance of the OK button is bound to a rule which calls the current function and the model of the OK button is bound to an **after** rule which always closes the window. In this way, the closing of the window will be centrally and does not always have to be programmed.

4 Named Rules (Subprograms)

Besides the possibility of binding rules to objects, there is also the possibility to define rules independently of an object. These rules are used like subprograms and can be called from several program places. The IDM can never call this rule by a user interaction, but always by another rule.

These so-called named rules can have up to 16 parameters. These parameters can be considered mere input values, mere output values or input/output values from the perspective of a rule. This has to be defined in the declaration of the parameters.

Instead of the keyword **on** these rules begin with the keyword **rule** to mark that it is a completely different rule type.

Syntax

```
rule <return type> <rule name> ( { <parameter> { , <parameter> } } )
{
  <statements>
}

parameter ::=
  <data type> <parameter name> { := <default value> } { <kind> }
```

<parameter> can occur up to 16 times, i.e. the rule can have up to sixteen parameters.

<return type>

<data type>
All data types that exist in the IDM are available as valid data types for rule parameters and rule return value types. For the return value type there is also the type *void* which marks that the rule does not return anything.

<rule name>

<parameter name>
This has to be a valid identifier.

<default value>
Default values can be defined for input parameters (**input**). These parameters then do not have to be specified when the rule is called.
Parameters with default values must be at the end of the parameter list!

Attention
Optional parameters are not supported by the ***control*** object and the ***message*** resource. It is currently not possible to use default values when programming an OLE Server or an OLE Client with the OLE Interface.

<kind>

- **input** (default)
  only input parameter
- **output**
  only output parameter
- **input output**
  input as well as output parameter

Here input and output is always seen from the perspective of the rule being defined.

Example

```
rule integer Add (integer Arg1 input, integer Arg2 input)
```

This additional rule has two figures as input parameters; two parameters will be added and will have the result "sum of addition".

rule void CheckObjects ()

This defines a rule which is without parameter and which does not have any return value.

5 Special Objects and Object Referencing

In the following chapters you will find the description of how objects can be referenced in the interior of rules. Furthermore, two special objects will be introduced which facilitate the access to objects considerably.

5.1 this

In the ***this*** object available in every rule always the current object which has triggered the rule processing will be saved. In doing so, rules can be bound to models which only affect the current instance of the model. This is why normally all rules defined for models access the current object by ***this*** and start the relevant actions from there. The model will be accessed directly in the rules only very seldom. The same applies to changes to the rules.

Example

A model of a pushbutton will be defined as follows:

```
model pushbutton OK {}
```

The window is to be accessed in a rule. This would look as follows:

```
!! Rule is defined at the model
on OK select
{
  !! Rule accesses the corresponding window
  !! via the instance
  this.window.visible := false;
}
```

If the rule accessed the window via the object name, the reaction would be undefined because the instances are usually made visible individually and they would not inherit the visibility from the model any more.

5.2 Object Referencing

There are two ways to reference objects.

» By unambiguous names, i.e. the object has a clear name or a full path:

Example

```
END_Button
MyWindow.OKButton
```

» By parent specification, i.e. relating the object to a parent or child.

Examples

- `Object.parent`
  addresses the parent of the object;
- `Object.window`
  addresses the window in which the object is located.
- `Object.groupbox`
  addresses the groupbox in which the object is located.

The two possibilities of object referencing described above can be used in the event, condition and action lines, i.e., an object can be referred to with an ambiguous name or with relations.

The term "relations" summarizes the possibilities of referencing the objects among each other. The following keywords are available to form a relation:

- **child[i]**
  Sets the child "i" of an object (window, groupbox, menubox).
- **firstchild**
  Sets the first child of an object.
- **firstmenu**
  Indicates the first menu of an object.
- **groupbox**
  Indicates the groupbox in which the object is located.
- **lastchild**
  Sets the last child of an object.
- **lastmenu**
  Indicates the last menu of an object.
- **menu[i]**
  Requests child "i" in the menu hierarchy of an object.
- **parent**
  Indicates the parent object.
- **this**
  Refers to the object which has triggered the relevant event.
- **window**
  Indicates the next (parent) window higher in the hierarchy.

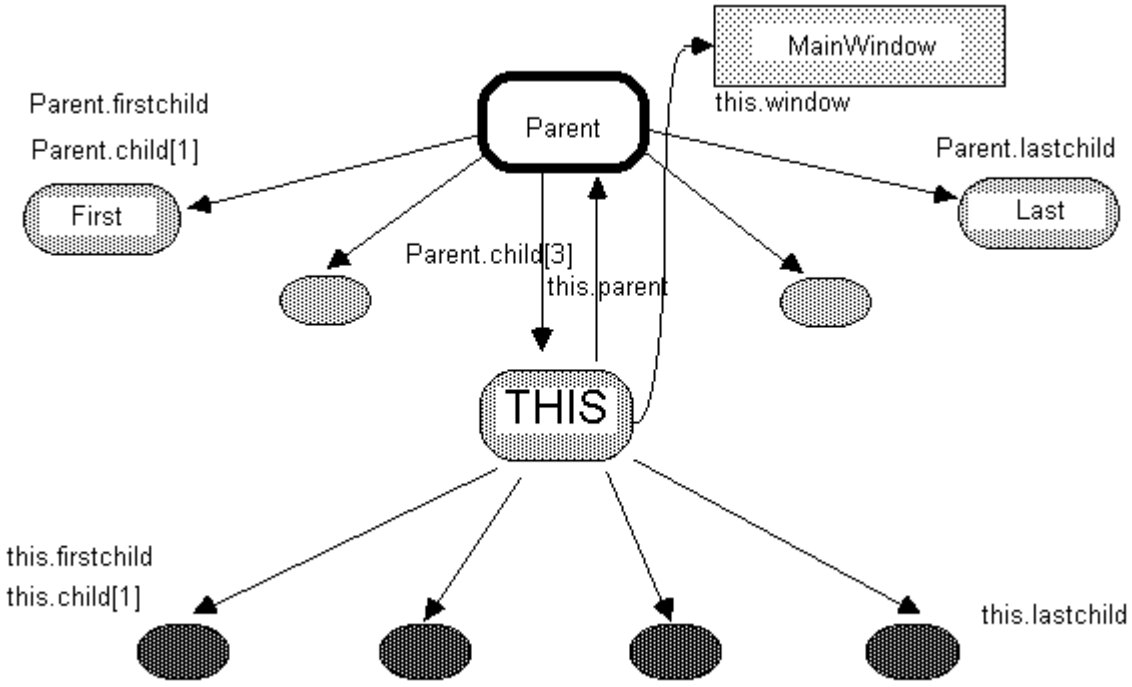

Figure 3: *Reference Options*

To refer to a certain object in the hierarchy, several of the keywords named above can be strung together. At the end of such a referencing action, the attributes of the thus found object can be specified.

```
<Object>::=
  <Objectpath>{<Relation>}[<Attribute>]

<Objectpath>::=
  <Objectidentifier>{.<Objectidentifier>}|
  <Variable>|
  this
```

Example

```
{
  MainWindow.ytop := this.window.ytop + this.window.height;
}
```

Here the window "MainWindow" is positioned at the bottom edge of the window in which the object ***this*** is located.

Note

Use of relations is appropriate only if the general relationships of objects to each other are already known.

If dynamic objects are generated, this way of object referencing is often the best in order to indicate an object.

Forward Referencing

Apart from referencing already defined objects, it is possible at certain positions to define objects after referencing:

- in rules (expressions) and
- in attributes of objects.

5.3 null Object

The ***null*** object is an object with a special identifier, the IDM ID 0. This object can be used to reset resource attributes. It can be returned by DM_SetValue, and can be used for DM_GetValue, or for function arguments.

The ***null*** object is type-independent, which means that there is only one ***null*** object for all resources. The ***null*** object can be used for all data types which are >= DT_instance. In DM_GetValue, the real type of the ***null*** object depends on the attribute type.

Implications

- `object.text := object.accelerator`
  False is returned even if no accelerator is linked to the object, since the text is the type DT_text and the accelerator is of the type DT_accel.
- `object.text := null`
  Eliminates the text in the object.
- Definition:

```
default pushbutton
{
}
model pushbutton PBM
{
  .fgc RED;
}
pushbutton PB1
{
}
pushbutton PB2
{
  .model PBM;
}
pushbutton PB3
{
  .model PBM;
```

```
  .fgc null;
}
```

In this definition, the pushbuttons PB1 and PB3 do not have a color, PB2 has the foreground color RED.
The assignment explicitly prohibits that PB3 inherits the color from its model.

5.4 Event Object thisevent

Apart from the other special objects ***setup*** and ***this***, there is the special object ***thisevent***.

With the help of the keyword **thisevent** you can obtain information on a current event that has occurred. It returns the (virtual) event which possesses various attributes. These attributes can be read.

Attributes of Event Object thisevent

Attribute	Type	Significance	Valid for
.accelerator	object	accelerator identifier	*key*
.attribute	attribute	changed attribute	*changed*
.count	integer	number of timer events that occurred	*select* in timer
.event_code	integer	code of external event	extevent
.eventcount	integer	number of events	all events
.event[EV]	boolean	true if EV is included	all events
.event[I]	event	type of event I	all events
.index	integer	selected item	*select*, *cut*, *paste* in listbox, poptext, treeview
.index	index	selected field	*select*, *cut*, *paste* in tablefield
.type	enum	kind of object affected by Drag-and-Drop operation	*paste*
.value	anyvalue	data that has been moved by an Drag-and-Drop operation	*paste*
.x	integer	x-coordinate of mouse (in pixels)*	*select* in window, groupbox

Attribute	Type	Significance	Valid for
.y	integer	y-coordinate of mouse (in pixels)*	*select* in window, groupbox

* Mouse coordinates are available only in the objects window and groupbox, and are relative to the object to which the event refers.

If an attribute is not valid, it contains *void* data. The existence of such data can be inquired with the built-in function: **typeof**.

A rule can be triggered by several events at the same time. All events are contained in the (indexed) attribute *.event[I]*.

The number of events can be requested with *.eventcount*.

If *.event* is indexed with an integer value, an event is returned that occurred at the same time.

If *.event* is indexed with a value of type *event*, the return value is *boolean true* if the indexed event is one of the events that occurred.

You can enter an arbitrary event for EV; you can e.g. query if there is a special event: *.event[select]*.

6 Data Types

Like in other programming languages, in the Rule Language different data types are defined which are described below.

Data Type	Description
anyvalue	This data type is an arbitrary data type. Only by current allocation it gets its actual data type. In a value defined with this data type everything can be saved. On inquiring it is important which kind of value has been saved last.
attribute	This data type is an attribute of the ISA Dialog Manager. It can be assigned all attributes defined by the IDM like .visible or a user-defined attribute.
boolean	This data type represents a boolean value. Its values are either *true* or *false*.
class	This data type specifies the class of an object or resource. Values include, for example, classes such as *pushbutton, color, function, ...*
datatype	This data type represents a data type. Values such as string, object or integer can be stored here.
enum	This data type is an enumeration type. Here values defined by the IDM are stored. This enumeration type is needed for example by the messagebox, if the user presses one of the offered pushbuttons.
event	This data type represents an event in the ISA Dialog Manager. Here values such as *select*, *dbselect*, *key*and *help* can be stored.
index	This data type can be used for the addressing of two-dimensional attributes such as the contents of a tablefield. In this data type two whole numbers can be stored, which together address a row and a column, i.e. exactly one cell.
integer	This data type represents a whole number (integer). The value range is from -2^{31} to 2^{31}.
method	This data type represents a method in the ISA Dialog Manager. It can adopt values like **:insert**, **:delete**, **:clear** or **:exchange**.
object	This data type represents an object in the sense of the IDM, i.e. everything that has a name or can get a name is an object. This applies to all resources, functions, variables, rules and objects.
pointer	This data type represents an unknown type for the Rule Language. It can be used to notice application-specific data in the dialog without that the dialog having to know the contents of the data.

Data Type	Description
string	This data type represents a character string of any length in the ISA Dialog Manager. The length will be adapted automatically to this type when allocated. In this way, no measures have to be taken by the application..

6.1 Collection Data Types

The following **collection data types** are allowed in the IDM. Please note that only scalar types are permitted for the addressing as well as for the addressed single values. Therefore a structure of multidimensional lists is not possible.

Table 1: *Data types for collections*

Data Type	Meaning
list	This data type defines a list of arbitrary values in the addressing range 0 … 2^{31}. Via *0* the default value is addressed, which is inherited by all list values. This data type is intended and optimized for the manipulation of lists in the Rule Language.
vector	This data type defines a list that contains values of the same type. There is no default value. Therefore, the addressing range is from 1 … 2^{31}. This data type is especially intended for communication with object attributes and is **not** optimized for fast manipulation.
refvec	This data type defines a list that contains only values of type *object*. The addressing range is 1 … 2^{31} and does not provide a default value. Each object ID is only included once, a *null* is not added to the list.
matrix	This data type defines a two-dimensional array whose values can be addressed with the *index* data type. If the specified row (first value of the index) or the column (second value of the index) is *0*, it is a default value that is passed on in the addressing order [0,0] → [row,0] → [0,column] → [row,column].
hash	This data type defines an associative array with any scalar addressing range and arbitrary values and an optional default value that is returned for an access that does not contain a key.

Similar to predefined vector and matrix attributes like *.content[]*, *.userdata[]* etc. an expansion of collections directly after their last element is possible and allowed.

However, it is not possible to use these collection data types for user-defined attributes. User-defined and predefined attributes do not have a unique assignment (attributes exist on an object in **both** a scalar **and** a field variant) and allow for unindexed access to the default value.

6.1.1 Syntax of Expressions

List and Hashes

Collections with the data types *hash*, *list*, *matrix*, *refvec* or *vector* can be defined using the following syntax within the rule code. A definition in the static part of a dialog or module is not possible with expressions, only constant values may be used. Without explicitly specifying a list data type, a list of the data type *list* is created. When the reference operator => is used, a *hash* (associative array) is automatically created, with the expression before => defining the key or index, and the second defining the value.

Attention

For compatibility reasons, index expression takes priority. Thus, to build a two-element list of two integer values, the list data type should be prepended.

Syntax

```
<list> = [ <data type> ] '[' [ <expression> { ',' <expression> } ] ']'
<hash list> = [ <data type> ] '[' [ <expression> '=>' <expression>
    { ',' <expression> '=>' <expression> } ] ']'
```

Examples

```
[1, .xleft, 17+4, Wi.Pb]
["CBS" => 2, itoa(7) => 7]
matrix[ [1,1]=>"first name", [1,2]=>"last name", [1,3]=>"telephone no." ]
```

6.1.2 List Size and Default Values

In principle, each collection data type has a current size, which results from the values set. There are no value gaps in this currently valid indexing range, only unset values (see also the descriptions of **itemcount()**, **countof()**).

In order to achieve consistency with the previous static local variables, the data types *hash*, *list* and *matrix* allow for a default value.

This is displayed at unset value positions within the currently valid indexing range up to the current size. For *hash* data types, if the default value is set, it is returned for all possible indexings.

The sequence of access to the default values for an unset value at the position *<row>,<col>* in a matrix *M* is *M[<row>,<col>]* → *M[0,<col>]* → *M[<row>,0]* → *M[0,0]*.

The access sequence to the default value for an unset value at *<index>* in a list *L* is *L[<Index>]* → *L[0]*.

6.1.3 Access and Assignment of Collections

The indexed values (items) of a collection are accessed via the [ ] operation, just as known for local variables or predefined and user-defined attributes.

Without the [ ] operation, the entire value is fetched or set.

Attention

It should be noted that accessing a user-defined or predefined attribute without the [ ] index fetches or sets the default value or scalar attribute. To fetch or set all values of an attribute, the functions **getvector()** or **setvector()** should be used.

Examples

```
variable list Primes := [2,3,5,7,11,13];
variable vector[integer] Numbers;
variable hash Months := [1=>"Jan", 2=>"Feb", 3=>"Mar"];

print Primes[1];
Months[4] := "Apr";
Numbers := Primes;
Primes[0] := -1;
Primes := [17, 19, 23];
```

Output of collections via **print** or **sprintf()** is also possible. However, no default values are output. For the data types *hash* and *matrix* the output is including the index.

Specific Features of *refvec*

It should also be noted that the *refvec* data type is handled in a special way because it ensures uniqueness of the values and does not store *null* values.

With an indexed assignment of an already existing object ID, the object ID at this position is replaced and afterward uniqueness is ensured, which may also result in a shift of the already existing object ID to another position and a reduction of the *refvec* list.

In case of an indexed assignment of a *null*, the object ID is deleted from the *refvec* list and the following object IDs advance, which in turn results in a reduction of the *refvec* list.

6.1.4 Automatic Conversion

In assignments or parameter calls, collections are automatically converted to the required list type. All values without index are copied in their “natural” order, including the default values if possible, e.g. *[0]* for a *list* variable will be passed on if the target type also has a default value.

Exception

When assigning a *hash* with *index* data type to a *matrix*, the index will be taken over. With the reverse assignment of a *matrix* to a *hash*, the index will also be transferred. To prevent this, the built-in function **values()** should be used.

6.1.5 Performance

In general, it should be considered that the use of collections (entire values) within expressions initially requires the values to be copied, but then in reading operations only references of this list are used, i.e. without further copying. This ensures the necessary performance when working with collections. Passing on to application functions or manipulation by built-in functions, however, requires another copy action, possibly with a code page conversion of string values.

For this reason, it should be kept in mind that complex expressions that process collections may well temporarily lead to large amounts of data. Coding and splitting of expressions should take into account the maximum list sizes.

6.1.6 Local and Global Variable

The collection data types are available for local and global variables. It is also possible to initialize a global collection with constant values.

Until now, local, static variables could also be an array or an associative array, which only could be initialized in a very limited way. Collection data types are now allowed for both local and local static variables. The initialization expression is no longer restricted.

Attention Behavior Change

The existing notation of arrays and associative arrays for static local variables is still allowed and mapped to the data types *list* or *hash*. The static initialization via `.<identifier>[<index>] := <value>;` after the variable part is **no longer possible** to ensure consistency with other local variables! The initial value sets the entire value of the variable, not just the default value!

Example

```
dialog D

variable hash Prices := [ "iMac" => 1300, "Samsung Tab" => 300 ];
variable vector[string] Weekdays := ["Mon","Tue","Wed"];

on dialog start
{
  variable string Stations[integer] := [1=>"ABC", 2=>"CBS"];
  variable hash Stations2 := [3=>"NBC", 4=>"HBO"];
  static variable list AllStations := join(Stations,Stations2);
  /* No longer allowed:
   * variable string Stations[integer] := "UNKNOWN";
   * .Stations[1] := "ABC";
   * or
   * static variable integer AssArray[string] := -1;
   * Instead possible:
   * static variable integer AssArray[string] := [nothing=>-1];
   */
```

```
  exit();
}
```

6.1.7 Relational Operators

For collections, only the relational operators = and <> are allowed.

Equality prevails under the following conditions:

1. The data type must be the same.
2. All contained values or index/value pairs must be equal, including the default values.
3. The number of values contained and their positions must be the same.

Comparison between a string and a text ID is done on a string basis.

When using relational operators, particular attention should be paid to the data type.

Example

```
dialog D
window Wi {
  listbox Lb {
    .content[1] "Sat";
    .content[2] "Sun";
  }
}
on dialog start {
  variable list WeekendDays := ["Sat", "Sun"];
  variable hash DayNumber := ["Sat"=>6,"Sun"=>7];
  variable list Days;

  print WeekendDays = getvector(Lb, .content);
  print vector["Sat","Sun"] = getvector(Lb, .content);
  print values(WeekendDays) = values(getvector(Lb, .content));
  print WeekendDays = keys(DayNumber);
  Days := WeekendDays;
  WeekendDays[0] := "??";
  print Days=WeekendDays;
  exit();
}
```

Output

```
false => Different data types (list<>vector)
true
true
true
false => WeekendDays has a default value, but Days does not!
```

6.1.8 Uninitialized variables

As before, the IDM behaves differently when accessing uninitialized values. If an uninitialized local or static variable is accessed, *nothing* is returned. When accessing an uninitialized global variable (equivalent to a variable object) or a user-defined attribute, access is denied with a *“cannot get value”* error.

7 Functions

Functions are the interface to the used programming language. The keyword is **function** followed by the function type and the function identifier. As to function prototypes the data types of the parameters have to be specified instead of the function arguments. In this definition the functions have to be distinguished according to their later use.

Basically, several different kinds of functions can be distinguished here:

- Function called in the rules:
  This function type can have up to 16 arbitrary parameters, which have to be specified correspondingly in the function declaration.
- Function connected to an object (callback function):
  This function, which is specified directly in the object definition by *.function*, has to be defined as a callback function. The IDM defines which parameters this function will get. This is why they must not be specified in the declaration. In addition, the definition of this function specifies the user events for which the function is to be called.
- Function connected to a "canvas" object (canvas function):
  Since this object is a significant exception in the IDM, and has to be treated differently by the application, there is a special function type `canvasfunc` for it. Its function parameters are also predefined by the IDM and must not be given in the declaration.
- Data function:
  This function can be specified in the *.datamodel* attribute of all objects that also support user-defined attributes. A data function can serve as a Data Model (Model component) to provide the presentation objects with data values. The parameters of a data function are predefined by the ISA Dialog Manager and cannot be changed by the user.
- Format function:
  This function is indicated when defining format resources as well as at editable texts and tablefields (attribute .formatfunc). The function parameters are predefined by the IDM and must not be defined in the declaration.
- Reloading function for tablefields:
  This function is indicated in the attribute .contentfunc when defining tablefields. The function parameters are predefined by the IDM and must not be defined in the declaration.

Permitted languages are:

- C
- COBOL

If no language is defined, *C* is chosen as default language.

Permitted data types are:

Data Type	Value Range
anyvalue	Arbitrary, undefined data type; will be defined only by actual occupation.
attribute	Attribute data type in the IDM.
boolean	Boolean value (true | false).
class	Class of an object, e.g. "window".
datatype	IDM data type.
enum	Symbolic constant; enumeration type for special attributes, e.g. "sel_row".
event	Type of event, e.g. "select".
index	Index value for 2-dimensional attributes [I,J].
integer	Integral number ("long integer"), e.g. *42*.
method	Method, e.g. "delete".
object	Object in the sense of the IDM (actual objects, resources, functions, rules,…).
pointer	Pointer from the application.
string	Character string which can be arbitrarily long (zero terminated), e.g. "hello".
void	Data type without value.

Note for IDM on Microsoft Windows

The IDM data type *integer* corresponds to the data type *long* in the C Interface.

The ISA Dialog Manager uses the **Pascal Calling Convention** as default. It is advisable to have the IDM write and integrate the function prototypes.

7.1 Function Called in Rules

These functions can be explicitly called up from the rules. They must have one of the permitted function types and are parametrizable with a maximum of 16 arguments. The data type of any argument can be indicated in the function braces. If there is more than one argument, they have to be separated with a comma.

Syntax

```
{ export | reexport } function { <language> } <data type> <function name>
  ( { <parameter> { , <parameter> } } ) ;

language ::= c | cobol
```

```
parameter ::=
  <data type> { [ <size> ] } { <parameter name> { := <default value> } }
  { input } { output }
```

Within this declaration of the function parameters, the parameter values can be defined. There are three possibilities:

- Only **input** value
  If a parameter shall serve only as input value in a function, the keyword **input** is optional after the definition of the data type, because this type is the default. This parameter can only be read in the function by this declaration.
- Only **output** value
  If a parameter shall serve only as output value of a function, the keyword **output** has to be added after the definition of the data type. This means that the parameter has to be set in the application and can afterward be processed in the IDM.
- Input value as well as output value (**input** **output**)
  If a parameter shall serve as input value in a function and simultaneously as output value of this function, the keywords **input** and **output** have to be added after the definition of the data type. With this declaration, you have the possibility to specify an attribute value in a function as input parameter, to manipulate the value there, and to display the result of this manipulation in the relevant attribute.

Parameter of functions can also be defined by names, e.g. to be able to see for what the individual parameters are used.

Example

```
function c boolean ReadData(string Filename,
                integer Index,
                string Value output);
```

It is also possible to define default values for arguments of type input. These arguments are optional and need not be stated when calling the function.

Note

Arguments with default values must be at the end of the parameter list.

The length of a parameter can also be indicated to the system. This is especially important if a character string shall be passed to an application, e.g. with COBOL. The size has to be given in brackets ([<size>]).

7.1.1 Alias Name with Optional Code Page Definition

Availability

Since IDM version A.06.01.d

When defining application functions, the keyword **alias** followed by a string can be specified after the function parameters.

Syntax

```
{ export | reexport } function <language> <return_data_type>
    <function_identifier> ( { parameter [ , parameter ] } )
    alias <alias_string>  ; | <code_block>

alias_string ::=
    "<alias-name>{ ;<code_page> } | <code_page>"
code_page ::=
    [CP=]<code_page_identifier>
```

The alias specification is used to provide a correct function name for dynamic binding (i. e. without the restrictions that apply to function identifiers).

In the alias string a function-specific code page for string processing can be specified as well. This has to begin with “CP=”, immediately followed by the code page identifier (analog to the CP defines from **IDMuser.h**, but without the prefix “CP_”).

If an alias name is defined in addition, the code page specification must follow it, separated by a semi-colon (“;”).

Examples

```
function integer Atoi (string S) "myatoi;CP=utf8";
function string CurTime() "CP=utf16b";
```

7.1.2 Calling Parameters of a Function during Runtime

It is possible to query the parameters of a function also during runtime. For this purpose the attributes .type for the data type and .input for the input value or .output for an output value have to be specified each with the index of the parameter to be queried. If the relevant parameter is of the type string, you can query the size of the parameter with the attribute .size.

The following attributes are valid for a function:

Attribute	Description
.count	Number of parameters.
.language	Programming language in which function is to be written.
.type	Data type of the return value of function.
.input[I]	Parameter I is input value.
.output[I]	Parameter I is output value.

Attribute	Description
.size[I]	Size of string parameters (only relevant for COBOL).
.type[I]	Data type of I-th parameter.

Example

```
dialog Test
{
}

function void MyTestFunc(integer A input output,
    string[20] B output, boolean C input,
    pointer D input output);

on dialog start
{
    variable integer I;
    print MyTestFunc.count;
    print MyTestFunc.type;
    print MyTestFunc.language;
    for I := 1 to MyTestFunc.count do
    print MyTestFunc.input[I];
    print MyTestFunc.output[I];
    print MyTestFunc.size[I];
    print MyTestFunc.type[I];
    endfor
}
```

results in the following tracefile output

```
[XR] rule on dialog start (this=dialog Test)
  4
  void
  "default"
  true
  true
  0
  integer
  false
  true
  20
  string
  true
  false
  0
  boolean
```

```
  true
  true
  0
  pointer
[XD] rule on dialog start
```

7.2 Callback Function

A callback function is called when an object to which the function is attached receives an event which was specified when defining this callback function. The callback function is called before the event rule processing. The corresponding event rules are only processed if the callback function returns the value *true*.

Syntax

```
{ export | reexport } function { <language> } callback <function name> ( )
    for <event> [ , <event> ] ;
```

Callback functions are later linked to an object in the object definition with the attribute *.function*. The definition of the function specifies which user events trigger calls of this function, e.g. *select*, *activate*, *deselect* or *charinput*.

The parameters of the function are predefined by the IDM and cannot be changed.

Example

```
function callback ObjectCall () for close, move, activate;

window TestWindow
{
  .function ObjectCall;
  .xleft 17;
}
```

This function is called when the user is closing, moving or activating the window.

See Also

Chapter “Object Callback Functions” in manual “C Interface - Basics”

7.3 Canvas Function

Canvas functions serve for processing of special canvas events and allow the display of graphics within a canvas. The canvas events and how the graphics are displayed in the canvas are window system-dependent.

Syntax

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

These functions can be indicated when defining canvas objects with the attribute *.canvasfunc*.

The parameters of the function are predefined by the IDM and cannot be changed.

Example

```
function canvasfunc HandleCanvas ();

window CanvasWindow
{
  .xleft 17;

  child canvas Dynamic
  {
    .canvasfunc HandleCanvas;
   .width 100;
  }
}
```

See Also

Chapter “Canvas Functions” in manual “C Interface - Basics”

7.4 Data Function

A data function can serve as a Data Model (Model component) to provide the presentation objects with data values. The function is called when data values shall be synchronized, i.e. either to retrieve data values from the Data Model or to assign them.

Syntax

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

These functions can be specified in the *.datamodel* attribute of all objects that also support user-defined attributes.

The data function can be implemented in C/C++, as of IDM version A.06.01.d COBOL is also supported as application language with the COBOL interface for MICRO FOCUS VISUAL COBOL.

The data function may be implemented on the DDM server side.

The parameters of the function are predefined by the IDM and cannot be changed.

See also

Chapter “Data Functions” in manual “C Interface - Basics”

Chapter “Data Functions” in manual “COBOL Interface”

Chapter “Datamodel” in manual “Programming Techniques”

Attributes .dataget[attribute], .datamodel[attribute], .dataoptions[enum], .dataset[attribute]

7.5 Format Function

Format functions serve the formatting of edittext and tablefield contents.

Syntax

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

These functions can be specified when defining format resources. When defining edittexts and tablefields they can be specified with the attribute *.formatfunc*.

The parameters of the function are predefined by the IDM and cannot be changed.

See Also

Chapter “Format Functions” in manual “C Interface - Basics”

7.6 Reloading Function

Reloading functions can be used for dynamic reloading of tablefield contents and are called by the IDM when a user is scrolling in a tablefield in such a way that the area visible after the scroll action contains rows and columns without contents. The reloading function can then load the missing contents into the tablefield and, if necessary, delete contents which are not in the visible area and which are not needed anymore.

Syntax

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

These functions can be given when defining tablefields with the attribute *.contentfunc*.

The parameters of the function are predefined by the IDM and cannot be changed.

See Also

Chapter “Reloading Functions” in manual “C Interface - Basics”

7.7 Simulation of Functions

Functions which are not linked to the ISA Dialog Manager can be simulated by rules. When large applications are developed the programmer of the graphic user interface often cannot dispose of all functions he needs for the process or simulation of his surface. He can only test parts of his

application. To make the dialogs independent of the missing functions, these functions can be simulated by rules. Functions which are bound to the Dialog Manager are called as usual.

The following functions can be simulated:

- functions called out of rules
- callback functions

On defining the function you usually also indicate the rule. Instead of concluding the definition with a semicolon, you type an opening brace to define the simulation rule. The simulation rule has the same attributes like a named rule.

Example of a function definition without simulation rule

```
function c boolean CheckOnline();
```

Example of the same function with simulation rule

```
function c boolean CheckOnline()
{
  return( true );  // always online simulation
}
```

The simulation rules can be used similarly to the named rules. They have the same parameters as the simulated function.

Example

```
function c void ConvertDate(boolean CurrentDate input,
    string Date input output)
{
  // Date will be in format YYYYMMDD and we are to lazy to do
  // this here, it will be sufficient for us to do this with
  // two different dates
  if ( CurrentDate ) then
    Date := "11.11.2011";    // simulate always with this
                             // date as current
  else
    Date := "1.4.2010";      // no conversion supplied,
                             // this is sufficient
  endif
}

function cobol boolean GetExchangeRate( string From input,
    string To input, string ExchangeRate output)
{
    case From
    in "EUR":
        case To
        in "EUR":    ExchangeRate := "1.000";
```

```
        in "USD":    ExchangeRate := "1.3249";
        in "GBP":    ExchangeRate := "0.8550";
        ...     // many more currencies
        endcase
    in "USD":
        case To
        in "USD":    ExchangeRate := "1.000";
        in "EUR":    ExchangeRate := "0.7548";
        ...
    ...
    endcase
    return (true);
}

record Currency
{
    string Which;
    string EUR;
    string USD;
    string CHF;
    ...
}

function cobol boolean GetCurrencyRates(record Currency
    input output)
{
    case Currency.Which
    in "EUR":
        Currency.EUR := "1.000";
        Currency.USD := "1.3249";
        ...
    in "USD":
    ...
}
```

This simulation should suffice most of the applications. In most cases, a complete substitution of the function is not necessary, since it is enough to control the dialog process usefully.

The ISA Dialog Manager enables the developer to specify directly the functions for events. These are called directly on occurrence of the specified event. For these callback functions you can also indicate simulation functions. In doing so, you can program a useful function simulation for the dialog simulation and the testing procedures. The simulated callback function has the same attributes as an event rule. The accompanying event rules, if there are any, are always executed after the simulated callback function.

Example

```
function callback PushbuttonSelect() for select, focus
```

```
{
    if ( thisevent.event[select] ) then
        // the simulation code for select
    endif
    if ( thisevent.event[focus] ) then
        // the simulation code for focus
    endif
}

default pushbutton
{
    .function PushbuttonSelect;
}

on PUSHBUTTON select
{
    // is also executed
}
```

The developer can control the dialog and react to the events without having to implement additional event rules.

7.8 Functions in modularized dialogs

In order to use functions in modularized dialogs the attributes *.application* on the ***import*** object and *.masterapplication* on the ***module*** or ***dialog*** object are available. Thus all functions of a module can be assigned from "outside" to a special application. The procedures for both variants - for import and for use - are described in chapter “Programming Techniques” / “Modularization” / “Object Application”.

8 Global Variables

A variable declaration consists of the keyword **variable**, the data type, the variable identifier and the terminator **;**. In one declaration more than one variables of the same data type can be defined. In this case the identifiers of the variables have to be separated by **,** (commas).

If a variable is declared and not within a rule or method (see also chapter “Local Variables”), it is called a **global** variable.

In IDM **global variables** are treated like objects so that not only the content of a variable but the variable itself can be changed at runtime (see chapter “Attributes of Global Variables that Can Be Changed Dynamically”).

8.1 Variable Definition

Syntax

```
{ export | reexport } variable <data type> <variable name>
  [ , <variable name> ] ;
```

The following data types are available:

Data type	Value Range
anyvalue	Arbitrary, undefined data type; will be defined only by actual occupation.
attribute	Attribute data type in the IDM.
boolean	Boolean value (true | false).
datatype	IDM data type.
class	Class of an object, e.g. "window".
enum	Symbolic constant; enumeration type for special attributes, e.g. "sel_row".
event	Type of event, e.g. "select".
index	Index value for 2-dimensional attributes [I,J].
integer	Integral number ("long integer"), e.g. *42*.
method	Method, e.g. "delete".
object	Object in the sense of the IDM (actual objects, resources, functions, rules,…).
pointer	Pointer from the application.

Data type	Value Range
string	Character string which can be arbitrarily long (zero terminated), e.g. "hello".
void	Data type without value.

8.2 Variable Definition with Initialization

The following definition initializes the generated variable immediately with a value. The data type of the value has to be identical with the data type of the variable.

Syntax

```
{ export | reexport } variable <data type> <variable name> := <value>
  [ , <variable name> := <value> ] ;
```

Object references, numbers, colors, strings, and boolean values, i.e. anything which is supported by the ISA Dialog Manager, can be stored in variables of the type *anyvalue*.

Example

```
variable integer Filling_level;
variable boolean Valve_state := true;
variable string  Valve_identifier := "main valve";
variable enum    Mode;
variable object  PbSwitch;  // Let PbSwitch be a pushbutton
```

A character string is always in quotation marks.

```
Valve_identifier := "main valve";
```

In this example the words "main valve" are a string.

8.3 Configurable Variables

The additional keyword **config** marks a global variable as configurable, i.e. its value can be set in a configuration file.

Syntax

```
{ export | reexport } { config } variable <data type>
  <variable name> { := <value> } [ , <variable name> { := <value> } ] ;
```

The configuration file is loaded with the functions **loadprofile()**, DM_LoadProfile or DMcob_LoadProfile.

See Also

Attribute .configurable

Chapter “Configuration File” in manual “Development Environment”

8.4 Protecting Variables against Changes

Changes to the content of a global variable can be prevented by using the keyword **constant** instead of **variable** in the declaration or by setting the attribute *.constant* of the variable to *true*.

Syntax

```
{ export | reexport } constant <data type> <variable name> := <value>
  [ , <variable name> := <value> ] ;
```

See Also

Attribute .constant

Example

```
dialog D

variable integer V := 123;
constant integer C := 456;

on dialog start
{
  C:=V;      // Evaluation error because C cannot be changed
  V := 234;  // OK
  V.constant := true;
  V := 345;  // Error – Variable is write-protected now
}
```

8.5 Attributes of Global Variables that Can Be Changed Dynamically

The table below shows those attributes of variables, that can be queried and changed dynamically at runtime.

Attribute	Value Range	Meaning
.value	anyvalue	Value
.type	datatype	Data type
.constant	boolean	Changeability of content

In order to refer to the variable itself and not to its content the attribute *.self* is used. This is followed by the attribute that shall be addressed.

Example

```
variable integer I := 2;

print I.self.type    // integer
print I.self.value   // 2

I.self.type := boolean;
print I.self.type;  // boolean
```

Note

This applies to global variables only and not to local variables (i.e. variables declared within rules and methods).

9 Validity Range for Better Type Checking

User-defined attributes, variables, local variables (in rules), parameters and return types of rules, user-defined methods, *extevent* rules and application functions can be enhanced with a range of validity. This has been introduced so that a stricter type checking can take place during the initial loading process enabling errors in the dialog to be detected early on.

Sample use case: restriction of attributes or variables to objects that are derived from a certain model.

The validity range is an add-on to a "value container" e.g. an attribute or a variable (incl. parameters) and does not form a type of its own. A defined range of validity ensures that only settings that fit within this range are allowed – so the "value container" always has a consistent value. The corresponding tests for this happen during the loading and execution of rule code.

Secondly, the range of validity can also limit further access from the "value container". Access in this sense refers to the access to a child, attribute or method. Such access can lead to the return of a value with a derived range of validity.

Example

```
dialog D
// Basic model with a child
model window MWi {
  statictext StHeader {}
}
// Derived and expanded model
MWi Wi {
  .StHeader {
    integer Width := 10;
  }
  edittext Et {
  }
}
// A second model
model window MWi2 {}
// Entity from the second model
MWi2 Wi2 {
  // Attribute that can only be contained in MWi2-derived objects
  object[MWi2] Ref := Wi; // Syntax error due to violation of validity range
}
// Rule that can only be used on objects derived from MWi
rule void SetHeader(object[MWi] O, string Header) {
  O.StHeader.text := Header;  // O can never be Wi2!
  O.Et.xauto := 0; // Syntax error due to access violation
  O.StHeader.Width := 20; // Syntax error due to access violation
```

```
}
on dialog start {
  variable object[MWi] O; // Variable only for MWi-derived objects
  SetHeader(Wi,"First");
  SetHeader(Wi2,"Second"); // Syntax error due to violation of validity range
  O := D.child[2]; // Dynamic violation of the validity range
}
```

9.1 Restrictions

The following restrictions are currently possible for the data types listed below

Data Type	Validity Value	Value and Access Restrictions, Derivation of the Validity Range
string	integer value (>0)	No restrictions. As before this validity value serves to represent the string size that is necessary for the generation of trampoline data for COBOL.
object	object	The value can be *null* or it must be identical with the validity value or rather be derived from it (comparable to **:instance_of** method). Access restrictions to user-defined attributes, children and methods that exist on the object given as validity value. Derivation of the validity range when accessing a child object.
object	class	The value can be *null* or it must be an object of the corresponding class. No access restrictions.

A validity range can be defined in the following cases:

1. Data type of a user-defined attribute
2. Index data type of a user-defined associative field
3. Data type of a global variable
4. Return type of an application function, named rule or user-defined method
5. Parameter types of application functions, named rules and user-defined methods as well as event rules
6. Data types of local variables in named rules and user-defined methods as well as event rules

Example

```
dialog D
model window MWi {}                                    // Following numbers
MWi Wi {                                               // refer to the
record Rec {                                           // previous list
  object[MWi] Ref := Wi;                               // 1)
  boolean AssArray[object[MWi]];                       // 2)
  .AssArray[Wi] := true;
}
variable object[MWi] GlobalVariable := Wi;             // 3)
function object[color] GetBgc(object[MWi] Window);     // 4), 5)
rule object[MWi] GetModel(object[Wi] P) {              // 4), 5)
  return P.model;
}
on Wi extevent 123 (object[MWi] P) {                   // 6
  variable object[MWi] V := P;
}
```

9.2 Access and Value Tests

Static definitions containing validity ranges are always tested during the loading process. This is also true for assignments and accesses in rules that are recognizable as constants. In this sense object paths (e.g. "Wi" in the example below) are to be seen as a "constant" value that can be tested during the loading process.

A dynamic test always takes place in references to value containers with a set validity range or rather in value calls or accesses from within these, and as a result create a **Fail** and can be intercepted within the parenthesis of a **fail()** construct. Call parameters and returns of user-defined methods are possibly only dynamically tested. This is due to the validity information not necessarily being available at loading time when using attributes and methods of imported objects.

As a rule predefined "variables" such as `this` or `thisevent` are not bound to a validity range. This is also true for return types and parameters of predefined attributes, methods and built-in functions. Therefore the usage should be carried out with "special cautiousness".

Since the validity range has no type of its own, no type checking or type conversion takes place. A "casting", as is used in other programming languages, is not necessary most of the time. This can take place via the intermediate storage on a variable of the data type *object*.

The access testing only takes into consideration the actual status (objects, existing children and attributes).

The following short example contains correct and incorrect applications and shows the expected time at which the test is carried out (load time or run time).

```
dialog D
:
model window MWi { child edittext Et{} }
MWi Wi { child pushbutton Pb {} }
```

```
model window MWi2 { }
MWi2 Wi2 { checkbox Cb {} }
:
variable object[MWi] I1 := Wi
variable object[MWi2] I2 := Wi2
variable object O;
:
O := I1;  // Validation test not necessary
I1 := O;  // Validation test during loading process
I2 := I1; // Validation test during loading process – VALIDATION ERROR!
I2 := W1; // Validation test during loading process – VALIDATION ERROR!
:
// Valid/allowed access to child object:
print I1.Et;
print O.Pb;

// ACCESS ERROR to child object and its test time period:
print I1.Pb;      // Access test during loading process because I1 possesses
validity range
print O.Unknown;  // Access test during loading process
print I2.Cb;      // Access test during loading process
print I2.Unknown; // Access test during loading process
print getvalue(I2,. Unknown); // Access test during loading process
```

9.3 Derivation of the Validity Range

The validity range is derived further when children and attributes are accessed.

This happens when an attribute with a validity range is accessed. This is also true when access to a child object occurs from a value container with a validity range.

```
dialog D
model window MWi {
  child groupbox Gb {

    child edittext Et {
      rule void Apply() {}
    }
  }
}
MWi Wi {
  object[MWi] Ref := MWi;
  .Gb {
    checkbox Cb { rule void Apply(); }
  }
  rule void Apply() {
    variable object[MWi] This := this;
    This.Gb.Et:Apply();
```

```
    This.Gb.Cb:Apply();     // Access error because Cb is not in MWi.Gb
    this.Ref.Gb.Cb:Apply(); // Access error because Cb is not in MWi.Gb
  }
}
```

9.4 Enhancement of the IDM Syntax

Syntactic changes to the Rule Language are as follows (highlighted red):

General Definitions

```
<DataType> ::= anyvalue | attribute | boolean | class | enum | event |
       index | integer | method | object | pointer
<ParameterType> ::= { input } { output }
<ReturnType> ::=  void | <DataType>
<Validity> ::= '[' <ValidityValue> ']'
<ValidityValue> ::= <Class> | <Object> | null | <IntegerValue>
<Object> ::= <Identifier> [ .<Identifier> { :'[' <IntegerValue> ']' } ]
```

Definitions for Named Rules and Methods

```
<Rule> ::= { export | reexport } { public } { extern } rule
       <ReturnType> { <Validity> } <Identifier>
       '(' { <Parameter> [ , <Parameter> ] } ')' { <ReposId> }
       <OptionalProgramBlock>
<Parameter> ::=
       <DataType> { <Validity> } <Identifier> { := <Value> } <ParameterType>
<OptionalProgramBlock> ::= ';' | <ProgramBlock>
```

Definitions for External Events

```
<ExternalEventRule> ::= on <Object> extevent <ConstantValue>
       { '(' { <Parameter> [ , <Parameter> ] } ')' }
       { <RuleCondition> }
       <ProgramBlock>
```

Definitions for Local Variables

```
<VariableStatement> ::= { static } variable <Variable> [ ',' <Variable> ] ';'
<Variable> ::= <DataType> { <Validity> } <Identifier> { := <Value> }
```

Definitions for Global Variables

```
<GlobalVariable> ::= { export | reexport } { config } <VariableType>
<DataType>
       { <Validity> } <Identifier> { <ReposId> } { := <Value> } ';'
```

```
<VariableType> ::= constant | variable
```

Definitions for User-defined Attributes

```
<AttributeDefinition> ::=
        <DataType> { <Validity> } <Identifier> <AttributeType> ';'
<AttributeType> ::= <ScalarAttribute> | <IndexedAttribute> |
        <AssociativeAttribute> | <ShadowAttribute>
<ScalarAttribute> ::= { := <Value> } ';'
<IndexedAttribute> ::= '[' <IntegerValue> ']' { := <Value> }
<AssociativeAttribute> ::= '[' <DataType> { <Validity> } ']' { := <Value> }
<ShadowAttribute> ::=
        shadows { instance } <Object> <Attribute> { '[' <Index> ']' }
```

Definitions for Application Functions

```
<ApplicationFunction> ::= { export | reexport } function { <Language> }
        <ReturnType> { <Validity> } <Identifier>
        '(' { <FunctionParameter> [ , <FunctionParameter> ] } ')'
        { alias <String> } { <ReposId> } <OptionalProgramBlock>
<FunctionParameter> ::= <Parameter> | <RecordParameter>
<RecordParameter> ::= record <Object> { := <Value> } <ParameterType>
```

9.5 Attributes in Relation to Validity Ranges

The following attributes exist for querying validity ranges dynamically. Further information can be found at the respective attribute descriptions in the “Attribute Reference”.

- scope[attr]
  Returns the validity range of user-defined attributes.
- indexscope[attr]
  Queries the validity range for the index of user-defined associative attributes (associative arrays).
- typescope
  Retrieves the validity range for return types of user-defined functions and rules.
- typescope[l]
  Queries the validity range for the parameters of user-defined functions and rules.

10 Actions in the Program Block

The functionality of the rules is described in the program block, stated in braces ({ }), following the event line. With this rule component, the entire dialog execution including its functionality can be described by the DM.

Actions can be e.g

- changing of attributes or attribute values
- calling of application functions
- calling of built-in functions
- "if-then-else-endif" constructs
- sub-rules
- variables.

The program block contains assignments of any type (logical, arithmetic and strings), calculations and mechanisms for the conditional execution of commands.

Example

```
on PB_Show select                       /* 1 */
{                                       /* 2 */
  if ( MainWindow.visible = false )     /* 3 */
  then                                  /* 4 */
    MainWindow.visible := true;         /* 5 */
  endif                                 /* 6 */
}                                       /* 7 */
```

Line 1 (event line) defines that if the pushbutton “PB_Show” is selected, the following program block in lines 2 to 7 is to be executed.

The braces in lines 2 and 7 limit the program block connected to the event line. Line 3 will trigger a conditional program execution, leading to processing the commands in line 5 if the expression in brackets is *true*. If this expression is *false*, line 5 will **not** be executed.

Line 5 assigns the value *true* to the object attribute *.visible* of the main window, so this window becomes visible.

10.1 General Structure of a Statement

Each instruction in the Rule Language will be concluded by a semicolon. This character marks the end of a statement. Exceptions here are the keyword **end** and the derived words such as **endif**, **endfor**, **endcase**, **endwhile**, which do not need a semicolon.

Syntax

```
<statement> ;
```

If this semicolon is missing, the system usually outputs an error for the following line.

10.2 Commenting Instructions

Comments can be added to the rule structure almost at any position. Comments are introduced by

```
!!
```

marking that the entire following line has to be considered as comment. Comments may be made before the event line and before every statement in the rule interior. They must not occur before the declaration of local variables and within expressions.

Example

```
!! Something has been selected in the list. This is why
!! the pushbuttons Delete and Information are released
on LListe select
{
  !! Switching the Delete pushbutton
  this.window.PLoeschen.sensitive := true;

  !! Switching the Information pushbutton
  this.window.PInfos.sensitive := true;
}
```

Of course, comments can be placed at attribute definitions. Predefined attributes like *.visible* or *.text* may be commented as well as indexed attributes. The same applies to user-defined attributes.

Beispiel

```
!! Example
dialog Comments

!! Communication model for data
model record MRComm
{
  !! Unique identifier for the respective record
  integer ID;
  !! Database identifier
  string S[3];
  !! Host
  .S[1] := "aldadin";
  !! Database
  .S[2] := "inforaclemix";
  !! Query language
```

```
  .S[3] := "SQL";
}

model listbox MList
{
  !! First entry
  .content[1] "Yes";
  !! Second entry
  !! with two lines
  .content[2] "It works";
  !! "normals" work too
  .width 100;
}
!! End of dialog
```

Please note the following with regard to comments in text files when using the IDM Editor:

With !! comments placed before attributes, the ISA Dialog Manager does not distinguish between attribute declaration and assignment of a default value, as in output both will be put on the same line anyway.

Since IDM version A.05.01.e !! comments for the same attribute are merged in order to e.g. preserve all comments for multiple assignments as well as for separated declarations and assignments of default values together with the last value.

Comments for shadow attributes should be used with caution. These comments can only be maintained for the attribute declaration as the actual value is usually stored at another object.

10.3 Comments

The ASCII dialog file may contain comments. There are two types of comments.

1. Comments which are preserved when the file is read or written by the IDM editor, indicated by "!!"
   The comments depend on objects. They can be defined in special positions:
   - » immediately in front of the object to which the comment is assigned,
   - » in rules between statements.

   When the ASCII file is written, such comments are automatically indented, depending on their position, and not depending on the indentation that may have existed when the file was read.
2. Comments which are not preserved when the file is read or written by the IDM editor, indicated by
   /* ... */
   // ...
   Such comments can be inserted anywhere in the ASCII file (between statements). Their position does not influence the IDM parser.

10.4 Operators in the Rule Language

10.4.1 Assignment Operators

With the assignment operator the left side of the operator is assigned the value of the right side. An assignment operator is represented in the Rule Language through a colon followed by an equals sign.

```
:=
```

or through two colons followed by an equals sign.

```
::=
```

The ":=" assignment always triggers *changed* events. With the "::=" assignment operator **no** *changed* events are triggered.

With assignments it is important that the data types on both sides correspond, i.e. a numerical value can only be assigned to a numerical value but not to a string, for example.

Example

```
dialog D
record R
{
  integer I;
  on .I changed
  {
    print "Event!";
  }
}

on dialog start
{
  !! triggers event
  R.I := 1;
  !! triggers no event
  R.I ::= 2;
}
```

10.4.2 Comparison Operators

In the Rule Language values can be compared with comparison operators (as in other programming languages).

To do so, there are the following comparison operators in the Rule Language:

Operator Characters	Description
<	less
<=	less or equal
=	equal
>=	greater or equal
>	greater
<>	unequal

With the help of these expressions, *integer* values can be compared. For the operators “=” and “<>” all data types are allowed.

10.4.3 Arithmetical Operators

Like in other programming languages, expressions or values in the Rule Language can be connected with the help of arithmetical operators.

The following arithmetical operators exist in the Rule Language:

Operator Characters	Description
+	Addition of two values.
-	Subtraction of two values.
/	Division of two values.
*	Multiplication of two values.
%	Modulo formation of two values (remainder with division).

All these operations can be used for *integer* values.

Also, two strings can be concatenated with the “+” operator.

10.4.4 Logical Operators

In the Rule Language expressions can be combined with the help of logical operators.

For this there are the following logical operators:

Operator	Description
`and`	By means of this operator two expressions are combined by a logical AND. All partial expressions are evaluated, even if the result is already known, and only finally the result is calculated.
`andthen`	Like `and`, but the expression is evaluated only until the remaining part of the expression does not affect the result. Evaluation is stopped at this point and the result is returned immediately.
`or`	By means of this operator two expressions are combined by a logical OR. All partial expressions are evaluated, even if the result is already known, and only finally the result is calculated.
`orelse`	Like `or`, but the expression is evaluated only until the remaining part of the expression does not affect the result. Evaluation is stopped at this point and the result is returned immediately.
`not`	By means of this operator an expression is negated.

For instance, the operator `andthen` comes in handy to check if an object has been created before some action is performed with it:

```
variable object O := null; !! not instantiated yet

if O <> null andthen O:DoSomething() then
```

This does not raise an error message, because `andthen` stops the evaluation after *O <> null* has been evaluated to *false*.

10.5 Brackets in the Rule Language

In the Rule Language there are three kinds of brackets with different meanings.

Bracket Type	Description
`{}`	With the help of braces the rule bodies can be combined. Within these braces the actual rule, i.e. the part which is to execute something, is written.
`[]`	With the help of square brackets a value will be indexed. For example, the fifth element of a field can be accessed. If two indices are needed for the access, both indices will be written within the brackets separated by commas.

Bracket Type	Description
( )	With the help of these round brackets function and rule call can be executed. Within these brackets the current parameters of the rule or function are indicated. As a further function, these brackets can be used to change the processing order within a statement. The expressions in brackets will be connected first. The result hereof will then be connected with the expressions outside the brackets.

10.6 Changing Attribute Values

Each object attribute with write access can be arbitrarily changed.

Attributes can be changed with the help of a combination of

- attributes
  Please refer to manual “Attribute Reference” for further details.
- functions
  Please see chapter “Functions of the DM Interface” in manual “C Interface - Functions” for further details.
- variables
  Please see chapter “Global Variables” for further details.

Examples

- Setting the visibility of a window:
```
MyWindow.visible := true;
```
- Setting two lines in a listbox:
```
MyListbox.content[1] := "first string";
MyListbox.content[2] := "second string";
```
- Combination of two edittexts in a third edittext with a blank in between:
```
Edittext3.content := Edittext1.content + " " + Edittext2.content;
```

10.7 Control of the Program Flow

In a rule, there are different constructs available for the control of the program flow.

- if-then-else
- if-elseif-else
- case statement (“switch”)

10.7.1 if-then-else

An **if** statement can be used to define a program branch with statements that is only processed if a condition is satisfied. Optionally, another program branch with statements can be defined, which is executed if the condition is not met.

Syntax

```
if <boolean expression>
then
  <statements>
{ else
    <statements> }
endif
```

If <boolean expression> is *true*, the <statements> between **then** and **else** will be executed.

If <boolean expression> is *false*, the <statements> between **else** and **endif** will be executed.

The **else** part may be left out in this statement.

Example

```
on PbBlink select
{
  if (WnMain.visible)
  then
    WnMain.visible := false;
  else
    WnMain.visible := true;
  endif
}
```

When confirming the “PbBlink” button it is first checked whether the window “WnMain” is visible (value of *WnMain.visible*).

If the window is visible (*WnMain.visible* = *true*), it is set invisible with the statement *WnMain.visible* := *false*.

If the window is invisible (*WnMain.visible* = *false*), it is set visible with the statement *WnMain.visible* := *true*.

10.7.2 if-elseif-else

The “if-elseif-else” construct is a shortened notation for nested “if-then-else-if”.

Syntax

```
if <boolean expression>
then
```

```
  <statements>
[ elseif <boolean expression>
  then
    <statements> ]
{ else
    <statements> }
endif
```

elseif parts can be repeated in this statement as often as you want.

Example

```
on PbBlink select
{
  if (WnMain.childcount = 0)
  then
    print "Have no objects.";
  elseif (WnMain.childcount = 1)
  then
    print "Have one object.";
  else
    print "Have several objects.";
  endif
}
```

10.7.3 case Statement

A **case** statement can be used instead of several “if-then-endif” constructs. It is useful when a program is to branch itself at a certain position.

Syntax

```
case <expression>
[ in <criterion> [ , <criterion> ] :
    <statements> ]
{ otherwise :
    <statements> }
endcase

criterion ::= <value> | <min> .. <max>
```

Example

```
case Edittext.content
  in "Hello":
    print "Yes";
  in "Bye":
```

```
    exit();
  otherwise:
endcase
```

A case statement checks for each **in** expression, if it is true.

Example

```
variable integer V := 10;
case V
  in 10:
    print "10";
  in 1..100:
    print "1..100";
  in 10..19:
    print "10..19";
  in 20..29:
    print "20..29";
  otherwise:
    print "nothing";
endcase
```

Output

```
"10"
"1..100"
"10..19"
```

All statements which fulfill the value are enlisted here.

If none of the **in** expressions is fulfilled, then – and only then – **otherwise** will be carried out, if it is given.

As soon as one or more **in** expressions are fulfilled, **otherwise** is not carried out any more.

In contrast to C and other programming languages, variable values may occur in **in** expressions, too.

Example

```
on PushbuttonOK select
{
  case true
    in Checkbox_1.active:
      // action for first checkbox if it is active
    in Checkbox_2.active:
      // action for second checkbox if it is active
    in Checkbox_3.active:
      // action for third checkbox if it is active
    // otherwise is not necessary here
```

```
  endcase
}
```

Since in **in** expressions calculations are also allowed, the above example can be expanded for the case that a certain action can be carried out for inactive checkboxes (`not`).

```
on PushbuttonOK select
{
  case true
    in Checkbox_1.active:
      // action for first checkbox if it is active
    in Checkbox_2.active:
      // action for second checkbox if it is active
    in Checkbox_3.active:
      // action for third checkbox if it is active
    in not Checkbox_1.active:
      // action for first checkbox if it is not active
    in not Checkbox_2.active:
      // action for second checkbox if it is not active
    in not Checkbox_3.active:
      // action for third checkbox if it is not active
    // otherwise still not necessary
  endcase
}
```

Remark

Any variables are allowed. Please take care that you are using the correct type.

10.8 Loop Constructs

10.8.1 for Loop

This construct is used for the formulation of loops, if the number of runs is defined before the entry in the loop.

Syntax

```
for <counter> := <start> to <end> { step <increment> } do
  <statements>
endfor
```

At the loop start, the value <start> is assigned to <counter>. After each loop run, <increment> is added to <counter>. If <increment> was not given, the default increment *1* is added. The loop runs until <counter> becomes larger than <end>. At each loop run, the <statements> are performed which can access the current value of <counter>. <start>, <end> and <increment> have to be *integer*

values. <counter> has to be able to accept *integer* values. <start>, <end> and <increment> are calculated only once before the first loop run.

Example

```
for Index := 1 to Window.childcount do
  print Window.child[Index];
endfor
```

It is not possible to cancel a `for` loop before it is terminated.

If the cancel criteria cannot be reached at `for` loops, the loop will not be carried out.

Example

```
for I := 1 to 10 step -1 do
```

10.8.2 foreach Loop

This construct is used to create loops that iterate over all elements of a collection.

Syntax

```
foreach <item> in <expression> do
  <statements>
endfor
```

The loop terminates with an error if the *<expression>* is not a collection or the loop value cannot be assigned to the *<item>*. The *<expression>* is evaluated only once initially. The control variable *<item>* may be changed within the *<statements>*. A change of the control variable does not affect the number of cycles or the value of the control variable in the next cycle.

Example

```
dialog D
window Wi
{
  listbox Lb
  {
    .xauto 0;  .yauto 0;
  }
  on close { exit(); }
}

on dialog start
{
  variable list StationList:= ["ABC", "CBS", "NBC", "ESPN"];
  variable string Station;
```

```
  foreach Station in StationList do
    Lb.content[Lb.itemcount+1] := Station;
  endfor
}
```

10.8.3 while Loop

This construct is a repetition statement, a loop statement, at which the number of repetitions depends on a condition. The condition is checked at the beginning of a new run.

Syntax

```
while <condition> do
  <statements>
endwhile
```

The value <condition> is calculated before each run of a loop. It has to be a *boolean* value. If it is *true*, the <statements> will be carried out and the loops will be started again from the beginning. If the value is *false*, the loop will be canceled. There is no other way to end a `while` loop.

Example

```
while (Index < Window.childcount) do
  print Window.child;
  Index := Index + 1;
endwhile
```

10.9 Calling Named Rules

Named rules are called by giving the name of the rule and the current setting of the parameters.

Example

A rule is defined as follows:

```
rule integer Calculate ( integer Val1 input, integer Val2 input)
```

The call of this rule can be as follows:

```
Calculate ( 14, 3);
```

10.10 Local Variables

Local variables can be used to store any value within a rule. They are only known in the rule in which they were defined. This is where they differ from global variables, which are known in the entire dialog or module (see chapter “Global Variables”).

Within the rule where they are defined, local variables overlay any existing global variables with the same identifier.

Local variables can be declared as “normal” variables or as **static** variables:

- Normal variables lose their value after the rule has ended. They are reinitialized each time the rule is executed.
- Static variables retain their value even after the rule has ended. They are only initialized when the rule is executed for the first time.

10.10.1 Normal Local Variables

Syntax

```
variable <data type> <variable name> { := <expression> }
                [ , <variable name> { := <expression> } ] ;
```

Several variables of the same data type can be declared in one statement. The identifiers of the variables are separated by , (comma).

Local variables can be initialized with a value or an expression directly in the definition. If an expression is used for initialization, the following should be noted:

- The evaluation of the expression must yield a value with the data type of the variable.
- Variables used in the expression must be declared before, i.e. already known.

Example

```
rule integer R1(integer X)
{
  variable integer Z := X * 3;

  if (X > 100) then
    Z := Z / 2;
  endif

  return Z;
}
```

A definition without direct initialization is also possible. In this case, however, a value must have been assigned before the first read access to the variable.

Example

```
rule void R2 ()
{
  variable integer MyNumber;
```

```
  MyNumber := 5;

  print MyNumber;
}
```

Once local variables contain a value, they can be assigned to attributes of objects. It only needs to be ensured that the data types of the local variable and the attribute allow this (identical data types or object attribute of type *anyvalue*).

Example

```
rule void R3 ()
{
  variable string MyWord;

  MyWord := "Hello world";
  Edittext.content := MyWord;
}
```

A variable of the type *object* can be used like an object, i.e. every attribute of the object stored in the variable can be changed.

Example

```
rule void R4 ()
{
  variable object MyObject;

  MyObject := MyWindow;
  MyObject.title := "New Title";
  MyObject.visible := true;
}
```

10.10.2 Static Variables

Static variables are declared by the additional keyword **static** before the keyword **variable**. Unlike normal local variables, they can only be initialized with values or other variables, but not with expressions.

Syntax

```
static variable <data type> <variable name> { := <value> | <variable> }
                          [ , <variable name> { := <value> | <variable> } ] ;
```

Static variables may have the same data types as global or local variables.

The **initializations** in the declarations of static variables are only carried out **in the first rule call**, but not in further calls. This also applies if the value of the variable used for initialization has changed in the meantime. Instead, their values from the last rule execution remain unchanged.

To initialize static variables, only variables that were **previously** defined in the rule code or as global variables in the dialog or module may be used.

Examples

```
variable integer G:=10;
rule Example (object O input)
{
  variable boolean B;
  static variable integer X:=1, Y:=G, Z:=Y;
  static variable object  Q:=O;
  static variable boolean J:=B;
  variable integer I:=X;
}
```

```
rule Example2
{
  static variable anyvalue A;
  if A = 1 then
    print "something";
  else
    A := 1;
    print "further initializations";
  endif
}
```

10.11 Return of Values

With the `return` instruction, every rule can be left at the current position and – if necessary – can return a value to the calling rule.

Syntax

```
return { <expression> } ;
```

<expression> is required if the rule has a return type other than *void*. The data type of <expression> must match the return type of the rule.

Example

```
rule integer Addivide (integer Arg1 input, integer Arg2 input,
                       integer Arg3 input)
{
```

```
  variable integer Sum;
  Sum := Arg1 + Arg2;
  if (Arg3 <> 0) then
    return (Sum / Arg3);
  endif
  return (0);
}
```

10.12 Call of Application Functions

Any functions which have been specified in a dialog file can be called directly out of the rules.

Exception

Canvas functions, object callback functions, fomat functions and reloading functions cannot be called out of the rules!

The function parameters have to be substituted by the current values. Their types have to correspond to the specification of the function.

Examples

Declaration:

```
function c integer Add (integer, integer);
```

Call:

```
Edittext.content := itoa (Add (atoi (Edittext2.content),
                    atoi (Edittext3.content)));
```

Meaning:

input:	two numbers
output:	sum is returned

Declaration:

```
function c void GetAddress (string, string output, string output);
```

Call:

```
GetAddress (Edittext1.content, Edittext2.content, Edittext3.content);
```

Meaning:

input:	e.g. name
output:	e.g. address (street and city)

11 Built-in Functions

The IDM has standard functions to determine the length of a string and to convert number strings and character strings. Thus, you do not have to define these functions yourself.

These built-in functions can be included into the action part of a rule.

11.1 append()

With this function a new value can be appended to a collection (data types *hash*, *list*, *matrix*, *refvec* and *vector*) or a string once or several times.

In general, the data type of the (optional) new value has to be a scalar and match the value type of the collection.

The number of repetitions for the appending (*Count* parameter) has to be >= *0*.

In case of an error, the function call is aborted with a `fail`.

Definition

```
anyalue append
(
      anyvalue ListValue  input,
      anyvalue NewValue   input
  { , integer  Count := 1 input }
  { , enum     Dir   := dir_row input }
)
```

Parameters

anyvalue *ListValue* input
This parameter specifies the collection or string where a value shall be appended.

anyvalue *NewValue* input
This parameter defines the new value to be added. The value must match the value type of the collection.

integer ***Count*** *:= 1 input*
This optional parameter defines the number of times a new value (respectively how many rows or columns) shall be appended. Values >= *0* are allowed, with no appending being performed for *0*. By default, the new value is appended once.

enum ***Dir*** *:= dir_row input*
For collections of data type *matrix*, this optional parameter controls whether new rows (*dir_row*) or new columns (*dir_column*) are added.

Return value

The function returns the modified collection respectively the modified string.

Particularities

» *hash*
Since this collection data type does not have a defined index range, appending takes place after the last *integer* index. Thus a *hash* with an indexing of *1,2,3…* is assumed. The last set index of data type *integer* is determined and then it is appended after it.

» *matrix*
Appending to a matrix always happens by rows or columns. This is controlled by the *Dir* parameter. By default (*dir_row*) new rows are added. If *dir_column* is given, new columns are added.

» *refvec*
Appending *null* objects is not allowed. If the object ID already exists, it is repositioned to the last position.

» *string*
The new value is appended to the end of the string.

» *Default Values*
To determine the index position for the new values, default values are also taken into account except for a *hash* collection.

Examples

» Multiple appending to a string

```
print append("hi", " ho", 3);
```

Output

```
"hi ho ho ho"
```

» Multiple appending to a *list* collection

```
variable list L := [4711, "cologne"];
print append(L, true, 2);
```

Output

```
[4711,"cologne",true,true]
```

» Appending two rows to a matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a", [1,2]=>"b",
                       [2,2]=>"c", [3,1]=>"end" ];
print append(M, "new", 2);
```

Output

```
[[1,1]=>"a",[1,2]=>"b",[2,1]=>"?",[2,2]=>"c",[3,1]=>"end",[3,2]=>"?",
[4,1]=>"new",[4,2]=>"new",[5,1]=>"new",[5,2]=>"new"]
```

» Appending to a hash with mixed indexing

```
variable hash H := [false=>.xleft, 31=>.width, 33=>.ytop, "x"=>1];
print append(H, .height);
```

Output

```
[false=>.xleft,31=>.width,33=>.ytop,34=>.height,"x"=>1]
```

» Appending to a *vector* list

```
variable vector[integer] V := [9, 8, 7];
print append(V, 6);
```

Output

```
[9,8,7,6]
```

Availability

Since IDM version A.06.02.g

See also

Built-in function **insert()**

11.2 applyformat()

With this function, a string can be converted by applying a format to it.

The return value is the same string that is displayed by an edittext when it uses the same format and has the given string as its contents.

Definition

```
string applyformat
(
  string FormatString input | object FormatResource input ,
  string String input
)
```

A further definition of the functions is as follows:

```
string applyformat
(
  object FormatFunc   input,
  string FormatString input,
  string String input
)
```

Parameters

string *FormatString* input
In this parameter the format string with the help of which the string is to be formatted is indicated.

object *FormatResource* input
In this parameter, you declare the format resource with the help of which the string is to be formatted.

object *FormatFunc* input
In this parameter the format function which is to format the string is indicated.

string *String* input
In this parameter the string to be formatted is indicated.

Return value

The formatted string is returned as a result.

Example

```
applyfomat("NN-NN-NN", "121295");
```

Output

```
12-12-95
```

11.3 atoi()

This function converts an optionally signed number string to a decimal number (**a**scii **to** **i**nteger). The following is valid:

- One sign + | -
- Digits *0…9*
- Space as fill-in at each position

Let us e.g. assume that *5* is to be added to number *123* in an editable field. For this purpose, the strings "123" and "5" are interpreted as numbers *123* and *5*. The result is *128*.

Definition

```
integer atoi
(
  string IntString input
)
```

Parameters

string *IntString* input

In this parameter the string to be changed into a number is indicated.

Return value

The value in which the string was converted.

Remark

This function call ends with an error which can be caught by the function **fail()**, when the indicated string does not contain any number or other characters.

Example

```
Sum := atoi(Edittext1.content) + atoi(Edittext2.content);
```

11.4 beep()

With this function you can create a tone. According to the used window system, you can specify by parameters in which form and how long the tone is to be created.

Definition

```
void beep
(
  { enum Mode input }
)

void beep
(
  integer Volume    input,
  integer Duration  input,
  integer Frequency input
)
```

Parameters

enum ***Mode*** *input*

With this parameter, you can select a tone by using the following symbols:

beep_error
: Window system specific sound when an error has occurred.

beep_note
: Window system specific sound for an information.

beep_ok
: Window system specific sound for a confirmation.

beep_question
: Window system specific sound for a prompt to the user.

beep_warning
: Window system specific sound for a warning.

The molding of the tones or if the tones differentiate at all depends very much on the used window system.

integer *Volume* input

This parameter specifies the **volume** in which the tone is to be created. The definition is based on the usual X Windows system.
The value range is between -100 and 100. Value 0 defines the default volume. For negative values the volume varies in percentage between “inaudible” (-100) and the default volume. For positive values the volume varies in percentage between the default volume and the maximal possible volume (= 100).
Note: Although this parameter must be specified under Microsoft Windows, it is ignored. The volume set in the system is used.

integer *Duration* input

With this parameter, you can define the tone's **duration** in milliseconds (range between 0 and 60.000). Value 0 defines the default duration.
Note: Under Microsoft Windows, a default tone is generated with the value *0*.

integer *Frequency* input

The tone's **frequency** (range between 0 and 20,000 Hz) can be defined with this parameter. Value 0 defines the default frequency.
Note:Under Microsoft Windows, the value range is 37 - 32767 Hz. For other values, a standard tone is generated.

Warning

If and how these parameter are considered when creating a tone depends very much on the used window system and possibly of the used hardware, too. The behavior when creating different tones very quickly one after each other is also very much system-dependent.

This function is not meant to create tunes of tones but to draw the end-user's attention to specific situations with the help of single tones.

When this function is called, the ISA Dialog Manager tries to create the tone according to the parameters as far as is possible on the used system.

Remarks

If the function is created without parameter, it creates a single tone.

The parameters *Volume*, *Duration* and *Frequency* have to be declared in the given sequence. Parameters which were not used can be omitted at the end.

Examples

```
beep();
beep(beep_warning);
beep(90, 10000);
beep(0, 0, 1200);
```

11.5 closequery()

This function closes a dialogbox (a window with the attribute *.dialogbox* set to *true*) by setting the *.visible* attribute to *false*. Furthermore, this function sets a return value for the dialogbox.

Definition

```
void closequery
(
  anyvalue RetVal input
)
```

Parameters

anyvalue *RetVal* input

This parameter specifies the return value that the function **querybox()** shall return.

Notes

The function **closequery()** should only be used, when the dialogbox has been opened with the function **querybox()**.

The parameter *ShipEvent* of the **querybox()** function controls, whether a *changed* event for *.visible* is triggered for the dialogbox.

Example

```
dialog D

window Box
{
  .dialogbox true;
  .visible   false;
  .title     "Enter password";

  edittext E
  {
    .format "S";
  }

  pushbutton P
  {
    .text "OK";

    on select
    {
      closequery(E.content);
    }
  }
```

```
}

window WnLogin
{
  child pushbutton PbLogin
  {
    .text "Login";

    on select
    {
      if (querybox(Box) <> "password") then
        exit();
      endif
    }
  }
}
```

11.6 concat()

The **concat()** function concatenates several strings into one string. During concatenation, a separator string can be inserted between the strings or the strings can be appended to each other repeatedly.

Definition

If the first parameter is a string, this string is always inserted between two strings to be concatenated.

```
string concat
(
      string   Separator input,
      anyvalue Value1    input
  { , anyvalue Value2    input
  ...
  { , anyvalue Value15   input } }
)
```

If the first parameter is an *integer* value, it defines how often the concatenation of the strings is repeated.

```
string concat
(
      integer  Repeat  input,
      anyvalue Value1  input
  { , anyvalue Value2  input
  ...
  { , anyvalue Value15 input } }
)
```

Parameters

string *Separator* input

This parameter contains a delimiter string that is inserted between two strings during concatenation.
Separator cannot be used together with *Repeat*.

integer *Repeat* input

This parameter defines how often the concatenation of the strings shall be repeated. First, a string is created from the passed *Value* parameters, then this string is appended to each other as often as specified in *Repeat*. If *Repeat* <= *0*, an empty string is returned.
Repeat cannot be used together with *Separator*.

anyvalue *Value1* input

anyvalue ***Value2*** *input*

…

anyvalue ***Value15*** *input*

In these (optional) parameters, the values that shall be concatenated to a string are passed.

Scalar values are converted to strings for concatenation. If the parameters contain collections, then first the values contained therein are concatenated (without default values) and then the parameters are concatenated to each other. The approach is the same as calling *sprintf("%s", <Value>)*.

Return value

A string in which the passed values have been concatenated.

Examples

» Concatenation of three values with a delimiter

```
text TxTres "Tres";
...
print concat(",", 1, "two", TxTres);
```

Output

```
"1,two,Tres"
```

» Repeated concatenation

```
print concat(3, "|", [".", ".."]);
```

Output

```
"|...|...|..."
```

11.7 countof()

This function returns the size of a collection. This is usually the highest index value.

Typically, **countof()** and **itemcount()** return the same result for values of the data types *refvec*, *vector*, and *list*. However, **itemcount()** offers the more generic use, whereas **countof()** is better suited for structured, type-adapted use.

Definition

```
anyvalue countof
(
  anyvalue Value input
)
```

Parameters

anyvalue *Value* input

This parameter specifies the value for which the indexing type or the highest index value shall be determined.

Return value

nothing
: The passed value is scalar.

1 … 2^{31}
: Highest index of the passed list (data types *list*, *vector*, *refvec*).

[0 … 65535,0 … 65535]
: Highest index of the passed matrix.

anyvalue
: Data type of the index of the passed associative array (data type *hash*).

Example

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france"
    /* [2,2] => inherited from default [0,0] */ ];
  variable integer Row, Col;
  variable anyvalue Count, Idx;

  /* print the Matrix values [0,0] [0,1] ... [2,2] */
  Count := countof(Matrix);
```

```
  for Row:=0 to first(Count) do
    for Col:=0 to second(Count) do
      Idx := [Row,Col];
      print sprintf("%s : %s", Idx, Matrix[Idx]);
    endfor
  endfor
  exit();
}
```

See also

Built-in functions itemcount(), valueat()

Method index

Attribute count

C function DM_ValueCount

11.8 create()

New objects can be created in the IDM during runtime with the function **create()**. The function returns the new generated object as result.

Definition

```
object create
(
      object  Class  input,
      object  Parent input
  { , object  Dialog input }
  { , integer Type   := 3 input }
  { , boolean CreateInvisible := false input }
)
```

Parameters

object *Class* input

This parameter indicates the object from which a new instance is to be created. Thus, the object may be a default or a model.

object *Parent* input

This parameter indicates the object parent.

*object **Dialog** input*

The optional parameter indicates the dialog to which the object is to belong.

*integer **Type** := 3 input*

This parameter (optional) indicates whether a default, a model or an object is to be created:

1
: Default

2
: Model

3
: object

If this parameter is not declared, an object will be generated automatically.

*boolean **CreateInvisible** := false input*

With this optional parameter you can define whether the object is to be created invisibly or as it was defined in the model.

true
: The object is always created invisibly.

false
: The visibility is taken from the model or default.

Return value

objectId
: Identifier of the newly created object.

null
: Null-ID - since the object could not be created.

Example

```
dialog CREATE

variable object OBJ_Create := null;
variable integer Y := 50;

default window
{
}

default pushbutton
{
}

window W1
{
  .title "creating objects";
  .visible true;

  child pushbutton DO_create;
  {
    .xleft 10;
    .ytop 10;
    .text "create";
  }
}

on DO_create select
{
  !! Creating object and specifying it as child of W1
  OBJ_Create := create (pushbutton, W1);
  !! Checking if the object has been created
  if OBJ_Create <> null then
    !! Setting coordinates
    OBJ_Create.ytop := Y;
    Y := Y + 30;
  endif
}
```

See also

Method :create()

C function DM_CreateObject in manual “C Interface - Functions”

COBOL function DMcob_CreateObject in manual “COBOL Interface”

11.9 delete()

This function can be used to delete one or more values from a collection (data types *hash*, *list*, *matrix*, *refvec* and *vector*) at a given position. The subsequent values are shifted forward. The data type *string* is also supported to remove characters from a string.

The values are deleted at the specified position. The position must be >= *1*, i.e. it is not possible to delete default values at the indices *[0]* or *[0,0]* with this function. The largest possible position is the current *countof(ListValue)*.

The number of values to be deleted is determined by the *Count* parameter and should be >= *0*.

In case of an error, the function call is aborted with a `fail`.

Also considered an error is the attempt to delete more values respectively rows or columns than actually exist.

Definition

```
anyalue delete
(
      anyvalue ListValue input
      integer  Pos   input
  { , integer  Count := 1 input }
  { , enum     Dir   := dir_row input }
)
```

Parameters

anyvalue *ListValue* input

This parameter specifies the collection or string where values shall be deleted.

integer *Pos* input

In this parameter, the index position is defined at which inclusively deleting begins. Values >= *1* are permitted. For *matrix* collections, depending on the *Dir* parameter, *Pos* represents either a row position (*dir_row*) or a column position (*dir_column*).

*integer **Count** := 1 input*

This optional parameter indicates the number of values (respectively rows or columns for a matrix) to be deleted. Values >= *0* are allowed, with no deleting being performed for *0*. The number should not exceed the possible position range when deleting.

*enum **Dir** := dir_row input*

This optional parameter defines the orientation of the index position, which is important for a matrix. The default value *dir_row* means that rows are deleted in the matrix. With *dir_column* columns are deleted.

Return value

The function returns the modified collection respectively the modified string.

Particularities

» *hash*
Since this collection data type does not have a defined index range, deleting takes place based on the *integer* indices. Thus a *hash* with an indexing of *1,2,3…* is assumed. There are no limitations concerning the largest possible index position.

» *matrix*
Deleting in a matrix always happens by rows or columns. This is controlled by the *Dir* parameter. By default (*dir_row*) rows are deleted. If *dir_column* is specified, columns are deleted. This also means that the position refers to a row in the first case and to a column in the second. The matrix is reduced accordingly.

» *string*
Here the position refers to the character position in the string, i.e. the string is treated as an array of characters.

Examples

» Multiple deletion of values from a list

```
variable list L := [17, "x", window, .xleft, null];
print delete(L, 2, 3);
```

Output

```
[17,null]
```

» Deleting a column from a matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a1", [1,2]=>"b1",
                       [2,2]=>"b2", [2,3]=>"c2" ];
M := delete(M, 2, dir_column);
print M;
print countof(M);
```

Output

```
[[1,1]=>"a1",[1,2]=>"?",[2,1]=>"?",[2,2]=>"c2"]
[2,2]
```

» Removing two characters from a string

```
print delete("hello", 3, 2);
```

Output

```
"heo"
```

» Deleting items from a hash with *integer* indexing

```
variable hash H := [2=>.xleft, 31=>.width, 33=>.ytop];
H := delete(H, 30, 3);
print H;
```

Output

```
[2=>.xleft,30=>.ytop]
```

» Deleting from a *vector* list

```
variable vector[boolean] V := [true, false, false, false, true];
print delete(V, 2, 3);
```

Output

```
[true,true]
```

» Deleting the first two items from a *refvec* list

```
variable refvec R := [PbApply, PbCancel, PbRevert];
print delete(R, 1, 2);
```

Output

```
[pushbutton D.PbRevert]
```

Availability

Since IDM version A.06.02.g

See also

Built-in function **insert()**

11.10 destroy()

By means of this function any object or model in a dialog can be deleted. All children of this object are also deleted.

Definition

```
boolean destroy
(
      object  Object input
  { , boolean DoIt   := true input }
)
```

Parameters

object *Object* input
In this parameter the object to be deleted is specified.

*boolean **DoIt** := true input*
This optional parameter controls how the relevant object is to be deleted. If this parameter is specified as *true*, the object is deleted and all rule parts using this object are changed so that the corresponding commands are removed. If the object to be deleted is a model and if the second parameter is *false*, the model will only be deleted if it is not used by any other object. If *true* is given here, the model is deleted and all objects using this model refer to the next superordinate model or default.

Return value

true
The object could be deleted.
false
The object could not be deleted.

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

Example

```
dialog Destroy

default window
{
}

default pushbutton
{
}

window W1
```

```
{
  child pushbutton P1
  {
  }
}

on dialog start
{
  destroy (P1, true);
}
```

See also

Method :destroy()

C function DM_Destroy in manual “C Interface - Functions”

COBOL function DMcob_Destroy in manual “COBOL Interface”

11.11 dumpstate()

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

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

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

The dumpstate information is always encased between *"*** DUMP STATE BEGIN ***"* and *"*** DUMP STATE END ***"* and can have the following segments:

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

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

Definition

Since IDM version A.05.02.g:

```
void dumpstate
(
  {   anyvalue Filename input
  { , enum     State    := dump_error input } }
)

void dumpstate
(
  { { anyvalue Filename input, }
      enum     State    := dump_error input }
)
```

Up to IDM version A.05.02.f4:

```
void dumpstate
(
  enum State input
)
```

Parameters

anyvalue ***Filename*** *input*

File to which the status information is written (optional parameter, available since IDM version A.05.02.g).
Default value: When the parameter is missing, the status information is output to the trace or log file.
Up to and including IDM version A.05.02.f4 the output is always written into the trace or log file.

enum ***State*** *:= dump_error input*

This parameter influences the sections of the status information that are output (optional parameter).
Since IDM version A.05.02.g this parameter is optional. When it is missing, the default value *dump_error* applies.

Value range

dump_all
All sections are written out in an abbreviated form.
This corresponds to the output in case of a FATAL ERROR.

dump_error
The sections ERRORS, CALLSTACK and EVENTS are written out in an abbreviated form.
This is the normal output in the case of EVAL ERRORS.

dump_events
The sections THISEVENTS and EVENT QUEUE are written out in full.

dump_full
All sections are written out in full.

dump_locked
The section SLOTS is written out in full. In addition, for locked objects their attribute values are written out.

dump_memory
The section MEMORY is written out in full.

dump_none
No action (nothing is written out).

dump_process
The section PROCESS is written out in full.

dump_short
All sections (excluding SLOTS) are written out in an abbreviated form.

dump_slots
The section SLOTS is written out in full.

dump_stack
The section CALLSTACK is written out in full.
dump_usage
The section USAGE is written out in full.
dump_uservisible
The section VISIBLE OBJECTS is written out in full for all visible top-level objects including their children, the pre-defined and user-defined attributes.
dump_visible
The section VISIBLE OBJECTS is completely written out.

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

Example

```
dialog D

window Wi
{
  .title "dumpstate()-example";

  edittext Et
  {
    .content "66";
    .xauto 0;

    on deselect_enter
    {
      variable string Content := this.content;

      if fail(Pg.curvalue := atoi(Content)) then
        print "Conversion errror!";
        dumpstate(dump_error);
      endif
    }
  }

  pushbutton Pb
  {
    .ytop 33;
    .text "dump objects";

    on select
    {
      dumpstate(dump_visible);
    }
  }
```

```
  progressbar Pg
  {
    .yauto -1;
    .xauto 0;
  }

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

Availability

IDM versions A.05.01.g3, A.05.01.h, since A.05.02.e

See also

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

11.12 exchange()

With this function two values of a collection (data types *hash*, *list*, *matrix*, *refvec* and *vector*) can be swapped. The data type *string* is also supported to swap individual characters within a string.

The two position values (*Pos1* and *Pos2*) specify the index positions whose values are exchanged. The positions must be >= *1* and must not exceed the possible number of items in the collection.

In case of an error, the function call is aborted with a `fail`.

Definition

```
anyvalue exchange
(
     anyvalue ListValue input,
     integer  Pos1 input,
     integer  Pos2 input
 { , enum     Dir  := dir_row input }
)
```

Parameters

anyvalue *ListValue* input
This parameter specifies the collection or string where values shall be exchanged.

integer *Pos1* input
This parameter specifies the first index position whose value is swapped with that of the second index position (*Pos2*). Values >= *1* are permitted. For *matrix* collections, depending on the *Dir* parameter, *Pos1* represents either a row position (*dir_row*) or a column position (*dir_column*).

integer *Pos2* input
This parameter specifies the second index position whose value is swapped with that of the first index position (*Pos1*). Values >= *1* are permitted. For *matrix* collections, depending on the *Dir* parameter, *Pos2* represents either a row position (*dir_row*) or a column position (*dir_column*).

*enum **Dir** := dir_row input*
This optional parameter defines the orientation of the index position, which is important for a matrix. The default value *dir_row* means that rows are swapped in the matrix. With *dir_column* columns are swapped.

Return value

The function returns the modified collection respectively the modified string.

Particularities

» *hash*
Since this collection data type does not have a defined index range, exchanging takes place based on the *integer* indices. Thus a *hash* with an indexing of *1,2,3…* is assumed.

» *matrix*
Swapping in a matrix always happens by rows or columns. This is controlled by the *Dir* parameter. By default (*dir_row*) entire rows are swapped. If *dir_column* is specified, entire columns are swapped. This also means that the position refers to a row in the first case and to a column in the second.

» *string*
Here the position refers to the character position in the string, i.e. the string is treated as an array of characters.

» *Default Values*
The collection data types *list*, *hash* and *matrix* allow default values. The exchange of such inherited values at valid positions is carried out consistently.

Examples

» Exchanging values in a list

```
variable list L := ["a", "c", "b", "d"];
print exchange(L, 2, 3);
```

Output

```
["a","b","c","d"]
```

» Swapping two rows of a matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a", [1,2]=>"b",
                       [2,2]=>"end", [3,1]=>"c" ];
M := exchange(M, 2, 3);
print M;
```

Output

```
[[1,1]=>"a",[1,2]=>"b",[2,1]=>"c",[2,2]=>"?",[3,1]=>"?",[3,2]=>"end"]
```

» Swapping two characters in a string

```
print exchange("X-Y", 1, 3);
```

Output

```
"Y-X"
```

» Exchanging two items in a hash with *integer* indexing

```
variable hash H := [2=>.xleft, 31=>.width, 33=>.ytop];
print exchange(H, 31, 32);
```

Output

```
[2=>.xleft,32=>.width,33=>.ytop]
```

» Exchanging values in a *vector* list

```
variable vector[boolean] V := [true, false];
print exchange(V, 1, 2);
```

Output

```
[false,true]
```

» Swapping the last two items of a *refvec* list

```
variable refvec R := [PbApply, PbCancel, PbRevert];
print exchange(R, 2, 3);
```

Output

```
[PbApply,PbRevert,PbCancel]
```

Availability

Since IDM version A.06.02.g

11.13 execute()

With this function an additional program can be started from the dialog script. Depending on the used operating system different types of programs to be started are supported.

Definition

```
boolean execute
(
      string  Command   input
  { , string  Arguments input }
  { , boolean Synchronous := false  input }
  { , enum ExeType    := exenormal  input }
  { , enum WindowType := showwindow input }
  {   object  Object input
    , integer Event  input
  { , anyvalue ReplyData input } }
)
```

Parameters

string *Command* input

In this parameter the name of the program to be executed and its path are given. The path can be defined in the usual way according to the operation system; relative paths are also valid. In addition the path can be defined in the usual IDM way by using environment variables.
If this function is used on Microsoft Windows you can specify a special Windows command instead. In this case Microsoft Windows will start the program of the relevant document and will execute the indicated command, provided that the program supports this command.
Microsoft Windows currently supports the following commands: “open”, “print” and “explore”. Please refer to the Microsoft Windows manual, function **ShellExecute()** for all commands currently valid. If you want to start the program depending on the document, you have to specify *execommand* for the parameter *ExeType*.

*string **Arguments** input*

This parameter is optional. In this parameter you can give arguments for the program to be started.

*boolean **Synchronous** := false input*

This optional parameter defines whether the program is to be started synchronously or asynchronously to the current program. If using the synchronous start the IDM will wait until the started program has been ended. During this time no processing is possible in the IDM.

Note

We do not recommend the synchronous option, as you cannot go on processing the current program. In this case **all** events will be queued until processing is continued.

enum ***ExeType*** *:= exenormal input*

This optional parameter specifies the type of the program to be executed. This parameter is evaluated on the operation system Microsoft Windows.

Value range

exenormal
The program is a “normal” program.

exeshell
The indicated program can be started within a shell only, e.g. a copy command. This is why a shell will be started before the actual program.

execommand (IDM FOR WINDOWS only)
The specified program is a command which is to be executed by the program relevant to the document. In this case it is not possible to query the return value of the program. In addition the *Synchronous* parameter will be ignored (see parameter *Command*).

enum ***WindowType*** *:= showwindow input*

This optional parameter specifies the window type of the program that has been started. This parameter is valid for the operation system Microsoft Windows.

Value range

hidewindow
The start window of the program to be started is to be hidden, as usually only one command is to be executed.

maxwindow
The start window of the program to be started is to be opened but not activated.

minwindow
The start window of the program to be started is to be opened as icon.

showinactive
The start window of the program to be started is to be opened as icon.

showwindow
The start window of the program to be started will be the active window.

Attention

The started application does not necessarily need to observe this parameter.
If the *ExeType* parameter has a value *exeshell*, the command interpreter is started maximized, not the application that the command interpreter starts.

object ***Object*** *input*

In this optional parameter you can specify an object to be sent to an external event once the program to be started has been ended. If this parameter is used the following parameter *Event* must also be specified.

integer ***Event*** *input*

This parameter must be specified only if the parameter *Object* is being used. This parameter defines the number of the external event which is to be sent to the relevant object after ending the program.

anyvalue ***ReplyData*** *input*

This optional parameter is allowed only if an object for an external event has been given. In this parameter any value is specified. This value will be passed on to the rule after having ended the program.

Return value

true

The program could be started.

false

The program could not be started.

Reaction to Program Exit

To react to the program exit the IDM offers the possibility to send an external event containing the exit code of the program and a user-defined value. The rule to react to the program exit will thus have two parameters. In the first *integer* parameter the exit code of the program will be given; the second one is an *anyvalue* parameter in which the value given on calling **execute()** is passed on.

Utilization of Parameters

The utilization of the individual parameters depends very much on the platform, as the following table illustrates:

	Synchronous	ExeType	WindowType	External Event
Windows	is supported	is supported	is supported	is supported (not with *ExeType* = *execommand*)
Motif	is ignored	is ignored	is ignored	is supported

On Windows the return value of the command is not waited for with *ExeType* = *execommand*.

On Windows a maximum of 32 dialog processes can be reacted to.

Note on Quotation Marks

Enclosing the *Command* parameter in quotation marks is not necessary, but it is required for the other parameters.

On MICROSOFT WINDOWS, the *Command* parameter is automatically enclosed in double quotation marks for convenience if this parameter contains spaces but no double quotation marks. Additionaly, the *ExeType* parameter must be set to the value *exenormal* or *exeshell* here.

Furthermore, commands composed of the *Command* and *Arguments* parameters are automatically quoted if the *Command* parameter begins with double quotation marks (already indicated or automatically added) and the *ExeType* parameter has the value *exeshell*.

If this behavior is not desired, the *Command* parameter can be left empty (*null* or *""*) and the actual command to be executed – in this case mandatorily – can be passed in the *Arguments* parameter.

Example

```
dialog Exe

on dialog start
{
  variable boolean Started;

  !! start idm without waiting and event
  Started := execute("idm", "-IDMversion");
  print "idm started";
  print Started;

  !! start with event
  Started := execute("idm", "-IDMversion", this, 100, "testmessage");
  print "External event 100 should appear";
  print Started;

  !! start with options for the startwindow
  !! idm ignores this option
  Started := execute ("idm", "-IDMversion", hidewindow, this, 100);
  print "Window should not appear";
  print Started;

  !! start idm synchronous
  Started := execute ("idm", "-IDMversion", true);
  print "Synchronous start";
  print Started;

  !! example with searchsymbol
  Started := execute ("IDM_PATH:idm", "-IDMversion");
  print "Started with searchsymbol";
  print Started;
}

on dialog extevent 100 (integer Exitcode, anyvalue Message)
{
  print "Returnvalue: " + itoa(Exitcode) + " " + Message;
}
```

11.14 exit()

This function can be used for a controlled exiting of all dialogs. All running dialogs are terminated, their *on dialog finish* rules are called. Afterward, the IDM event loop is quit. The dialogs, however, are not deleted.

Definition

```
void exit
(
  { boolean StopDialog := true input }
)
```

Parameters

boolean ***StopDialog*** *:= true input*

If this parameter is set to *true* (default), the behaviour described above occurs. If *false* is passed, however, the event loop is terminated without the dialog being closed, which also means that the *finish* rule of the dialog is not executed.

Example

```
on END_Button select
{
  exit();
}
```

Important

A rule is not exited immediately with the **exit()** function, but is processed to the end. If the rule is to be exited immediately, a `return` has to be given after the **exit()** call.

```
on END_Button select
{
  exit();
  print "Still running";
}

on END_Button select
{
  exit();
  return;
  print "Not running anymore";
}
```

See also

Built-in function **stop()**

11.15 fail()

This function serves to check the correct execution of a function. If the function call in **fail()** returns an error, this functions can avoid that this error will be logged in the error file. The application itself can then react to this error.

Definition

```
boolean fail
(
  anyvalue Expression input
)
```

Parameters

anyvalue *Expression* input

This parameter can contain an arbitrary expression which uses an internal function of the ISA Dialog Manager.

Return value

true

On calling the included function an error has occurred.

false

The function has been executed correctly.

Catching and Passing of Errors

The following applies to checking and catching errors with **fail()** and passing them on with `pass` and `throw`:

- From redefined **:get()** and **:set()** methods as well as from user-defined rules, methods and simulation rules, errors are passed to the caller.
- Errors from the methods **:init()**, **:clean()**, **:setclip()** and **:action()** are not passed to the caller.

Example

A typical use of this function is as follows.

```
if ( fail( atoi(TextVariable) ) )
then
  Value := -1;                  /* error */
else
  Value := atoi(TextVariable);  /* normal */
endif
```

The preceding example sets the variable value to *-1* if the transformation of *TextVariable* to an integer value was not successful.

An error occurs e.g. if *TextVariable* contained "123abc45". Normally *TextVariable* contains "12398745".

The function **fail( )** returns the boolean expression *true*, if the contained function, e.g. atoi (TextVariable), could not be executed. If the function was executed, **fail( )** returns the value *false*.

11.16 find()

This function searches for a specified value in a list of values and returns the first found index where this search value can be found.

The search never includes the default element (e.g. index *[0]* or *[0,0]*).

By specifying a start and end index, the search can be restricted. The valid values are:

- No value specified.
- *integer* value in the range *1 … field size* (but must be > *0*) for collections with data types *vector*, *refvec*, *list* or *hash*.
- *index* value in the range from *[1,1]* to *[rowcount,colcount]* where *rowcount* and *colcount* must also be > *0*.
- Valid index of a *hash*.

When searching for strings it is also possible to search case-insensitive or for the first occurrence as prefix.

Since the indices of hashes are not subject to any order, only an *integer* value in the range *1 … itemcount()* should be specified as index, which indicates the position of the start or end element.

For two-dimensional arrays (similar to the **:find()** method for *.content[]* at the ***tablefield***), specifying an index range *[Sy,Sx]* to *[Ey,Ex]* restricts the search scope to the range *[min(Sy,Ey),min(Sx,Ex)] … [max(Sy,Ey),max(Sx,Ex)]*. Thus the restriction to rows and columns is easily possible.

Definition

```
anyvalue find
(
      anyvalue ListValue  input,
      anyvalue Value      input
  { , anyvalue StartIndex input
  { , anyvalue EndIndex   input } }
  { , boolean  CaseSensitive := false       input }
  { , enum     MatchType     := match_exact input }
)
```

Parameters

anyvalue *ListValue* input

In this parameter, a list value is expected in which to search.

anyvalue *Value* input

The value specified in this parameter is searched for. Only values are compared whose types are "equal" or can be converted (a ***text*** is converted to a *string* for this purpose).

anyvalue ***StartIndex*** *input*

This optional parameter specifies the start index from which to search for the value in the collection. If no value is specified here, *1* is assumed for one-dimensional and *[1,1]* for two-

dimensional collections.

anyvalue ***EndIndex*** *input*

This optional parameter specifies the end index up to which the collection shall be searched for the value. This argument is only permitted if a start index has also been specified.

boolean ***CaseSensitive*** *:= false input*

This optional parameter only applies to *string* values and defines whether the search should be case-sensitive or not. The default value is *false*.

enum ***MatchType*** *:= match_exact input*

This optional parameter only applies to *string* values and defines how to search for strings.

Value range

match_begin
: Finds the first string that starts with the search string.

match_exact
: Finds the first string that exactly matches the string that is searched for.

match_first
: Finds the string whose beginning has the largest match with the search string. If there is a string that matches the search string exactly, its index will be returned.

match_substr
: Finds the first string that contains the string that is searched for.

Return value

0
: Item was not found in the passed list (data types *list*, *vector*, *refvec*).

[0,0]
: Item was not found in the passed matrix.

nothing
: Item was not found in the passed associative array (data type *hash*).

otherwise
: Index of the item in the passed collection. The value returned has the index data type of the collection passed.

Fault behavior

The function call fails if the *ListValue* parameter is not a collection, the index ranges are outside the valid range, or another invalid parameter value exists.

Remark

If the *StartIndex* and *EndIndex* parameters have a data type other than *integer* when searching in hashes, the position of the start and end elements is calculated from the specified indexes. In this respect, searching in hashes with the **find()** function differs from searching in associative arrays with the **:find()** method.

Example

```
dialog D

on dialog start
{
  variable hash Hash := [
    "030"  => "berlin",
    "0711" => "stuttgart",
    "089"  => "munich" ];
  variable matrix Matrix := [
    [1,1] => "area code", [1,2] => "call number",
    [2,1] => 089,         [2,2] => 713400,
    [3,1] => 0711,        [3,2] => 805439 ];
  variable anyvalue Idx;

  Idx := find(Hash, "stuttgart");
  if Idx <> nothing then
    print "Found: stuttgart => " + Idx;
    Idx := find(Matrix, atoi(Idx), [2,1], [3,1]);
    if Idx <> [0,0] then
      print "Call: " + Matrix[first(Idx),2];
    endif
  endif
  exit();
}
```

See also

Built-in function **sort()**

Method :find()

11.17 first()

The function serves to query the first value of an index.

Definition

```
integer first
(
  index Idx input
)
```

Parameters

index *Idx* input

In this parameter, you can declare the index value of which the first part, usually the row part, is to be inquired.

Return value

The first of the two values contained in the index.

Example

```
on Table select
{
  variable integer Row := 0;

  !! Querying the row part of the active cell.
  Row := first(Table.activeitem);
}
```

See also

Built-in function **second()**

11.18 getvalue()

With this function, you can inquire attributes of objects.

You can find the attributes available for the relevant object type in the “Object Reference”.

Definition

```
anyvalue getvalue
(
      object    Obj  input,
      attribute Attr input
  { , integer IntIdx input | index IdxIdx input }
)
```

Parameters

object *Obj* input
This parameter indicates the object whose attribute you want to inquire.

attribute *Attr* input
This parameter indicates the object attribute you want to inquire.

*integer **IntIdx** input*
*index **IdxIdx** input*
With this optional parameter you can declare the index of the requested attribute. You have to declare an integer value if the requested attribute is one-dimensional (e.g. with a listbox or a pop-text). You have to declare an index value if the requested attribute is two-dimensional.

Return value

Value of the requested attribute.

Examples

» Querying the content in the field *1,1* of a tablefield

```
getvalue(Tablefield, .content, [1,1]);
```

» Querying the content of an edittext

```
getvalue(Edittext, .content);
```

» With this function you can also change attributes via variables, e.g.

```
rule void Set(attribute A)
{
  setvalue(Win1, A, true);
  Pb1.sensitive := getvalue(Win2, A);
}
```

See also

Method :get()

C function DM_GetValue in manual “C Interface - Functions”

11.19 getvector()

This function may be used to obtain the entire value of vector or matrix attributes (both user-defined and predefined), i.e. a list of all indexed values, or a section thereof, in a *vector*.

Definition

```
vector getvector
(
      object    Object     input,
      attribute Attribute  input
  { , anyvalue  FirstIndex input
  { , anyvalue  LastIndex  input } }
)
```

Parameters

object *Object* input

This parameter defines the object whose attribute values shall be queried.

attribute *Attribute* input

This parameter defines the attribute to be queried.

anyvalue ***FirstIndex*** *input*

This optional parameter defines the start index as of which the element values are retrieved. For one-dimensional array attributes an *integer* value should be specified, for two-dimensional arrays an *index* value. The default value for one-dimensional array attributes is the *integer* value *1*, for two-dimensional arrays the *index* value *[1,1]*.

anyvalue ***LastIndex*** *input*

This optional parameter specifies the end index up to which the values from the attribute are taken into the resulting value list. Again, an *integer* value is expected for one-dimensional array attributes and an *index* value for two-dimensional arrays. If no *LastIndex* parameter is specified, all values up to the end of the array are included. The *LastIndex* value should be after the *FirstIndex* value.

Return value

The indexed values determined from the object attribute are returned as a value list with type *vector*.

Fault behavior

The function call fails for an invalid object or attribute or for an invalid index range.

Example

```
dialog D
{
  string Extra[integer];
  .Extra[1] := "Anna";
```

```
  .Extra[2] := "William";
}

window Wi
{
  .title "Names";

  listbox Lb
  {
     .xauto 0;  .yauto 0;
     .content[1] "..NAMES..";
     .content[2] "Irene";
     .content[3] "David";
     .content[4] "Henry";
  }

  on close { exit(); }
}

on dialog start {
  variable vector Names;

  Names := join(getvector(Lb, .content, 2), getvector(D,.Extra));
  setvector(Lb, .content, sort(Names), 2);
}
```

See also

Built-in function **setvector()**

C functions DM_GetVectorValue(), DM_SetVectorValue()

11.20 indexat()

This function returns the index of a collection at a specific position. The allowed positions are *1 … itemcount()* and thus allow a loop through all indexed values.

For values of type *list*, *refvec* and *vector*, the position is identical to the actual index. For a *hash* value, there is no defined order of the indexes returned. For *matrix* values, the increasing positions are mapped in a sequence where all (column) values of a row are arranged in ascending order.

The function returns an error (`fail`) if the position is outside the allowed range or if the *Value* parameter is not a collection.

Definition

```
anyvalue indexat
(
  anyvalue Value input,
  integer  Pos   input
)
```

Parameters

anyvalue *Value* input
This parameter specifies the value list from which the index shall be queried.

integer *Pos* input
Position for which the index value shall be determined.

Return value

Index value at the position passed as parameter.

Example

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france" ];
  variable integer Pos;

  /* print the Matrix elements [1,1] [1,2] ... [2,2] */
  for Pos:=1 to itemcount(Matrix) do
    print sprintf("%s : %s", indexat(Matrix, Pos), valueat(Matrix, Pos));
  endfor
  exit();
```

```
}
```

See also

Built-in functions itemcount(), **keys()**, **valueat()**

Method index

C function DM_ValueIndex()

11.21 inherited()

This function checks whether the value is inherited. *true* is returned, if the attribute value to be read out last was inherited by a model; otherwise, false is returned.

Definition

```
boolean inherited
(
)
```

Return value

true
: The attribute value which was inquired last has been inherited.

false
: The attribute value which was inquired last has not been inherited.

Example

Inquiry of the contents of an input field and checking afterward if this value has been inherited from the underlying model.

```
print Edittext.content;
print inherited();
// true if the content was inherited from the Model or Default.
```

11.22 insert()

With this function a new value can be inserted into a collection (data types *hash*, *list*, *matrix*, *refvec* and *vector*) or a string once or several times.

The new values are inserted before the specified position. The position must be >= *1*, i.e. it is not possible to insert default values at the indices *[0]* or *[0,0]* with this function. The largest possible position is *1* higher than the current *countof(ListValue)* to enable appending to an existing collection.

In general, the data type of the (optional) new value has to be a scalar and match the value type of the collection.

The number of repetitions for the inserting (*Count* parameter) has to be >= *0*.

In case of an error, the function call is aborted with a `fail`.

Definition

```
anyvalue insert
(
      anyvalue ListValue  input,
      integer  Pos   input
  { , integer  Count := 1 input }
  { , enum     Dir   := dir_row input }
  { , anyvalue NewValue   input }
)
```

Parameters

anyvalue *ListValue* input

This parameter specifies the collection or string where a value shall be inserted.

integer *Pos* input

In this parameter, the index position is defined before which the insertion occurs. Values >= *1* are permitted. For *matrix* collections, depending on the *Dir* parameter, *Pos* represents either a row position (*dir_row*) or a column position (*dir_column*).

*integer **Count** := 1 input*

This optional parameter defines the number of times a new value (respectively how many rows or columns) shall be inserted. Values >= *0* are allowed, with no insertion being performed for *0*.

*enum **Dir** := dir_row input*

This optional parameter defines the orientation of the index position, which is important for a matrix. The default value *dir_row* means that rows are inserted in the matrix. With *dir_column* columns are inserted.

*anyvalue **NewValue** input*

This optional parameter defines the new value to be inserted. The value must match the value type of the collection.

Return value

The function returns the modified collection respectively the modified string.

Particularities

» *hash*
Since this collection data type does not have a defined index range, inserting takes place based on the *integer* indices. Thus a *hash* with an indexing of *1,2,3…* is assumed. There are no limitations concerning the largest possible index position.

» *matrix*
Inserting into a matrix always happens by rows or columns. This is controlled by the *Dir* parameter. By default (*dir_row*) new rows are inserted. If *dir_column* is given, new columns are inserted. This also means that the position refers to a row in the first case and to a column in the second.

» *refvec*
Inserting *null* objects is not allowed. Multiple insertion is also not supported, since the same object ID may only be contained once. If the object ID already exists, it is repositioned.

» *string*
Here the position refers to the character position in the string, i.e. the string is treated as an array of characters.

» *Default Values*
In general it is useful to give a new value when inserting. If this is omitted, it depends on the collection type which values will be at the inserted positions. The collection types *list*, *hash* and *matrix* allow default values, so that the default values appear at the inserted positions when inserting without a new value.

Examples

» Multiple insertion of the string *"X"* into a list

```
variable list L := ["a", "b", "c"];
print insert(L, 1, 2, "X");
```

Output

```
["X","X","a","b","c"]
```

» Appending a row to the end of a matrix

```
variable matrix M := [ [0,0]=>"?", [1,1]=>"a1", [1,2]=>"b1",
                       [2,1]=>"a2", [2,2]=>"b2" ];
M := insert(M, 3, "NEW");
print M;
```

Output

```
[[1,1]=>"a1",[1,2]=>"b1",[2,1]=>"a2",[2,2]=>"b2",[3,1]=>"NEW",
[3,2]=>"NEW"]
```

Inserting a new 1st column without value

```
M := insert(M, 1, dir_column);
print M;
```

Output

```
[[1,1]=>"?", [1,2]=>"a1",[1,3]=>"b1",[2,1]=>"?",[2,2]=>"a2",[2,3]=>"b2",
[3,1]=>"?",[3,2]=>"NEW",[3,3]=>"NEW"]
```

» Inserting *"ll"* into a string

```
print insert("heo", 3, 2, "l");
```

Output

```
"hello"
```

» Inserting into a hash with *integer* indexing

```
variable hash H := [2=>.xleft, 33=>.ytop];
H := insert(H, 10, .width);
print H;
```

Output

```
[2=>.xleft,10=>.width,34=>.ytop]
```

» Inserting into a *vector* list

```
variable vector[boolean] V := [true, true];
print insert(V, 2, 3, false);
```

Output

```
[true,false,false,false,true]
```

» Inserting an already existing *refvec* item at another position

```
variable refvec R := [PbApply, PbCancel, PbRevert];
print insert(R, 1, PbRevert);
```

Output

```
[pushbutton D.PbRevert,pushbutton D.PbApply,pushbutton D.PbCancel]
```

Availability

Since IDM version A.06.02.g

See also

Built-in functions **append()**, **delete()**

11.23 itemcount()

This function returns the number of indexed values in a collection. The default value(s) are excluded from that number. Together with **indexat()** and **valueat()** it is thus easy to loop through any collection.

Typically, **countof()** and **itemcount()** return the same result for values of the data types *refvec*, *vector*, and *list*. However, **itemcount()** offers the more generic use, whereas **countof()** is better suited for structured, type-adapted use.

Definition

```
integer itemcount
(
  anyvalue Value input
)
```

Parameters

anyvalue *Value* input
This parameter specifies the value for which the number shall be determined.

Return value

0
The passed value is scalar or an empty collection.

1 … 2^{31}
Number of values in the collection without the default values.

Example

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france"
    /* [2,2] => inherited from default [0,0] */ ];
  variable integer I;
  variable anyvalue Idx;

  /* print the Matrix values [1,1] [1,2] ... [2,2] */
  for I:=1 to itemcount(Matrix) do
    Idx := indexat(Matrix, I);
    print sprintf("%s : %s", Idx, Matrix[Idx]);
  endfor
  exit();
```

```
}
```

Output

```
"[1,1] : germany"
"[1,2] : berlin"
"[2,1] : france"
"[2,2] : -?-"
```

See also

Built-in functions **countof()**, **indexat()**, **valueat()**

Method index

Attribute itemcount

C function DM_ValueCount()

11.24 itoa()

This function converts a number to a string (**i**nteger **to a**scii).

Definition

```
string itoa
(
  integer IntValue input
)
```

Parameters

integer IntValue input

In this parameter the figure which is to be turned into a string is indicated.

Return value

As a result the function issues the figure which has been turned into a string.

Example

Filling an input field with a figure.

```
Edittext.content := itoa(5);
```

11.25 join()

This function creates a new value list from the specified parameters. The type of the value list is determined by the first call parameter. If this is a collection or a collection data type (*list*, *vector*, *refvec*, *hash*, *matrix*), the resulting value list has the same type. A data type as the first parameter value never is included in the result list. If the specified parameters are collections, only the values (without default values) are used. Scalar values are added to the resulting value list.

The order of the values in the resulting value list will correspond to the order of the parameters and for collections to their "natural" order (see **indexat()**).

Indexing is usually done by incrementing the last index value or the number of values (see **countof()** and **itemcount()**). That is, the values in the result list usually get an indexing by *1, 2, 3, …* or *[1,1], [1,2], [1,3], ….*

If the result list type and the parameter value are both a *hash*, the index of the hash values is adopted. Added scalar parameters will get the increased **itemcount()** value of the result list as index.

When using *refvec* as the result list type, it should be noted that an error (`fail`) occurs if a value is not of the data type *object*.

Definition

```
anyvalue join
(
      anyvalue Value input
  { , anyvalue Par2  input }
  ...
  { , anyvalue Par16 input }
)
```

Parameters

anyvalue *Value* input

This parameter specifies the value on which the function is applied. The parameter is also used to determine the type for the result list.

*anyvalue **Par2** input*

…

*anyvalue **Par16** input*

Additional optional parameters on which the function is applied.

Return value

A collection containing all values from the passed parameters. If the first parameter is a collection data type or a collection, the returned collection has this data type. A data type as the first parameter is not included in the returned collection.

Änderung ab IDM-Version A.06.01.b bei der Übergabe des Datentyps *string* als ersten Parameter

Modification as of IDM Version A.06.01.b When Passing the Data Type *string* as First Parameter

Wird der Funktion **join()** als erstes Argument der Datentyp *string* übergeben, dann werden die nachfolgenden Argumente in den Datentyp *string* konvertiert und anschließend verkettet. Sind die nachfolgenden Argumente Sammlungen, werden zunächst deren Werte in der natürlichen Reihenfolge des Index zu einem String verkettet.

If the data type *string* is passed as first argument to the function **join()**, the subsequent arguments are converted to the data type *string* and then concatenated. If the subsequent arguments are collections, first their values are concatenated to a string in the natural order of the index.

Falls das erste Argument ein Datentyp, aber weder *string* noch ein Sammlungsdatentyp ist, gibt **join()** einen Fehler zurück.

If the first argument is a data type that is neither *string* nor a collection data type, **join()** returns an error.

Example

```
dialog D

on dialog start
{
  variable hash DomainHash := [
    ".de" =>" germany",
    ".us" => "usa",
    ".fr" => "france",
    ".uk" => "united kingdom" ];
  variable list Countries;
  variable hash Domains;

  /* join the values to a list */
  Countries := join(list, DomainHash, "norway", "spain");
  /* join the values and keys into a hash */
  Domains := join(DomainHash, [".us" => "united states of america"]);

  print Countries;
  print Domains;

  exit();
}
```

Output

```
["norway","spain","germany","usa","france","united kingdom"]
[".de"=>"germany",".us"=>"united states of america",".fr"=>"france",
".uk"=>"united kingdom"]
```

See also

Built-in function **keys()**

C function DM_ValueChange()

11.26 keys()

This function returns a list of all index values in a collection. The indexes of the default values are not included in the list. If a scalar is specified as parameter, an empty list is returned.

Definition

```
list keys
(
  anyvalue Value input
)
```

Parameters

anyvalue *Value* input

This parameter specifies the value on which the function is applied.

Return value

A list containing all indexes of a collection.

Example

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france" ];
  variable anyvalue Idx;

  /* print the Matrix elements [1,1] [1,2] ... [2,2] */
  foreach Idx in keys(Matrix) do
    print sprintf("%s : %s", Idx, Matrix[Idx]);
  endfor
  exit();
}
```

See also

Built-in functions **countof()**, **indexat()**, **itemcount()**, **values()**

Chapter “foreach Loop”

11.27 length()

This function returns the length of a string.

Definition

```
integer length
(
  string Str input
)
```

Parameters

string *Str* input

In this parameter the string whose length is to be inquired is indicated.

Return value

As a result this function returns the string length.

Example

The following line is given in the DM program:

```
WnMain.title := "Example";
```

Then by

```
Len := length ( WnMain.title );
print Len;
```

or by

```
Len := length ( "Example" );
print Len;
```

the value of the “Len” is 7.

11.28 load()

With this function, a dialog is loaded, but not yet started.

Definition

```
object load
(
  string Filename input
)
```

Parameters

string *Filename* input

In this parameter the file name of the dialog to be loaded is indicated. This indication can be made via the usual mechanism in the IDM
<environment variable>:<filename>
to make the loading process as flexible as possible. The environment variable will be interpreted as path in which the indicated file is to be searched for.

Return value

DialogID
Here the ID of the newly loaded dialog has been returned.

null
The dialog could not be loaded.

Example

Loading a dialog and starting the newly loaded dialog.

```
variable object Dialog := null;

Dialog := load("SearchPath:Test.dlg");
!! checking whether dialog could be loaded
if Dialog <> null then
  !! starting the dialog
  run (Dialog);
endif
```

See also

Built-in function **run()**

C function DM_LoadDialog in manual “C Interface - Functions”

11.29 loadprofile()

This function reads in the values of the configurable ***record*** instances (*.configurable = true*) and **global variables** (declared with **config**) of a dialog or module from a configuration file (profile).

Definition

```
boolean loadprofile
(
      string Filename  input
  { , object Module := null input }
)
```

Parameters

string *Filename* input

This parameter defines the file name of the configuration file. A file path can be specified which may also contain an environment variable.

*object **Module** := null input*

This optional parameter contains the identifier of the dialog or module whose ***record*** and variable values are to be read in from the file.
With *Module = null*, the module ID of the current rule is used.

Return value

true

Reading in the values from the configuration file has been successful.

false

The values could not be read in.
This may be due to errors accessing the file or an invalid module ID.

Availability

Since IDM version A.06.02.g

See also

Built-in function **saveprofile()**

C function DM_LoadProfile()

11.30 max()

This function returns the largest integer value found in all its call parameters.

The call parameters may be scalar integer values or collections (*vector*, *list*, *matrix*, *hash*). For the latter, the indexed elements (without the default values) are used to determine the maximum value.

Definition

```
integer max
(
      anyvalue Par1  input
  { , anyvalue Par2  input }
  ...
  { , anyvalue Par16 input }
)
```

Parameters

anyvalue *Par1* input

*anyvalue **Par2** input*

...

*anyvalue **Par16** input*

These parameters specify the values on which the function is applied.

Return value

The largest integer value to be found will be returned.

Fault behavior

The function call fails if the arguments do not contain a scalar integer value or if arguments or their elements contain a value that is not an integer value.

Example

```
dialog D

on dialog start
{
  variable list List := [17, 3, 23, 5];

  print max(List, 24, 4);  /* print 24 */
  exit();
}
```

See also

Built-in function **min()**

11.31 min()

This function returns the smallest integer value found in all its call parameters.

The call parameters may be scalar integer values or collections (*vector*, *list*, *matrix*, *hash*). For the latter, the indexed elements (without the default values) are used to determine the minimum value.

Definition

```
integer min
(
      anyvalue Par1  input
  { , anyvalue Par2  input }
  ...
  { , anyvalue Par16 input }
)
```

Parameters

anyvalue *Par1* input

*anyvalue **Par2** input*

...

*anyvalue **Par16** input*

These parameters specify the values on which the function is applied.

Return value

The smallest integer value to be found will be returned.

Fault behavior

The function call fails if the arguments do not contain a scalar integer value or if arguments or their elements contain a value that is not an integer value.

Example

```
dialog D

on dialog start
{
  variable list List := [17, 3, 23, 5];

  print min(List, 24, 4);  /* print 3 */
  exit();
}
```

See also

Built-in function **max()**

11.32 parsepath()

With this function an object can be requested with the help of the object name.

Definition

```
object parsepath
(
      string  ObjectName input
  { , object  Parent := null input }
  { , object  Dialog := null input }
  { , integer Index  := 0 input }
)
```

Parameters

string *ObjectName* input

This parameter indicates the name of the searched object.
The value specified in the *ObjectName* parameter represents a search path starting from the object specified in the *Parent* parameter. An empty string in the parameter *ObjectName* (*""*) corresponds to an empty search path.

object ***Parent*** *:= null input*

This optional parameter denotes the parent of the searched object. If this parameter is indicated, the given object will be searched for below the object. If the *null* object is specified for this parameter, the object will be searched for in the indicated dialog or in the current dialog. If resources are searched for, this parameter has to be specified by the *null* object.
If functions in applications are to be searched, the application has to be given.

object ***Dialog*** *:= null input*

This optional parameter denotes the dialog of the searched object. If this parameter is not specified, the object will be searched for in the current dialog. If an object is to be searched in the module, the corresponding module must be specified in this parameter.

integer ***Index*** *:= 0 input*

If *Index* is greater than *0*, then all children in the child children of the object are searched for that match the corresponding *ObjectName*. The index > *0* tells the function what number of occurrences of *ObjectName* to search for. If no child exists for this index, then *null* is returned.

Return value

ObjektID

The searched object will be returned.

null

The specified object could not be found or the object name is not unique.
This means that there is either none or more than one object with the specified name.

Examples

» Search for the color “RED” in the current dialog

```
Obj := parsepath("RED");
```

» Search for the rule “Test” in the module “M1”

```
Obj := parsepath("Test", null, M1);
```

» Search for the pushbutton “OK” in the window “W1”

```
Obj := parsepath("OK", W1);
```

» Search for the function “ApplFunc” in the application “A1”

```
Obj := parsepath("ApplFunc", A1);
```

» Search for the I-th occurrence of “SubR” in the window “W”

```
dialog D
window W
{
  record R
  {
    record SubR {}
  }
  child groupbox G
  {
    record SubR {}
  }
  rule void DoSomething()
  {
    variable integer I := 1;
    variable object O := D;
    while (O) do
      O := parsepath("SubR", this, null, I);
      if O <> null then
        I := I + 1;
        print O;
      endif
    endwhile
  }
}
on dialog start
{
  W:DoSomething();
}
```

See also

C function DM_ParsePath in manual “C Interface - Functions”

11.33 print()

This function provides the screen output of any values defined in the dialog during the test of the dialog development.

Note

On calling this function the braces may be omitted.

Definition

```
void print
(
  anyvalue PrintValue input
)
```

Parameters

anyvalue *PrintValue* input

In this parameter the value which is to be printed out is indicated.

Example

The following lines are given in the DM program:

```
WnMain.title := "XYZ";
Counter      := 650;
```

Then with the statements

```
print WnMain.title;
print Counter;
```

the folllowing is output in the error file:

```
WnMain.title = "XYZ"
Counter = 650
```

11.34 querybox()

With the built-in function **querybox()** messageboxes and dialogboxes (windows with the *.dialogbox* attribute set to *true*) can be opened. The processing of rules in the ISA Dialog Manager is interrupted until the messagebox or dialogbox has been closed.

Definition

```
anyvalue querybox
(
      object  Messagebox input
  { , object  Parent     input }
  { , boolean ShipEvent  := true input }
)
```

Parameters

object *Messagebox* input

In this parameter the messagebox or dialogbox to be opened is passed to the function.

*object **Parent** input*

This optional parameter indicates the parent object above which the messagebox is to be opened. This parameter may be a window as well as a null-ID. The messagebox will be displayed centered on the parent window, if the window system allows it. Otherwise the position will be determined by the window system itself (e.g. screen center).
The parameter is ignored for dialogboxes.

*boolean **ShipEvent** := true input*

This parameter controls, whether a *changed* event for *.visible* is triggered for the dialogbox. When the parameter is set to *true* (default), an event is triggered. With *false*, the event is canceled.

Return value

Messageboxes

button_abort
The ***messagebox*** was closed using the “Abort” button.
button_cancel
The ***messagebox*** was closed using the “Cancel” button.
button_ignore
The ***messagebox*** was closed using the “Ignore” button.
button_no
The ***messagebox*** was closed using the “No” button.
button_ok
The ***messagebox*** was closed using the “OK” button.
button_retry
The ***messagebox*** was closed using the “Retry” button.

button_yes
The ***messagebox*** was closed using the “Yes” button.
nobutton
The ***messagebox*** has not been opened due to an error.

Dialogboxes

For **dialogboxes**, the return value is set through the **closequery()** function.

Example

Opens a messagebox and queries which one of the provided pushbuttons the user has selected.

```
!! Program waits until the user has chosen a pushbutton
if button_ok = querybox(MyMessagewindow) then
  !! User wants the action to be executed
```

11.35 queryhelp()

Without argument or with *null* as an argument this function will switch the IDM into context help mode, where the next mouse click on an object triggers a *help* event on this object.

If an object ID is passed with the call, the function triggers a *help* event for that object.

Definition

```
void queryhelp
(
  { object Id := null input }
)
```

Parameters

object ***Id*** *:= null input*

In this optional parameter an object can be passed for which a *help* event shall be triggered.

Availability

Since IDM version A.06.02.g

11.36 random()

This function generates an integral random number.

Definition

```
integer random
(
  integer Range := 100 input
)
```

Parameters

*integer **Range** := 100 input*

This parameter specifies the upper limit for the random number. The generated number will be less than this limit.

Return value

The function returns an integral number r with 0 <= r < Range.

Example

```
!! returns an integer value in the range 0-4
random(5);
```

11.37 regex()

With this function a Regular Expression can be applied to a string or a list of strings.

In order to provide a maximum of functionality, the IDM dynamically integrates the free PCRE library. This enables pattern expressions analog to PERL. If you use this function in your product, you should pay attention to the license terms and documentation of PCRE (**www.pcre.org**). With regard to linking, chapter “PCRE Library for Support of Regular Expressions” should be observed. ISA recommends a PCRE library version *8.** for proper working. The PCRE functions are used for pattern matching and capturing of string parts. The IDM then handles evaluation and replacement.

Syntactically the following Regular Expressions are accepted by the IDM to enable the operations “matching” and “substitution”:

» matching

```
m/<pattern>/<modifiers>
```

» substitution

```
s/<pattern>/<replacement>/<modifiers>
```

The following <*modifiers*> are supported for configuring the operation:

s	single-line string (PCRE_DOTALL)
m	multi-line (PCRE_MULTILINE)
i	ignore case (PCRE_CASELESS)
g	global search – pattern matching is repeated
x	extended (PCRE_EXTENDED)
f	first line (PCRE_FIRSTLINE)
W	unicode (wide) character classes (PCRE_UCP)
X	extra (PCRE_EXTRA)
U	ungreedy (PCRE_UNGREEDY)

The parentheses indicate how the modifiers are passed to PCRE. This means that they are not processed by the IDM, but by the PCRE library.

The modifiers “o” and “e” are skipped without error message.

The actual application of <*pattern*> and <*replacement*> then happens through the PCRE library. More details and information on the Regular Expressions may therefore be found in the documentation of the PCRE library used.

The IDM also allows to split the parts <*pattern*> and <*replacement*> of the Regular Expression into two separate parameters, but in this case no <*modifiers*> are possible. Usually, the operation

determines the type of results returned. However, this can be controlled by an *Action* parameter, e.g. to return only the number of matches found, to achieve filtering or to get the values of the *<pattern>* variables.

These are the available actions with their corresponding evaluations:

***Table 2:** Actions, return types and evaluations of the regex function*

Action	Return Type	Evaluation
regex_eval	*boolean string list[string]*	Depending on the operation it is either returned whether at least one of the strings matches the Regular Expression (matching operation *true* or *false*). Or in case of substitution the replaced string(s) will be returned, if the pattern matches, otherwise the original string.
regex_match	*boolean list[string]*	Performs a check of the pattern only and returns *true* if the pattern matches, otherwise *false*. For a string list, only strings that match the pattern will be included in the result list.
regex_unmatch	*boolean list[string]*	Performs a check of the pattern only and returns *false* if the pattern matches, otherwise *true*. For a string list, only strings that do **not** match the pattern will be included in the result list.
regex_count	*integer*	Counts the number of matches found in the string or string list. For each string, a maximum of *+1* is counted, so that the result is *0* or *1* when applied to a single string and a value in the range *0 … itemcount()* when applied to a string list.
regex_locate	*integer list [integer]*	Returns the character position of the first match when applied to a single string. For a string list, the index positions in the list where the pattern matches are returned.

The IDM supports application of the **regex** function on any collection data type (*list*, *hash*, *matrix*, *vector*). However, a generated return list is always of the type *list* without a special indexing of the source list beiing adopted. Before the Regular Expression is applied, values that are not of the data type *string* are converted into a string like it happens with *print <Value>;* for instance.

Definition

```
anyvalue regex
(
      anyvalue StringOrList input,
      string   Pattern input
  { , string   Replace input }
  { , enum     Action  := regex_eval input }
)
```

Parameters

anyvalue *StringOrList* input

This parameter contains the string or the list of strings that the Regular Expression is applied on.

string *Pattern* input

This parameter contains either the Regular Expression or the pattern string (<*pattern*>) if the regular expression is split into <*pattern*> and <*replacement*>.

string ***Replace*** *input*

This optional parameter should hold the replacement string (<*replacement*>) if the Regular Expression is split into <*pattern*> and <*replacement*> for the function call.

enum ***Action*** *:= regex_eval input*

This optional parameter controls the results evaluation. These are the available actions:

regex_eval (default)
: Evaluation of the Regular Expression (matching or substituition).

regex_match
: Filtering for strings that match the pattern.

regex_unmatch
: Filtering for strings that do **not** match the pattern.

regex_count
: Number of matches found (*+1* for each search string).

regex_vars
: Returns the contents of all variables defined by the search pattern.

regex_locate
: Returns the character position or index position of the match found.

Return value

Return value and type depend on the evaluation action and are explained in “Table 2” (above).

Examples

1. Test for digits in a string

```
print regex("Is 127 a number?", "\\d+");
```

Output

```
true
```

2. Replace all decimal numbers with an “N”

```
print regex("42 is greater than 10", "s/(\d+)/N/g");
```

Output

```
"N is greater than N"
```

3. Output only strings that match the pattern

```
print regex("127,5", "^\\d+,\\d+$", regex_match);
print regex("1275", "^\\d+,\\d+$", regex_match);
print regex(["3,7", "17,5", "0", "21,03"], "^\\d+,\\d+$", regex_match);
```

Output

```
"127,5"
""
["3,7","17,5","21,03"]
```

4. Applying multiple Regular Expressions on a list
   - » list only values that contain a word
   - » count the number of words
   - » surround each item with “>> <<”
   - » list all birth years

```
variable list BirthDays := ["12-13-1973", "Amy", "1-7-1965", "Tom"];
print regex(BirthDays, "^\\w+$", regex_match);
print regex(BirthDays, "^\\w+$", regex_count);
print regex(BirthDays, "s/(.*)/>> $1 <</", regex_match);
print regex(BirthDays, "s/\\d+-\\d+-(\\d+)/$1/", regex_match);
```

Output

```
["Amy","Tom"]
2
[">> 12-13-1973 <<",">> Amy <<",">> 1-7-1965 <<",">> Tom <<"]
["1973","1965"]
```

5. List the variable values contained in a Regular Expression

```
print regex("+2500 dollars or more", "(\\d+)\\s+(\\w+)", regex_vars);
```

Output

```
["2500 dollars","2500","dollars"]
```

6. List the access indexes for the found matches in a list or string

```
variable list Locales := [ "de_DE.UTF8", "C.UTF-8", "de_AT.utf8",
                           "en_AU.utf8", "en_ZM", "POSIX", "de_CH.uf8" ];
print regex(Locales, "/^de_/", regex_locate);
print regex("Hello World", "/W/", regex_locate);
```

Output

```
[1,3,7]
7
```

7. Utilizing automatic conversion of values in a list into *string* values

```
record Rec4711 {}
print regex([123, winsys_x11, "Bond 007", opt_w2kprefsize_compat,
Rec4711],
            "s/\\d+/N/g");
```

Output

```
["N","winsys_xN","Bond N","opt_wNkprefsize_compat","RecN"]
```

Availability

Since IDM version A.06.02.g

PCRE Library for Support of Regular Expressions

To use Regular Expressions through the built-in function **regex()** or as a format in IDM, the free library PCRE (Perl Compatible Regular Expression, see also **www.pcre.org**) is required. Therefore, when using this feature in a product, the license terms of PCRE should be respected.

The IDM needs a PCRE library version *3* or higher with enabled Unicode support and the “standard” PCRE interface. The PCRE2 interface introduced with PCRE version *10* is not yet supported. The **latest stable version *8.**** of the PCRE library is recommended. Typically, most current Linux distributions are already equipped with the PCRE library by default or provide a trouble-free later installation. For the use on Windows, apart from compiling the library on your own, it may also be convenient to download a precompiled library, e.g. from **www.pcre.org** or **www.airesoft.co.uk**.

Important

Depending on the version, a varying feature set and error status of the PCRE library is always to be expected. Please note that ISA cannot give any warranty for the PCRE library and its functions.

The PCRE library is usually linked dynamically by searching for the functions **pcre_compile**, **pcre_study**, **pcre_exec**, **pcre_version** and **pcre_free**. The IDM passes strings in UTF8 encoding, hence the linked PCRE library should also have UTF8 support.

The following linking types and associated search orders are permitted by the IDM:

Table 3: *Linking types and search orders for the PCRE library*

Linking Type		Windows	Unix/Linux[1]
E	Function search directly in the executable		
A	Application-oriented (relative to the path of the application)	pcre3.dll dll\pcre3.dll pcre.dll dll\pcre.dll	pcre.(so|sl) lib/pcre.(so|sl) ../lib/pcre.(so|sl)[2]
S	System-specific library search (e.g. using the path variables PATH or LD_LIBRARY_PATH)	pcre3.dll pcre.dll	pcre.(so|sl)

When building an application with the IDM libraries, it is attempted to link in the order E – A – S. Thus, the easiest way to provide your own IDM application with Regular Expression support is to place the dynamic PCRE library next to the executable. Otherwise, the library existing in the system will be used.

The IDM applications supplied by ISA for development and simulation (IDM, RIDM*, IDMED and Debugger) already have the PCRE library built in statically and use the search A – E – S for binding, so that the use of an external PCRE library is possible as well.

If a static linking is also wanted for your own IDM application, the following should be noted: If the application is linked without referencing the PCRE functions, it must be pulled in completely (typical linker options are e.g. **--whole-archive**, **+forceload** or **/opt:notref**) and it has to be ensured that the PCRE functions are found by the system-specific function pointer search (this may require to export the functions of the application). However, ISA recommends linking via an external library (DLL, Shared Library), in order to facilitate an exchange of the PCRE version in your own product distribution.

The order for linking the PCRE library can be controlled by the application programmer through the interface function DM_Control or DM_ControlEx with the action *DMF_PCREBinding*.

[1]**Depending on the respective platform, the extension ".so" or ".sl" is used for the file name.**

[2]The search via *../lib/* will only take place if the application resides in a directory with the name "bin".

11.38 run()

With this function, a dialog can be started.

Definition

```
void run
(
  object Dialog input
)
```

Parameters

object *Dialog* input

In this parameter the dialog to be started is specified.

Example

Loading a dialog and starting the newly loaded dialog.

```
variable object Dialog := null;

Dialog := load("SearchPath:Test.dlg");
!! checking whether dialog could be loaded
if Dialog <> null then
  !! starting the dialog
  run (Dialog);
endif
```

See also

Built-in function **load()**

C function DM_StartDialog in manual “C Interface - Functions”

11.39 save()

With this function an object can be saved. This will only be done, if the relevant dialog is loaded in ASCII-form. If the dialog is loaded from a binary file, the command will not be executed and the function returns *false*.

Definition

```
boolean save
(
      object SaveObj  input
  { , string Filename input }
)
```

Parameters

object *SaveObj* input

In this parameter the object to be saved is specified.

*string **Filename** input*

In this optional parameter the name of the file is specified in which the object and its children will be saved. If this parameter is not specified, the object will be saved in the log file.

Return value

true

The object has been saved in the specified file.

false

An error has occurred during saving.

Note

This function only works in the development version of the ISA Dialog Manager. In the runtime version the return value is always *false*.

Example

An object is to be examined with regard to its attributes. To do so, it will be saved in the log file.

```
save (this);
```

11.40 saveprofile()

This function writes the current values of all configurable ***record*** instances (*.configurable = true*) and **global variables** (declared with **config**) of a dialog or module into a configuration file (profile), from which they can be reloaded using the function **loadprofile()**.

For ***records***, only values that are not inherited are written into the file by default. In order to also write the inherited values into the file, the parameter *All* needs to be set to *true*.

Only values from the indicated dialog or module are saved. ***Records*** and variables imported from other modules are omitted.

Definition

```
boolean saveprofile
(
      string  Filename   input
  { , object  Module  := null input }
  { , string  Comment := ""   input }
  { , boolean All :=  false   input }
)
```

Parameters

string *Filename* input

This parameter defines the file name of the configuration file. A file path can be specified which may also contain an environment variable.

*object **Module** := null input*

This optional parameter contains the identifier of the dialog or module whose ***record*** and variable values are to be written into the file.
With *Module = null*, the module ID of the current rule is used.

*string **Comment** := "" input*

In this optional parameter a text can be specified, which is written as a comment into the configuration file.

*boolean **All** := false input*

If this optional parameter is set to *true*, the inherited values are also written to the configuration file.
With the default value *false*, only are saved that are not inherited.

Return value

true

Saving the values in the configuration file has been successful.

false

The values could not be saved.
This may be due to errors accessing the file or an invalid module ID.

Availability

Since IDM version A.06.02.g

See also

Built-in function **loadprofile()**

C function DM_SaveProfile()

11.41 second()

The function serves to query the second value of an index.

Definition

```
integer second
(
  index Idx input
)
```

Parameters

index *Idx* input

This parameter denotes the index from which the second part, usually the column part, is to be extracted.

Return value

As a result of this function the second of the two values specified in the index is returned.

A typical application of this function is for example the query for the second index value (indicating the column number) of the attribute *.focus* in the tablefield.

Example

In the dialog a tablefield with vertical dynamic direction has been defined. On a mouse click the selected row and column are to be calculated.

```
on Table select
{
  variable integer Row;
  variable integer Column;

  Row    := first(thisevent.index);
  Column := second(thisevent.index);
}
```

See also

Built-in function **first()**

11.42 sendevent()

You can send **external events** in rules to other objects by using the function **sendevent()**. For mapping serves the *EventID* parameter, which may be assigned a number as well as any arbitrary value (e.g. a previously defined ***message*** resource). On the object given in the *Object* parameter a respective external event must be defined.

Definition

```
void sendevent
(
      object   Object  input,
      anyvalue EventID input
  { , anyvalue Arg1    input }
      ...
  { , anyvalue Arg14   input }
)
```

Parameters

object *Object* input
Object to which the external event is sent.

anyvalue *EventID* input
Unique identifier of the event.

*anyvalue **Arg1** input*
…
*anyvalue **Arg14** input*
Parameters of the external event.

Remarks

When objects, e.g. message resources, are used for the *EventID* parameter, these objects must not be destroyed **before** the asynchronous processing of the event. Otherwise the event cannot be mapped anymore.

Furthermore it has to be kept in mind, that the used event (see also chapter “External Events”) requires at most 14 parameters as imposed through the 16 parameter limit of the Rule Language, more parameters cannot be passed by the **sendevent()** function (the first two parameters are already occupied by the object and the event).

Please avoid any kind of cycles in messages. This may cause endless loops.

Example

```
dialog ExampleDialog

on dialog start
```

```
{
  sendevent(ExampleDialog, 42, "Answer");
}

on dialog extevent 42 (string Question)
{
  print Question;
  exit();
}
```

See also

Built-in function **sendmethod()**

C functions DM_QueueExtEvent() and DM_SendEvent() in manual “C Interface - Functions”

11.43 sendmethod()

This function puts a method call into the event queue to be executed asynchronously from the event loop (DM_EventLoop). It is therefore a convenience function for sending an external event with **sendevent()** and calling the method in the event rule for that external event.

sendmethod() supports a maximum of *14* arguments for the method call and cannot be used for methods with `output` parameters.

Return values from methods cannot be processed.

Definition

```
void sendmethod
(
      object   Object input,
      method   Method input
  { , anyvalue Arg1   input
    ...
    , anyvalue Arg14  input }
)
```

Parameters

object *Object* input
Object whose method shall be invoked asynchronously.

method *Method* input
Identifier of the method to invoke.

*anyvalue **Arg1** input*
…
*anyvalue **Arg14** input*
Arguments that are passed to the method.

Availability

Since IDM version A.06.02.g

See also

Built-in function **sendevent()**

C function DM_SendMethod()

11.44 setinherit()

With this function you can reset the attributes of objects to the value of the relevant Models or Defaults.

You can find the attributes available for the relevant object type in the “Object Reference”.

Definition

```
void setinherit
(
      object    Obj  input,
      attribute Attr input
  { , integer IntIdx input | index IdxIdx input }
  { , boolean SendEvent := true input }
)
```

Parameters

object *Obj* input
This first parameter describes the object whose attribute you want to reset.

attribute *Attr* input
This second parameter describes the attribute you want to reset on the object.

*integer **IntIdx** input*
*index **IdxIdx** input*
This optional parameter may be either an integer value or an index value. It has to be specified, if a vectorial attribute is to be reset to the value stored in the model. If a one-dimensional attribute is to be reset, then a number has to be specified. If a two-dimensional attribute is to be reset, an index has to be specified.

*boolean **SendEvent** := true input*
By means of this optional parameter you can control whether the IDM is to trigger the rule processing by successful resetting of the attribute or not. The parameter has to be specified as *false*, if no events are to be sent. If events are to be sent, the parameter has to be specified as *true* or it has to be omitted.

Examples

» Resetting the contents of field *1,1* in a tablefield without triggering rules

```
setinherit (Tablefield, .content, [1,1], false);
```

» Resetting the contents of an input field along with triggering rules

```
setinherit(Edittext, .content);
```

11.45 setvalue()

With this function, you can change attributes of objects.

You can find the attributes available for the relevant object type in the “Object Reference”.

Using **setvalue()** you can change these values without triggering a *changed* event. This is useful if within a rule a lot of values are to be changed at once with loop constructs. Otherwise you take the risk to overflow the internal setvalue event queue.

Definition

```
boolean setvalue
(
      object    Obj  input,
      attribute Attr input,
      anyvalue  NewVal input
  { , integer   IntIdx input | index IdxIdx input }
  { , boolean   SendEvent := true input }
)
```

Parameters

object *Obj* input
This parameter describes the object whose attribute you want to use.

attribute *Attr* input
This parameter describes the object attribute which you want to change.

anyvalue *NewVal* input
In this parameter the value to be adopted by the attribute is specified.

integer ***IntIdx*** *input*
index ***IdxIdx*** *input*
This optional parameter may be either an integer or an index value. It has to be set if a vectorial attribute has to be specified. If a one-dimensional attribute is to be set, a figure has to be specified. If a two-dimensional attribute is to be set, an index has to be specified.

boolean ***SendEvent*** *:= true input*
By means of this optional parameter you can control whether the ISA Dialog Manager is to trigger the rule execution by the successful setting of the attribute. The parameter has to be specified as *false*, if no events are to be sent. If events are to be sent, the parameter has to be specified as *true* or has to be omitted.

Return value

true
Setting the attribute has been carried out successfully.
false
The attribute could not be set.

Examples

» Setting field 1,1 in a table

```
setvalue(Tablefield, .content, true, [1,1]);
```

» With this function you can also change attributes via variables, e.g.

```
rule void Set(attribute A)
{
  setvalue(Win1, A, true);
  Pb1.sensitive := getvalue(Win2, A);
}
```

See also

Method :set()

C function DM_SetValue in manual “C Interface - Functions”

11.46 setvector()

This function may be used to modify vector attributes (predefined or user-defined). It is possible to assign new values to the attribute as a whole or to only a continuous portion of it.

Definition

```
boolean setvector
(
      object    Object     input,
      attribute Attribute  input,
      vector    NewValue   input
  { , anyvalue  FirstIndex input
  { , anyvalue  LastIndex  input } }
  { , boolean   SendEvent := true input }
)
```

Parameters

object *Object* input

This parameter defines the object whose attribute values shall be modified.

attribute *Attribute* input

This parameter defines the attribute to be set.

vector *Value* input

This parameter specifies the value list that will be assigned to the vector attribute.

anyvalue ***FirstIndex*** *input*

This optional parameter defines the start index as of which the element values are modified. For one-dimensional array attributes an *integer* value should be specified, for two-dimensional arrays an *index* value. The default value for one-dimensional array attributes is the *integer* value *1*, for two-dimensional arrays the *index* value *[1,1]*.

anyvalue ***LastIndex*** *input*

This optional parameter specifies the end index up to which the values of the attribute are are modified through the values from the value list. Again, an *integer* value is expected for one-dimensional array attributes and an *index* value for two-dimensional arrays. The *LastIndex* value should be after the *FirstIndex* value.

If no *LastIndex* parameter is specified (default value *nothing*), the size of the *Value* vector defines the number up to which the vector attribute is modified. The subsequent element values of the attribute are then truncated.

boolean ***SendEvent*** *:= true input*

This optional parameter controls whether setting the attribute should trigger an *attribute-changed* event. The default value is *true*, which causes the *changed* event to be sent for the attribute. If the parameter value is *false*, no event is sent.

Return value

true
: Attribute has been modified.

false
: Modification of the attribute could not be completed.

Fault behavior

The function call fails for an invalid object or attribute, for an invalid index range or if setting is not possible e.g. due to a mismatch between attribute type and value type.

Example

```
dialog D

window Wi
{
  .title "Cities";
  .width 300;

  tablefield TfCities
  {
    .xauto 0;  .yauto 0;
    .rowheight[0] 25;  .colwidth[0] 80;
    .rowcount 1;  .colcount 2;
    .rowheader 1;
    .direction 2;
  }

  on close { exit(); }
}

on dialog start
{
  variable hash Capitals := [
    "germany" => "berlin",
    "france"  => "paris",
    "england" => "london" ];

  // fill the title row
  setvector(TfCities, .content, ["Country", "City"]);
  // fill the country and city column
  setvector(TfCities, .field, keys(Capitals), [1,1],
            [itemcount(Capitals),1]);
  setvector(TfCities, .field, values(Capitals), [1,2]);
}
```

See also

Built-in function **getvector()**

C functions DM_GetVectorValue(), DM_SetVectorValue()

11.47 sort()

This function returns a sorted copy of a given list value. Only the values are sorted, not the indexes. Therefore, a sorting of values is also possible for associative arrays (*hash* data type).

Sorting is carried out grouped by the data types of the value elements, ascending by the ordinal of the data types and values. When sorting texts or strings, comparison is always done using the *string* data type, which is more suitable for sorting.

Sorting can be controlled like this:

» With the *Reverse* parameter the sort sequence can be inverted.

» Through the *SortType* parameter the sort sequence of strings and texts can be influenced.

Definition

```
anyvalue sort
(
      anyvalue ListValue input
  { , boolean  Reverse  := false       input }
  { , enum     SortType := sort_binary input }
)
```

Parameters

anyvalue *ListValue* input

This parameter specifies the list value to be sorted.

boolean ***Reverse*** *:= false input*

If this optional parameter is set to *true*, sorting is carried out in descending order. The default value is *false*, which corresponds to an ascending sort.

enum ***SortType*** *:= sort_binary input*

This optional parameter only applies to item values that have the data type *string* or are temporarily converted to a string for sorting.

Value range

sort_binary (default)
Sorting based on the Unicode character code.

sort_linguistic
Sorting based on the language and region settings (locale) of the system.
Depending on the system settings and the set code page, upper and lower case as well as umlauts are taken into account.
Since this sort requires code page conversions, it is slower than *sort_binary*.
For Unix or Linux we recommend using the UTF8 locale.

See also

Documentation of locale, LC_CTYPE, LC_COLLATE, strcoll for your operating system.

Return value

Sorted collection of the same data type as the passed collection.

Fault behavior

The function call fails if the *ListValue* parameter is not a collection or another parameter has an invalid value.

Example

```
// UTF8
dialog D

on dialog start
{
  variable list List := [ "Äcker", "Bande", "Bäcker", "binden",
                          "Bund", "Aachen", "an" ];
  variable string S;

  foreach S in sort(List, sort_linguistic) do
    print S;
  endfor
  exit();
}

/* returns: Aachen Äcker an Bäcker Bande binden Bund */
```

See also

Built-in functions **find()**, **keys()**, **values()**

11.48 split()

With the **split()** function, a string can be split at delimiters into a list of substrings. If no delimiters are specified, the string is separated into single characters.

Definition

```
list split
(
  string Separators input,
  string String     input
)
```

Parameters

string *Separators* input

This parameter specifies the delimiters where the *String* parameter is split. The delimiters will not be contained in the result list.

If *Separators* contains a string, each character from that string is used as a delimiter. If in *String* two delimiters follow each other directly, an empty string will be included in the result list at this position.

If *Separators* is an empty string, the *String* parameter will be split into a list of single characters.

string *String* input

In this parameter, the string to be split is passed.

Return value

A list of substrings.

Examples

» Separation into single characters

```
print split("", "Roy");
```

Output

```
["R","o","y"]
```

» Separation at different delimiters

```
print split(",.;", "1,23,4.5;;6,478-9");
```

Output

```
["1","23","4","5","","6","478-9"]
```

11.49 sprintf()

This function is used to format a string according to a given format string. It is similar to the functions available in the C library.

Definition

```
string sprintf
(
      string   Format     input,
      anyvalue Argument1  input
  { , anyvalue Argument2  input }
      ...
  { , anyvalue Argument15 input }
)
```

Parameters

string *Format* input

Format describes the formatting of the output string. The individual formats follow, to a large extent, the notation in ANSI-C. However, some restrictions are necessary. The IDM is not able to check whether the format string is correct. Incorrect specifications in the arguments for the function cause errors.
The data types of the ISA Dialog Managers will be output exclusively via “%s” (plus possible formats). The data types *integer* and *pointer* are exceptions; they are output via numerical format types. The possible values for the ISA Dialog Manager are described below in the section “Syntax of the Format String”.

anyvalue *Argument1* input

anyvalue ***Argument2*** *input*

…

anyvalue ***Argument15*** *input*

These (optional) parameters have to be specified according to the format string.
They are used for the corresponding format characters in the output string.

Return value

The return value is a string corresponding to the input format *Format*. In case of errors an empty string will be returned.

Syntax of the Format String

The first character in a format is “%”. The following term describes the syntax for a format of the IDM **sprintf()**:

```
%[argument$] [flags] [width] [.precision] type
%[argument$] [flags] [width] [.precision] ([singular],[plural])
```

The individual elements of the term are defined as follows:

argument

This parameter controls the order of the given parameters. The given arguments usually keep their order. However it is also possible to use any argument (i.e. not following the given order), provided that the relevant argument is accessed via its position (number). It is thus possible to use parameters several times.

Example

```
text DateFormat "Date"
{
  0: "%02d.%02d.%04d";        // German
  1: "%2$02d/%1$02d/%3$04d";  // British
}
...
sprintf(DateFormat,7,9,1965);
// variant 0:"07.09.1965"
// variant 1:"09/07/1965"
```

flags

IDM supports the following flags: +, -, *0*, #

The flags + and - define whether the string is to be formatted left-aligned (-) or right-aligned (+), i.e. whether necessary blanks are to be inserted at the beginning or at the end of the string.

It is also possible to specify *0* for the output of numbers. The strings will thus be filled with zeros. If # is specified, the hexadecimal number format will be *0x* or *0X*.

width

This parameter controls the minimal number of characters to be output. If less characters than specified are output, blanks will be inserted at the beginning or at the end of the string. This is controlled by the “%s” parameter. If the string is longer than the specified field length all characters will be output.

This parameter may also contain a star (*). This means that the following argument in the argument list will be used to define the output size.

Examples

`sprintf("%0*x",4,255)` is interpreted as `sprintf("%04x",255)`

`sprintf("%s %*s","Hallo",40,"Welt")` is interpreted as sprintf("%s %40s","Hallo","Welt")

.precision

This parameter controls the length of a string to be output. The first character is always a dot “.” and will lead, in contrast to the parameter *width*, to a clipping of the string.

Examples

```
sprintf("%.5s", "123456789")  // Result "12345"
sprintf("%.*s", 2, "1234")    // Result "12"
```

type

This parameter defines the type of the relevant argument. The following types are possible in the IDM:

Type	Output Format
d	number with prefix
u	number without prefix
b	number in binary format
o	number in octal format
x	number in hexadecimal format, using “abcdef”
X	number in hexadecimal format, using “ABCDEF”
c	output of a character
s	output of a string

For the use in IDM the formats for strings (*%s*) and numbers (*%d*, *%u*, *%x*, *%o*) have been expanded. The behavior is described in the following table. *[numeric]* is used instead of *d*, *x*, *X*, *b* or *u*.

Data Type	Control via %	Result
void	%s %[numeric]	empty string ERROR
string	%s otherwise	input string ERROR
text	%s %[numeric	relevant string textid in number format
object	%s %[numeric	name of object DM_ID in number format
pointer	%s %[numeric	ERROR address of pointers in number format

Data Type	Control via %	Result
event	%s %[numeric	name of event number of event in number format
enum	%s %[numeric	name of enumeration number of enumeration in number format
attribute	%s %[numeric	%name of attribute number of attribute in number format
method	%s %[numeric	name of method number of method in number format
integer	%s %[numeric	ERROR number in number format
index	%s %[numeric	both values of index ERROR
boolean	%s %[numeric	output of *true* or *false* as string output of *0* (*false*) or *1* in number format
class	%s %[numeric	name of class internal number of class in number format
datatype	%s %[numeric	data type as string data type in number format

Pluralism

This parameter is used to control an output that depends on the value of the argument. You can select the given singular value or plural value. The values must always be indicated in square brackets, separated by "|". The given value for the relevant part of the string in the parameter range determines the value to be displayed.

The following is valid:

1
: singular value

<> *1*
: plural value

Example

```
sprintf("%d %[house|houses]", 1);  // "1 house"
sprintf("%d %[house|houses]", 2);  // "2 houses"
sprintf("%d %[house|houses]", 0);  // "0 houses"
```

Summary Examples

```
!! output of a hexadecimal number in a statictext.
dialog HexOut

window W
{
  child statictext HexOutput { }
}

on dialog start
{
  variable integer Answer := 42;
  variable string S;
  S := sprintf("This is a hex number: %X", Answer);
  W.HexOutput.text := S;
}
```

```
!! Output of a date in a statictext.
!!  Variants are used to support country-specific features
!! (order of day, month and year).
dialog VariantDate
text DateFormat "DateFormat"
{
  0: "%02d.%02d.%04d";
  1: "%2$02d/%1$02d/3$04d";
}

window W
{
  child statictext Date{ }
}

on dialog start
{
  variable integer Day := 7;
  variable integer Month := 9;
  variable integer Year := 1965;
  variable string S;
  S := sprintf(DateFormat,Day,Month,Year);
  !! language variant 0: S = "07.09.1965"
  !! language variant 1: S = "09/07/1965"
  W.Date.text := S;
}
```

```
!! Example for the use of singular and plural values.
dialog SingularPlural
```

```
on dialog start
{
  sprintf("%d %[house|houses]", 1);  // "1 house"
  sprintf("%d %[house|houses]", 2);  // "2 houses"
  sprintf("%d %[house|houses]", 0);  // "0 houses"

  print sprintf("%3d file%[|s] copied", 1);
  // Result:  1 file copied
  print sprintf("%3d file%[|s] copied", 34);
  // Result: 34 files copied
  print sprintf("%3d file%[|s] copied", 0);
  // Result:  0 files copied
}
```

```
!! This dialog shows specific features of sprintf() in the IDM.
dialog DMInterna

window W1 { }

on dialog start
{
  print sprintf("Attribute %s has number %1$d", W1.visible);
  // Result: Attribute .visible has number 1

  print sprintf("Window %s has DM_ID %d", W1, W1);
  // Result: Window W1 has DM_ID 5

  print sprintf("Index %s", [1,2]);
  // Result: Index [1,2]
}
```

11.50 stop()

With this function you can stop a dialog. If this has been the last running dialog, the event loop will be exited. If no dialog is specified, the current dialog will be terminated. If only one single dialog is executed, **stop()** will have the same effect as **exit()**.

Definition

```
void stop
(
      object  Dialog  input
  { , boolean Destroy := false input }
)
```

Parameters

object *Dialog* input

In this parameter the dialog to be stopped has to be specified.

*boolean **Destroy** := false input*

If this optional parameter has the value *true*, the dialog will be deleted after the execution of the *finish* rules.

See also

Built-in function **exit()**

C function DM_StopDialog in manual “C Interface - Functions”

11.51 strcmp()

With this function, you can check whether two strings are identical. To do so, you can control via the parameters how many characters are to be compared at most and if upper/lower cases are to be distinguished or not.

Definition

```
integer strcmp
(
      string  String1 input,
      string  String2 input
  { , integer Length  input }
  { , boolean IgnoreCase := false input }
)
```

Parameters

string *String1* input
In this parameter the first of the strings to be compared is specified.

string *String2* input
In this parameter the second of the strings to be compared is specified.

*integer **Length** input*
If this optional parameter is specified, you can define how many characters are to be compared at most in both strings.

*boolean **IgnoreCase** := false input*
If this optional parameter is specified by the value *true*, no distinction will be made between upper case and lower case letters.

Rückgabewert

The function result can have the following values (always considered under the specified conditions like the length of the comparison or the distinction between upper and lower case letters):

-1
String1 lexically comes before *String2*
0
both strings are identical
1
String1 lexically comes after *String2*

Examples

» Comparing two strings up to length *4*

```
strcmp ("ABCDefgh", "ABCDfghijk", 4);
// Result: 0
```

» Comparing without considering upper case and lower case

```
strcmp("ABCDEF", "abcdef", true);
// Result: 0
```

11.52 stringpos()

This function is used to check if a string also occurs in another string.

Definition

```
integer stringpos
(
      string  String   input,
      string  Pattern  input
  { , integer StartIdx := 1 input }
)
```

Parameters

string *String* input

In this parameter that string is specified in which a given pattern is to be searched.

string *Pattern* input

In this parameter the searched string will be specified.

*integer **StartIdx** := 1 input*

In this optional parameter you can specify from where in the given *String* the *Pattern* is to be searched for. If this parameter is missing, the *Pattern* will be searched from the beginning of the *String*.

Return value

0

The searched string does not occur in this string.

> 0

The searched string does occur in this string and the start index will be returned as result. If the searched string does occur several times, the position of the first occurrence will be returned.
If the searched string is an empty string, the return value will be *1*.

Example

In an input field the pattern “EUR” is to be searched.

```
on Edittext charinput
{
  variable integer Idx;

  Idx := stringpos(this.content, "EUR");
  if Idx > 0 then
    print "Found";
  endif
}
```

11.53 strreplace()

The **strreplace()** function replaces substrings in a string with one or more substitute strings. The parts to be replaced may be defined either as a position with an optional length indication, a string, or a list of strings.

Definition

In the first form, the *Index* parameter specifies the position of the substring that shall be replaced by the substitute string *Replace* within *String*. The optional *Length* parameter defines how many characters will be replaced.

```
string strreplace
(
      string  String  input,
      integer Index   input,
      string  Replace input
  { , integer Length  input }
)
```

In the second form, each occurrence of *Match* in *String* is replaced by the substitute string *Replace*. The optional *IgnoreCase* parameter determines whether the comparison is case-sensitive.

```
string strreplace
(
      string  String  input,
      string  Match   input,
      string  Replace input
  { , boolean IgnoreCase := false input }
)
```

In the third form of **strreplace()**, in *String* all substrings from the *MatchList* collection are replaced by the single substitute string in *ReplaceListOrString* or the corresponding substitute string from the list *ReplaceListOrString*. The optional *IgnoreCase* parameter determines whether the comparison is case-sensitive.

```
string strreplace
(
      string   String    input,
      anyvalue MatchList input,
      anyvalue ReplaceListOrString input
  { , boolean  IgnoreCase := false input }
)
```

Parameters

string *String* input

In this parameter, the string is passed where substrings shall be replaced.

integer *Index* input

This parameter determines the position where the substring to be replaced begins within *String*. If *Index* is greater than the length of *String*, the substitute string is appended to *String*. *Index* < *1* is treated like *Index* = *1*.

string *Match* input

In this parameter, the substring to be replaced within *String* is passed. If *Match* is an empty string, there will be no replacement.

anyvalue *MatchList* input

In this parameter, a list of substrings is passed that are to be replaced within *String*. *MatchList* may be any collection, but all its values must have the data type *string* or ***text***.

string *Replace* input

In this parameter, the substitute string is passed which will replace the substrings in *String*. The substrings will be replaced in the order of their occurrence within *String*. There will be no recursive substitution.

anyvalue *ReplaceListOrString* input

This parameter contains a single substitute string or a list of substitute strings. The list may be any collection, but all its values must have the data type *string* or ***text*** and its length must be equal to the length of *MatchList*.

If *ReplaceListOrString* contains a collection, each occurrence of the substring *MatchList[I]* is replaced by the substitute string *ReplaceListOrString[I]*. The order of substitution is determined by the order of the substrings in *MatchList*, replacing the individual substrings in the order they occur within *String*.

There will be no recursive substitution, i.e. each part of *String* will be replaced at most once.

integer ***Length*** *input*

This optional parameter defines the number of characters to be replaced by the substitute string. *Length* can only be used together with *Index*. If *Length* is not specified, *Replace* will substitute the substring beginning at *Index* to the end of *String*. For *Length* <= *0*, the substitute string is inserted without replacing characters.

boolean ***IgnoreCase*** *:= false input*

This optional parameter controls whether the comparison of substrings is case-sensitive.

IgnoreCase	Meaning
false (default value)	The comparison of substrings is case-sensitive; upper and lower case have to be identical.
true	Upper and lower ase are ignored when comparing substrings and do not need to be identical.

Return value

A string in which substrings have been substituted by replacement strings.

Examples

» Multiple replacement of the letter *"l"* by the letter *"L"*

```
print strreplace("Hello World", "l", "L");
```

Output

```
"HeLLo WorLd"
```

» Replacement of multiple substrings

```
print strreplace("Anna", ["An", "n"], ["M", "i"]);
```

Output

```
"Mia"
```

» Replacement of multiple substrings
Each part of the string *"abc"* is replaced at most once, i.e. *"a"* by *"b"*, *"b"* by *"c"* and *"c"* by *"x"*. There is **no** recursive replacement of *"a"* by *"b"* by *"c"* by *"x"*.

```
print strreplace("abc", ["a", "b", "c"], ["b", "c", "x"]);
```

Output

```
"bcx"
```

» Replacement of multiple substrings by the character *"."*, case insensitive

```
print strreplace("Abracadabra Ricinus", ["A", "r", "ICI", "u"], ".",
true);
```

Output

```
".b..c.d.b.. ..n.s"
```

» Replacing a part of a string with another string

```
print strreplace("Bamjok", 3, "ngk", 2);
```

Output

```
"Bangkok"
```

» Insert at the beginning of a string

```
print strreplace("one", 0, "zero ", 0);
```

Output

```
"zero one"
```

» Append to the end of a string

```
print strreplace("one", 4, " two");
```

Output

```
"one two"
```

» Append outside the string length

```
print strreplace("one", 100, "...");
```

Output

```
"one..."
```

11.54 substring()

Parts of strings can be extracted with this function.

Definition

```
string substring
(
      string  String input,
      integer Index  input
  { , integer Length input }
)
```

Parameters

string *String* input

In this parameter the string from which a string part is to be copied is specified.

integer *Index* input

In this parameter the position in the string from which the string part is to be copied is indicated.

Note

This parameter has to be smaller than the length of parameter *string* and greater than *0* – otherwise, the function is not carried out.

*integer **Length** input*

By means of this optional parameter you can control the maximum of characters to be copied from the string. If this parameter is not specified, everything from the indicated position on to the end of the string will be copied.

Return value

""

The empty string will be returned if the specifications are incorrect.

copied string

The copied string will be returned as a result.

Examples

» Copy from position 4 of a string:

```
print substring("123456789", 4);
// Output: "456789"
```

» Copy from position 4 of a string, with a length of 3:

```
print substring("12345", 4, 3);
// Output: "45"
```

11.55 tolower()

With this function a string can be turned completely into lower case letters.

This function is based on the functions of the relevant operation system. It may happen, for example, that an **umlaut** is not treated correctly.

Definition

```
string tolower
(
  string String input
)
```

Parameters

string *String* input

In this parameter the string is passed in which all letters are to be turned into lower case letters.

Return value

The result of this function is a string in which all contained letters are lower case letters. The other characters remain unchanged.

Example

```
tolower("AbCdEf");
// Result: "abcdef"
```

11.56 toupper()

With this function a string can be transformed completely into upper-case letters.

This function is based on the functions of the relevant operation system. This is why it can happen that an **umlaut** cannot be transformed correctly.

Definition

```
string toupper
(
  string String input
)
```

Parameters

string *String* input
: In this parameter the string to be turned into capitals is specified.

Return value

The result of this function is a string in which all contained letters are capitals. The other characters remain unchanged.

Example

```
toupper("AbCdef1");
// Result: "ABCDEF1"
```

11.57 trace()

This function can be used for the output of values in a tracefile during the testing process of the dialog development.

Note

On calling this function the braces may be omitted.

Definition

```
void trace
(
  anyvalue TraceVal input
)
```

Parameters

anyvalue *TraceVal* input

In this parameter the value to be output is indicated.

Example

Output of the current object in a rule:

```
trace this;
```

11.58 trimstr()

With this function you can delete blanks at the beginning and/or end of a string.

Definition

```
string trimstr
(
  string  String input,
  boolean Start  input,
  boolean End    input
)
```

Parameters

string *String* input

In this parameter that string is indicated in which the blanks are to be deleted at the beginning and/or end.

boolean *Start* input

If this parameter is specified as *true*, all leading blanks will be deleted; otherwise, the leading blanks remain.

boolean *End* input

If this parameter is specified as *true*, all blanks at the end of the string will be deleted; otherwise, the blanks at the string end remain.

Return value

As a result of this function a string will be returned in which all specified blanks have been deleted.

Examples

Deleting blanks in a string:

```
» trimstr("      123     ", true, true);
// Result: "123"
```

```
» trimstr("      123     ", false, true);
// Result: "      123"
```

11.59 typeof()

By means of this function you can query or specify the data type of any expression.

Definition

```
datatype typeof
(
  anyvalue ReqVal input
)
```

Parameters

anyvalue ReqVal input

In this parameter the value whose data type is to be specified is indicated.

Return value

As return values of this function all data types defined by the ISA Dialog Manager can occur.

Examples

» Data type of the attribute *.xleft*

```
typeof(this.xleft);
// Result: integer
```

» Data type of the attribute *.bgc*

```
typeof(this.bgc);
// Result: color
```

11.60 updatescreen()

With this function you can display all internal SetVal events on the screen.

This function ensures that all internal changes to the attributes are displayed on the screen.

Definition

```
void updatescreen
(
)
```

Example

After several allocations to a listbox a function is to be called. First, however, the values are to become visible.

```
variable integer I;

for I := 1 to 1000 do
  Listbox.content[I] := itoa(I);
endfor
updatescreen();
SomeFunction();
```

11.61 valueat()

This function returns the indexed value in a collection at the specified position. The allowed positions are *1 … itemcount()* and thus allow looping through all indexed values without knowing the actual index.

The function returns an error (`fail`) if the position is outside the allowed range or if the *Value* parameter is not a collection.

Definition

```
anyvalue valueat
(
  anyvalue Value input,
  integer  Pos   input
)
```

Parameters

anyvalue *Value* input
This parameter specifies the value list from which the indexed value shall be determined.

integer *Pos* input
Position for which the indexed value shall be determined.

Return value

Value found in the collection at the position passed as a parameter.

Example

```
dialog D

on dialog start
{
  variable matrix Matrix := [
    [0,0] => "-?-",
    [1,1] => "germany",
    [1,2] => "berlin",
    [2,1] => "france" ];
  variable integer Pos;

  /* print the Matrix elements [1,1] [1,2] ... [2,2] */
  for Pos:=1 to itemcount(Matrix) do
    print sprintf("%s : %s", indexat(Matrix, Pos), valueat(Matrix, Pos));
  endfor
  exit();
}
```

See also

Built-in functions **indexat()**, **values()**

Method index

C functions DM_ValueGet(), DM_ValueIndex()

11.62 values()

This function returns a list of all values in a collection. Default values are not included. Applied to a scalar value, this is returned as a list.

Definition

```
anyvalue values
(
  anyvalue Value input
)
```

Parameters

anyvalue *Value* input
This parameter specifies the value on which the function is applied.

Return value

A list containing all values of a collection without the default values.

For a scalar parameter, a list containing only this parameter is returned.

Example

```
dialog D

on dialog start
{
  variable hash DomainHash := [
    ".de" => "germany",
    ".us" => "usa",
    ".fr" => "france",
    ".uk" => "united kingdom" ];
  variable string Country;

  /* print all country names from the DomainHash */
  foreach Country in values(DomainHash) do
    print Country;
  endfor
  exit();
}
```

Output

```
"germany"
"france"
"united kingdom"
"usa"
```

See also

Built-in functions **keys()**, **valueat()**

Method index

C function DM_ValueCount()

12 Formal Syntax of Rules and Statements

12.1 Operators

Operators can be used to formulate expressions.

- Arithmetical operators serve for the calculation of numbers (arithmetical values).
- Comparative operators serve for linking arithmetical and logical expressions.
- Logical operators link two expressions to form a new expression.

The operators are described in the following.

Arithmetical Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulo operator

Comparative Operator	Meaning
=	equal
<>	not equal
>=	greater than or equal
<=	less than or equal
<	less than
>	greater than

Logical Operator	Meaning
and	logical AND
andthen	logical AND evaluated only as far as necessary to determine the result

Logical Operator	Meaning
or	logical OR
orelse	logical OR evaluated only as far as necessary to determine the result
not	logical negation

Miscellaneous Operator	Meaning
()	priority control, bracket term
:=	assignment
::=	assignment without triggering event
-	algebraic sign

Priority of Operators

```
( )                          high
not

*     /     %
+     -
<>    >=    <=    <     >
=

and   andthen
or    orelse
:=    ::=                    low
```

Example

```
variable  boolean  Result;
:
Result := Button_1.active = true and Number = 10;
```

The variable “Result” will get the value *true* if “Button_1.active” is *true* and if at the same time the variable “Number” gets the value *10*; otherwise it will get the value *false*.

12.2 Expression

"Expression" is the generic term for the names of constants, variables, identifiers, function calls, objects and object attributes.

Numbers and strings are also expressions.

Operators have to be put between expressions.

An expression can have the following elements:

- string
- variable
- function
- object
- integer value
- boolean value
- datatype
- object class
- event
- attribute
- enum(eration)
- index
- built-in function
- expression
- expression attribute
- expression attribute index
- expression operator expression
- negative expression
- negated boolean expression

These elements are described in the following.

String

A string can contain all letters of the Latin alphabet as well as most national special characters, e.g. mutated vowels. A string is usually an arbitrary name to be used in the rule and is indicated by quotation marks.

Example

```
"Cancel"
"ISA Dialog Manager"
```

Variable

If you want to include a variable, specifying the variable name is sufficient. (See chapter "Rules" -> "Variables".)

Function

If you want to include a function, you first have to state the function name and then the parameters. The parameters are given in the form of expressions. (See chapter "Rules" -> "Call of Application Functions")

Example

```
Add (a,b);
```

Object

An object is defined by its object name. The listbox with all existent dialog object names is located in the upper area of the rule editing window.

Example

```
OK_Button
MainWindow.Groupbox1.ExitButton
```

Integer Value

An integer value is an integer between -2^{31} an $+2^{31}$.

Example

```
30
```

Boolean Value

A boolean value can have the value *true* or *false*.

Example

```
true
false
```

Data Type

The following types are valid:

- anyvalue
- attribute
- boolean
- class
- datatype
- enum
- event
- index
- integer
- object
- string
- void

Object Class

The following values are valid:

- accelerator
- application
- canvas
- checkbox
- color
- cursor
- dialog
- edittext
- font
- function
- groupbox
- image
- import
- listbox
- menubox
- menuitem
- menusep
- messagebox
- module
- notebook
- notepage
- poptext
- pushbutton
- radiobutton
- rectangle
- rule
- scrollbar
- statictext
- tablefield
- text
- tile
- timer
- variable
- window

Event

The following values are valid:

- activate
- changed
- charinput
- close
- dbselect
- deactivate
- deiconify
- deselect
- deselect_enter
- extevent
- finish
- focus
- help
- hscroll
- iconify
- key
- modified
- move
- resize
- scroll
- select
- start
- vscroll

Attributes

The identifiers of the attributes described in the “Attribute Reference” are valid.

Example

```
.visible
```

Enumeration (enum)

The following values are valid:

- noicon, icon_hand, icon_question, icon_exclamation, icon_asterisk, icon_information, icon_query, icon_warning, icon_error

- nobutton, button_ok, button_cancel, button_retry, button_abort, button_ignore, button_yes, button_no
- sel_row, sel_column, sel_area, sel_header, sel_single
- store_never, store_onfree, store_onchange
- edit_online, edit_offline, edit_locking
- winsys_x11, winsys_windows, winsys_pm, winsys_none
- toolkit_motif, toolkit_isa, toolkit_windows, toolkit_pm, toolkit_alpha
- coltype_bw, coltype_grey, coltype_color
- os_dos, os_os2, os_unix, os_vms, os_mpe, os_nt

Index

Syntax

```
[ Integer-Expression , Integer-Expression ]
```

Example

```
[3, TF1.colheader]
```

Built-in Function

Built-in functions are functions known to the ISA Dialog Manager as standard. See chapter “Built-in Functions”.

Expression

An expression can in turn contain another expression.

Example

```
a+b
```

Expression Attribute

An expression can be linked to an attribute. The expression has to provide an object.

Example

```
P1.visible
```

Expression Attribute Index

An expression can be linked to an indexed attribute. The expression has to provide an object. One- and two-dimensional indices can be given.

Example

```
L1.content[17]
T1.active[3,7]
```

Expression Operator Expression

An expression can be linked to another expression with an arithmetic operator.

Example

```
B.x := B.x + B.4;
```

Negative Expression

A negative expression can e.g. be a negative integer.

Example

```
-7
```

Negated Expression

The expression can be preceded by the word NOT, so that a switch becomes possible.

Example

```
Window.visible := not Window.visible;
```

The exact syntax of the various components of an assignment is listed in the following in Backus-Naur mode.

Definition

```
<expression>::=
                <string>|
                <variable>|
                <function>|
                <object>|
                <integer value>|
                <boolean value>|
                <data type>|
                <object class>|
                <event>|
                <attribute>|
                <enum>|
                <index>|
                <built-in function>|
                (<expression>)|
```

	<expression><operator><expression>|
	-<expression>|
	not<expression>

<object>::=	
	<objectpath>{{<relation>}{<objectpath>}} [<attribute>]

<built-in function>::=	
	<built-in> (<expression>)

<data type>::=	
	anyvalue | attribute | boolean | class | datatype | enum | event | index | integer | object | string | void

<object class>::=	
	accelerator | application | ... (see above)

<event>::=	
	activate | changed | ... (see above)

<enum>::=	
	noicon | icon_hand | ... (see above)

<index>::=	
	[<expression>, <expression>]

<operator>::=		
	+	|

```
                -   |
                *   |
                /   |
                %   |
                <   |
                <=  |
                =   |
                <>  |
                >=  |
                >   |
                and |
                or
```

```
<objectpath>::=
                    <objectidentifier>{.<objectidentifier>} |
                    <variable>
```

```
<objectpath>::=
                    <objectpath> |
                    this
```

```
<relation>::=
                .parent |
                .window |
                .menubox |
                .groupbox
                |
```

```
<string>::=
                "<ASCII>"
```

```
<ASCII>::=
```

	{<digit>|<letter>|<space>|\<octal number>|<special character>}

12.3 Statements

In order to form statements, operators and expressions are used.

<statement>	::=	
<object>	:=	<expression>;|
<variable>	:=	<expression>;|
if(<expression>)then<statementlist> [else<statementlist>]		
endif|		
exit		;|
print<expression>		;|
<function>		;
<statementlist>		::=
<statement>{<statement>}		

Syntax (Conditional Program Run)

The syntax of a conditional program run is listed in the following.

If_then_else::=	
	<condition line>
	then
	<statement>
	else
	<statement>
	endif
	<condition line>
	then

	<statement>		
	endif		
statement::=			
	<object>	:=	<expression>; |
	<variable>	:=	<expression>; |
	if(<expression>)then<statementlist> [else<statementlist>]		
	endif |		
	exit; |		
	print<expression>; |		
	<function>;		

statementlist::=	
	<statement> {<statement>}

Examples for Statements

```
window.visible  := true;
window2.x       := window1.height+window1.x+10;
```

Metalingual examples for the use of statements in rule definitions:

```
on Identifier event
if ( statement with boolean expression )
{
  Statements;
}
on Identifier event
{
  ...
  if ( statement with boolean expression )
  then
    Statements;
  endif
  ...
}
on Identifier event
{
  ...
```

```
  if ( statement with boolean expression )
  then
    Statements;
  else
    Statements;
  endif
  ...
}
```

Note

Boolean expressions are formed with logical operators.

Function calls for the application can be used in two ways:

function	The function is **not** called; the last result of the function is used.
function(parameter)	The function is called, the result of this call is used. Parameters can be transferred to the function optionally.

These functions form the interface to your application.

Example

```
on Variable_A.value changed
{
  print variable_value;
}
on Mainwindow close
{
  exit;
}
on Pushbutton select
{
  AP_Function (this.text);
}
```

Index

-

- 75

%

% 75

*****

* 75

/

/ 75

+

+ 75

<

< 75

<= 75

<> 75

=

= 75

=> 45

>

> 75

>= 75

A

accelerator 14, 41

access restriction 66

action 11

activate 14, 16-19, 21-22

addition 207

after 31-32

algebraic 208

alias 52

and 76, 207

andthen 76, 207

anyvalue 43, 51, 61-62

append() 88

application 15, 28, 60

application finish 28

application function 11, 71

  call 87

application start 28

applyformat() 91

assignment 208

  without event 208

assignment operator 74

atoi() 92

attribute 41, 43, 61, 77, 209

  change 71

attribute value

  change 71

attributes 51, 212

B

Backus-Naur 214

beep() 93

beep_error 93

beep_note 93

beep_ok 93

beep_question 93

beep_warning 93

before 31-32

boolean 43, 51, 61

boolean value 209-210

braces 76

bracket 208

brackets

- () 77
- [] 76
- {} 76
- round 77
- square 76

built-in function 11, 71, 88, 209, 213

- append() 88
- applyformat() 91
- atoi() 92
- beep() 93
- closequery() 95
- concat() 97
- countof() 99
- create() 101
- delete() 104
- destroy() 107
- dumpstate() 109
- exchange() 113
- execute() 116
- exit() 120
- fail() 122
- find() 124
- first() 127
- getvalue() 128
- getvector() 130
- indexat() 132
- inherited() 134
- insert() 135
- itemcount() 139
- itoa() 141
- join() 142
- keys() 145
- length() 146
- load() 147
- loadprofile() 148
- max() 149
- min() 150
- parsepath() 151
- print() 154
- querybox() 155
- queryhelp() 157
- random() 158
- regex() 159
- run() 165
- save() 166
- saveprofile() 167
- second() 169
- sendevent() 170
- sendmethod() 172

setinherit() 173
setvalue() 174
setvector() 176
sort() 179
split() 181
sprintf() 182
stop() 188
strcmp() 189
stringpos() 191
strreplace() 192
substring() 196
tolower() 197
toupper() 198
trace() 199
trimstr() 200
typeof 42
typeof() 201
updatescreen() 202
valueat() 203
values() 205
button_abort 155
button_cancel 155
button_ignore 155
button_no 155
button_ok 155
button_retry 155
button_yes 156

C

call of application functions 87
callback function 50, 55, 58
simulation 59
canvas 15
function 50, 87
canvas function 55
canvasfunc 55
case statement 77, 79
changed 14, 27
charinput 14, 16, 18, 21
checkbox 15
child[i] 38
class 43, 51, 61
clipboard 23
close 14, 22
closequery() 95, 156
code page 52
collection dtat type 44
comment 72-73
concat() 97
config 62
configurable 62
configuration file 62
constant 63
contentfunc 50, 57
count 41
countof() 99
create() 101
cut 14, 21, 23

D

data function 50, 56
Data Model 50
data type 43, 50, 61, 210
collections 44

hash 44
list 44
matrix 44
refvec 44
vector 44
datafunc 56
datatype 43, 51, 61, 209
dbselect 14, 17, 19, 21
deactivate 14, 16-19, 21-22
deiconify 14, 22
delete() 104
deselect 14, 16, 18
deselect_enter 14, 16, 19
destroy() 107
dialog 16
finish 28
start 28
division 207
division (/) 75
DM_LoadProfile 62
DM_QueueExtEvent 29
DM_SendEvent 29
DMF_PCREBinding 164
Drag&Drop 23
dump_all 110
dump_error 110
dump_events 110
dump_full 110
dump_locked 110
dump_memory 110
dump_none 110
dump_process 110
dump_short 110
dump_slots 110
dump_stack 111
dump_usage 111
dump_uservisible 111
dump_visible 111
dumpstate() 109
duration 94

E

edittext 16
else 78
elseif 78
endcase 71
endfor 71
endif 71, 78
endwhile 71
enum 43, 51, 61, 209, 212
enumeration 212
equal (=) 75
error
catching 122
passing 122
event 11, 13, 15, 43, 51, 61, 209, 212
external 13, 29
help 27
internal 13, 27
key 26
system 13, 28
user 13
event line 11, 13, 71
event object \ 41

event type 13
event types 13
event[EV] 41
event[I] 41
event_code 41
eventcount 41
example
  meta-lingual 218
exchange() 113
execommand 117
execute() 116
exenormal 117
exeshell 117
exit 29
exit() 120
expression 208-209, 213, 217
  negated 209, 214
  negative 209, 214
  syntax (assignment) 214
expression attribute 209, 213
expression attribute index 209, 213
expression operator expression 209, 213
external event 11, 13, 29
extevent 14-22

F

fail() 92, 122
false 43
find() 124
finish 14-16, 18, 28
finish rule 28-29
first() 127
firstchild 38
firstmenu 38
focus 14-21
for loop 81
foreach loop 82
formal syntax 207
format
  function 50
  resource 57
format function 57
formatfunc 50, 57
forward referencing 40
frequency 94
function 50-51, 60, 77, 209
  alias 52
  call 219
  callback 50, 55
  canvas 50, 55
  code page 52
  format 50, 57
  reloading 50, 57
  rule 50-51
  simulation 58
  type 50

G

getvalue() 128
getvector() 130
global variable 61
greater (>) 75
greater or equal (>=) 75
groupbox 16, 38

H

hash 44
  syntax 45
help 14-22, 32
help event 11, 13, 27
hidewindow 117
hierarchical inheritance 27
hscroll 14, 17-18, 21-22

I

iconify 14, 22
identifier 3
if 12, 78
if-elseif-else 77-78
if-then-else 77-78
if-then-else-endif 71
if-then-endif 79
image 17
import 17
  use 60
import object 60
index 41, 43, 51, 61, 209, 213
indexat() 132
inheritance 26
  hierarchical 27
  special 26
inherited() 134
initialization 62
input 36, 52-53
input parameter 36
input value 52
insert() 135
integer 43, 51, 61
integer value 209-210
internal event 11, 13, 27
itemcount() 139
itoa() 141

J

join
  string 143
join() 142

K

key 14, 16-22, 32
key event 26
keyboard
  event 11, 13
keys() 145

L

lastchild 38
lastmenu 38
length() 146
less (<) 75
less or equal (<=) 75
list 44
  syntax 45
listbox 17
load() 147
loadprofile() 148
local 15
local variable 83

logical and 76
logical AND 207
logical negation 76, 208
logical operator 75, 207
logical or 76
logical OR 208
loop construct 81
loop run 81
loop start 81

M

match_begin 125
match_exact 125
match_first 125
match_substr 125
matrix 44
max() 149
maxwindow 117
menu[i] 38
menubox 17
menuitem 17
menuseparator 18
messagebox 18
method 43, 51, 61
min() 150
minus (-) 75
minwindow 117
modified 15-16, 19, 21
modue 60
modularization 60
module 18
  finish 28
  start 28
module functions 60
modulo (%) 75
modulo operator 207
move 15, 22
multiplication 207

N

named rule 35
negated boolean expression 209
negated expression 214
negative expression 209, 214
nobutton 156
not 76, 208
notebook 18
notepage 18
null object 40

O

object 15, 43, 51, 61, 209-210
  application 28
  callback function 87
  class 209-210
  class events 11
  referencing 37
  special
    this 37
  this 37
  user 13
on 13
open 15

operator 207, 217
- 207-208
% 207
( ) 208
* 207
/ 207
::= 74, 208
:= 74, 208
+ 207
< 207
<= 207
<> 207
= 207
> 207
>= 207
and 76, 207
andthen 76, 207
arithmetic 207
assignment 74
comparative 207
logical 75, 207
not 76, 208
or 76, 208
orelse 76, 208
priority 208
or 76, 208
orelse 76, 208
otherwise 80
output 36, 52-53
output parameter 36
output value 52

P

parameter 52
input 36
length 52
output 36
parent 38
parsepath() 151
paste 15, 22-23
PCRE library
linking 163
version 163
plus (+) 75
pointer 43, 51, 61
poptext 18
precondition 12
print() 154
priority
operators 208
program block 11-12, 71
program flow 77
programming language
C 50
COBOL 50
pushbutton 19

Q

querybox() 95, 155
queryhelp() 157
Queue 29

R

radiobutton 19
random() 158
rectangle 19
reference operator 45
refvec 44, 46
regex() 159
regex_count 161
regex_eval 161
regex_locate 161
regex_match 161
regex_unmatch 161
regex_vars 161
relations 38
reloading function 50, 57
resize 15, 22
return type 35
rule 11, 35
    base 11
run() 165

S

save() 166
saveprofile() 167
scroll 15, 17-18, 20-22
scrollbar 19
second() 169
select 15-22
sendevent() 170
sendmethod() 172
setinherit() 173
setup 20
setvalue() 174
setvector() 176
showinactive 117
showwindow 117
signal handler 29
simulation 58
    callback function 59
simulation rule 58
size 53
sort() 179
sort_binary 179
sort_linguistic 179
spinbox 20
split() 181
sprintf() 182
start 15-16, 18, 28
start rule 28
statement 71, 217-218
statementlist 218
static 85
static variable 85
    initialization 86
statictext 20
status information 109
statusbar 20
stop() 188
strcmp() 189
string 44, 51, 62, 209
stringpos() 191
strreplace() 192
sub-rule 71

subprogram 35
substring() 196
subtraction 207
switch 77, 79
syntax
  conditional program run 217
  formal 207
  hash 45
  list 45
system event 11, 13, 28

T

tablefield 21
then 78
this 37-39
thisevent 41
timer 21
times (*) 75
tolower() 197
tone 93
  duration 94
  frequency 94
  volume 93
toupper() 198
trace() 199
tracing 199
treeview 21
trimstr() 200
true 43
type 41, 53
type checking 65
typeof() 201

U

unequal (<>) 75
updatescreen() 202
use 60
user
  event 11
user-initiated event 13
user event 13

V

validity range 65
value 27, 41
value restriction 66
valueat() 203
values() 205
variable 27, 61, 71, 77, 81, 83, 209
  configurable 62
  constant 63
  declaration 61
  global 61
  initialization 62
  local 83
  static 85
vector 44
void 51, 62
volume 93
vscroll 15, 17-18, 21-22

W

while loop 83
window 22, 38

X

x 41

Y

y 42