FICO® Xpress Mosel Libraries

6.12

REFERENCE MANUAL

FICO® Xpress Optimization

FICO

©2001–2026 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

Patent(s): www.fico.com/en/patents

Xpress Mosel 6.12 (FICO® Xpress 9.9)

Deliverable Version: A

Last Revised: April 2026

Contents

Introduction 1

1 Mosel Run Time Library 2

1.1 General . . . 3
1.1.1 Initialization and termination . . . 3
XPRMinit . . . 4
XPRMgetlicerrmsg . . . 5
XPRMfinish, XPRMfree . . . 6
XPRMgetdefworkdir . . . 7
XPRMgetlibpath . . . 8
XPRMfreelibpath . . . 9
XPRMsetdefworkdir . . . 10
XPRMsetlocaledir . . . 11
XPRMsetrestrictions . . . 12
XPRMsetsdmax . . . 13
XPRMremovetmpdir . . . 14
XPRMgetlocaledir . . . 15
XPRMgetsdmax . . . 16
XPRMgetversion . . . 17
XPRMgetversions . . . 18
1.1.2 Model management . . . 19
XPRMloadmod, XPRMloadmodsec . . . 20
XPRMsetbimprefix . . . 22
XPRMsetdefstream . . . 23
XPRMresetmod . . . 24
XPRMrunmod . . . 25
XPRMaddparamstr . . . 26
XPRMisrunmod . . . 27
XPRMstoprunmod . . . 28
XPRMtermrunmod . . . 29
XPRMunloadmod . . . 30
XPRMgetannotations . . . 31
XPRMgetbimprefix . . . 32
XPRMgetmodprop . . . 33
XPRMgetnextdep . . . 34
XPRMgetnextmod . . . 35
XPRMgetnextmoddso . . . 36
XPRMfindmod . . . 37
1.2 Post processing interface . . . 38
XPRMdsotyptostr . . . 39
XPRMfindident . . . 40
XPRMfindattrdesc . . . 43
XPRMfindtypecode . . . 44
XPRMgetattr . . . 45

XPRMgetnextanident . . . 46
XPRMgetnextident . . . 47
XPRMgetnextparam . . . 48
XPRMgetnextpbcomp . . . 49
XPRMgetnextpkgparam . . . 50
XPRMgetnextproc . . . 51
XPRMgetnextreq . . . 52
XPRMgetnextattrdesc . . . 53
XPRMgetprocinfo . . . 54
XPRMgettypeprop . . . 55
XPRMcsrtoref . . . 58
1.2.1 Lists . . . 59
XPRMgetlistsize . . . 60
XPRMgetlisttype . . . 61
XPRMgetnextlistelt . . . 62
XPRMgetprevlistelt . . . 63
1.2.2 Sets . . . 64
XPRMgetsetsize . . . 65
XPRMgetsettype . . . 66
XPRMgetelsetval . . . 67
XPRMgetelsetndx . . . 68
XPRMgetfirstsetndx . . . 69
XPRMgetlastsetndx . . . 70
1.2.3 Arrays . . . 71
XPRMgetarrdim . . . 72
XPRMgetarrtype . . . 73
XPRMgetarrsize . . . 74
XPRMgetarrsets . . . 75
XPRMgetfirstarrentry . . . 76
XPRMgetlastarrentry . . . 77
XPRMgetnextarrentry . . . 78
XPRMgetfirstarrtruentry . . . 79
XPRMgetnextarrtruentry . . . 80
XPRMchkarrind . . . 81
XPRMcmpindices . . . 82
XPRMgetarrval . . . 83
1.2.4 Records . . . 84
XPRMgetnextfield . . . 85
XPRMgetfieldval . . . 86
1.2.5 Unions . . . 87
XPRMgetnextuncomptype . . . 88
XPRMgetuntype . . . 89
XPRMgetuntypeid . . . 90
XPRMgetunvalue . . . 91
1.2.6 Problems . . . 92
XPRMgetprobstat . . . 93
XPRMexportprob . . . 94
XPRMgetobjval . . . 95
XPRMgetvsol . . . 96
XPRMgetcsol . . . 97
XPRMgetrcost . . . 98
XPRMgetdual . . . 99
XPRMgetslack . . . 100
XPRMgetact . . . 101
XPRMgetvarnum . . . 102

XPRMgetctrnum . . . 103
XPRMselectprob . . . 104
1.2.7 Miscellaneous . . . 105
XPRMfreememblk . . . 106
XPRMdate2jdn . . . 107
XPRMjdn2date . . . 108
XPRMtime . . . 109
XPRMpathcheck . . . 110
XPRMcb_sendint . . . 111
XPRMcb_sendreal . . . 112
XPRMcb_sendstring . . . 113
XPRMcb_sendctrl . . . 114
1.3 Debugger interface . . . 115
XPRMdbg_runmod . . . 117
XPRMdbg_getnextlocal . . . 119
XPRMdbg_getlndx . . . 120
XPRMdbg_getnblndx . . . 121
XPRMdbg_getlocation . . . 122
XPRMdbg_findprocblklndx . . . 123
XPRMdbg_setbrkp . . . 124
XPRMdbg_clearbrkp . . . 125
XPRMdbg_setstacklev . . . 126
1.4 Handling of modules . . . 127
XPRMsetdsopath . . . 128
XPRMgetdsopath . . . 129
XPRMregstatdso . . . 130
XPRMautounloaddso . . . 131
XPRMfinddso . . . 132
XPRMflushdso . . . 133
XPRMgetdsoannotations . . . 134
XPRMgetdsoparam . . . 135
XPRMgetnextdso . . . 136
XPRMgetnextdsoconst . . . 137
XPRMgetnextdsodep . . . 138
XPRMgetnextdsotype . . . 139
XPRMgetnextdsoparam . . . 140
XPRMgetnextdsoproc . . . 141
XPRMgetdsoprop . . . 142
XPRMgetnextiodrv . . . 143
XPRMpreloaddso . . . 144
1.5 Using IO drivers for data exchange . . . 145
1.5.1 `sysfd` driver . . . 145
1.5.2 `cb` driver . . . 145
1.5.2.1 Handling of general streams . . . 145
1.5.2.2 Handling of initializations blocks . . . 146
1.5.3 `mem` driver . . . 147
1.5.4 `raw` driver . . . 148
1.5.5 `bin` driver . . . 149

2 Mosel Model Compiler Library 150

2.1 Compilation . . . 150
XPRMcompmod, XPRMcompmodsec . . . 151
XPRMexecmod . . . 153

Appendix 154

A Contacting FICO 154

FICO Customer Support 154
Documentation 154
FICO Learning 155
Sales and maintenance 155
About FICO 155

Index 156

Introduction

The Mosel libraries may be used to embed the Mosel environment in applications developed in a programming language such as C.

The functions provided enable the user to:

- compile source model files into binary model (BIM) files
- load and unload BIM files handling several models at a time
- execute models
- access the Mosel internal database through the Post Processing Interface
- manage the dynamic shared objects used by Mosel

Two libraries are provided. The first one, the Run Time Library, contains the functionality required to load and run models that are already compiled. The second one, the Model Compiler Library, is the Mosel compiler that can be used to produce binary model files from source model files. In general, only the first library is used in an application, the models being provided in their binary form (which can be obtained using the Mosel executable).

This document gives a description of all functions included in the two libraries. For more details about how to compile and link programs with the Mosel libraries, please refer to the examples in the distribution of this software.

CHAPTER 1

Mosel Run Time Library

The Mosel Run Time (`xprm_rt`) Library provides a set of functions that may be used to load models in the form of *BIM* files, execute them and access model objects.

Programs using this library must include the header file `xprm_rt.h` that defines the following types:

`XPRMmodel:`	reference to a model stored in core memory
`XPRMdsolib:`	reference to a dynamic shared object descriptor
`XPRMattrdesc:`	reference to an attribute descriptor
`XPRMmpvar:`	reference to a decision variable
`XPRMlinctr:`	reference to a linear constraint
`XPRMset:`	reference to a set
`XPRMlist:`	reference to a list
`XPRMarray:`	reference to an array
`XPRMproc:`	reference to a procedure or function

The following basic types are also defined for completeness:

`XPRMinteger:`	integer value (C type `int`)
`XPRMreal:`	real value (C type `double`)
`XPRMboolean:`	Boolean value (C type `int`: 0 = false, 1 = true)
`XPRMstring:`	text string value (C type `const char *`)

Note that all text strings handled by functions of this library are encoded in UTF-8. It is therefore required to convert text strings to alternate encodings when exchanging data with other libraries not working with UTF-8. In particular the C library supports either wide characters (`wchar_t` type) or the default system encoding (that depends on the localisation of the system). These encoding conversions can be achieved with the help of the XPRNLS library (please refer to the XPRNLS Reference Manual for further details).

1.1 General

1.1.1 Initialization and termination

Each program using the Mosel libraries must start with a call to `XPRMinit`. If a Mosel library is loaded and unloaded dynamically at run time, the termination function `XPRMfinish` must be called before unloading the library in order to release the resources Mosel is using.

`XPRMfinish, XPRMfree`	Finish Mosel.	p. 6
`XPRMfreelibpath`	Release the memory allocated by XPRMgetlibpath.	p. 9
`XPRMgetdefworkdir`	Get default current working directory.	p. 7
`XPRMgetlibpath`	Get the location of the Mosel runtime library.	p. 8
`XPRMgetlicerrmsg`	Get license error message.	p. 5
`XPRMgetlocaledir`	Get the location of the translated messages.	p. 15
`XPRMgetsdmax`	Get the maximum depth of a stack dump.	p. 16
`XPRMgetversion`	Get the version number of Mosel.	p. 17
`XPRMgetversions`	Get version numbers.	p. 18
`XPRMinit`	Initialize Mosel.	p. 4
`XPRMremovetmpdir`	Remove the Mosel temporary directory.	p. 14
`XPRMsetdefworkdir`	Set default current working directory.	p. 10
`XPRMsetlocaledir`	Set the location of the translated messages.	p. 11
`XPRMsetrestrictions`	Set execution restrictions.	p. 12
`XPRMsetsdmax`	Set the maximum depth of a stack dump.	p. 13

XPRMinit

Purpose

Initialize Mosel.

Synopsis

```
int XPRMinit(void);
```

Return value

0 if executed successfully, 32 if Mosel is running in “trial mode”, other values indicate a license error.

Further information

This function initializes Mosel. It needs to be called before any other function described in this document may be executed. In case of failure, the function `XPRMgetlicerrmsg` may be used to obtain further information.

Related topics

`XPRMfinish`, `XPRMgetlicerrmsg`.

XPRMgetlicerrmsg

Purpose

Get license error message.

Synopsis

```
int XPRMgetlicerrmsg(char *msg, int maxlen);
```

Arguments

`msg` Pointer to an area where the error message is stored

`maxlen` Size of `msg`

Return value

Error code.

Further information

This function returns the last license error message.

Related topics

`XPRMinit`.

XPRMfinish, XPRMfree

Purpose

Finish Mosel.

Synopsis

```
int XPRMfinish(void);
```

Return value

0 if executed successfully, a non-zero value otherwise.

Further information

This function finishes a Mosel session. It unloads all modules that have been loaded, deletes the Mosel temporary directory and completely frees the memory used by Mosel.

Related topics

`XPRMinit`.

XPRMgetdefworkdir

Purpose

Get default current working directory.

Synopsis

```
const char *XPRMgetdefworkdir(void);
```

Return value

The current working directory or `NULL`.

Further information

1. The returned value may be `NULL` indicating that the process wide current working directory is used for this parameter.
2. This information can also be obtained using `XPRMgetdsoparam` asking for the `"workdir"` parameter.

Related topics

`XPRMsetdefworkdir`.

XPRMgetlibpath

Purpose

Get the location of the Mosel runtime library.

Synopsis

```
const char *XPRMgetlibpath(void);
```

Return value

The directory where the Mosel runtime library is stored.

Further information

1. The returned string is statically allocated the first time the function is called: the routines `XPRMinit` (upon failure) and `XPRMfinish` will release this memory block. It is also possible to free the allocated memory explicitly by calling `XPRMfreelibpath`.

2. This function may be called before Mosel has been initialised with `XPRMinit`.

Related topics

`XPRMgetdsopath`.

XPRMfreelibpath

Purpose

Release the memory allocated by XPRMgetlibpath.

Synopsis

```
void XPRMfreelibpath(void);
```

Further information

This function is automatically called by `XPRMinit` (when it fails) and `XPRMfinish`.

Related topics

`XPRMsetdsopath`.

XPRMsetdefworkdir

Purpose

Set default current working directory.

Synopsis

```
int XPRMsetdefworkdir(const char *path);
```

Argument

`path` New working directory or `NULL`

Return value

0 if successful, 1 otherwise.

Further information

1. Except for absolute file names, path resolution is relative to the *current working directory* defined by a call to this routine. This parameter is also used as the initial working directory when starting the execution of a model.

2. If the provided path is `NULL` (the default value after library initialisation) or an empty string, the system uses the current working directory of the process as provided by the operating system. Using this routine has no effect on the process wide current working directory.

3. The provided path is expanded (*i.e.* it can be relative to the current directory) and tested according to the current restriction settings (see `XPRMsetrestrictions`). If the restriction `NoRead` is active, this routine always fails and the current working directory is automatically set to the Mosel temporary directory.

Related topics

`XPRMgetdefworkdir`.

XPRMsetlocaledir

Purpose

Set the location of the translated messages.

Synopsis

```
void XPRMsetlocaledir(const char *localedir);
```

Argument

`localedir` Path to the NLS directory

Further information

This function can be used to specify the location of the translated messages (native language support) if they are not stored in the usual place.

Related topics

`XPRMgetlocaledir`.

XPRMsetrestrictions

Purpose

Set execution restrictions.

Synopsis

```
int XPRMsetrestrictions(unsigned int restr);
```

Argument

`restr` Restrictions as a bit encoded integer. The following values can be combined:

`XPRM_RESTR_NOWRITE`	Disable write access
`XPRM_RESTR_NOREAD`	Disable read access (implies `XPRM_RESTR_NOWRITE`)
`XPRM_RESTR_NOEXEC`	Disable routines allowing to execute external commands
`XPRM_RESTR_WDONLY`	Restrict disk access to current working directory
`XPRM_RESTR_NOTMP`	Disable temporary directory
`XPRM_RESTR_NODB`	Disable database access

Return value

0 if successful, 1 otherwise.

Further information

1. This routine activates the *restricted mode* by setting the restrictions to be applied. Retrictions can be set only once just after the library has been initialised (using `XPRMinit`), the routine will fail if called at a later stage.

2. If `XPRM_RESTR_NOREAD` is used, the restriction `XPRM_RESTR_NOWRITE` is implicitly selected and access to the local disk is disabled except for the temporary directory which becomes the current working directory. To disable entirely disk access to the local system, `XPRM_RESTR_NOTMP` has also to be selected.

3. The current active restrictions can be obtained using `XPRMgetdsoparam` asking for parameter `"restrict"`.

Related topics

`XPRMsetdefworkdir`.

XPRMsetsdmax

Purpose

Set the maximum depth of a stack dump.

Synopsis

```
void XPRMsetsdmax(int sdmax);
```

Argument

`sdmax` maximum number of levels to report

Further information

After a runtime error Mosel may report the content of the call stack for debugging purposes. This function defines the maximum number of levels to display in such a case. Using a value smaller than 1 disables stack dumps (this is the default).

Related topics

`XPRMgetsdmax`.

XPRMremovetmpdir

Purpose

Remove the Mosel temporary directory.

Synopsis

```
int XPRMremovetmpdir(void);
```

Return value

0 if successful, 1 otherwise

Further information

1. This function can be called only when no model is being executed and if the default working directory is not a subdirectory of the temporary directory.
2. The Mosel temporary directory is also removed by `XPRMfinish`.

Related topics

`XPRMsetdefworkdir`.

XPRMgetlocaledir

Purpose

Get the location of the translated messages.

Synopsis

```
const char *XPRMgetlocaledir(void);
```

Return value

Path to the NLS directory

Related topics

`XPRMsetlocaledir`.

XPRMgetsdmax

Purpose

Get the maximum depth of a stack dump.

Synopsis

```
int XPRMgetsdmax(void);
```

Return value

Maximum number of levels reported in a stack dump

Related topics

`XPRMsetsdmax`.

XPRMgetversion

Purpose

Get the version number of Mosel.

Synopsis

```
const char *XPRMgetversion(void);
```

Return value

The version number of Mosel as a text string.

Further information

This function returns the version number of Mosel.

XPRMgetversions

Purpose

Get version numbers.

Synopsis

```
int XPRMgetversions(int whichone);
```

Argument

`whichone` Version number to return:

0	Version of Mosel
1	Version of BIM format
2	Version of Native Interface
3	Version of Xpress

Return value

The version number requested or 0 in case of error.

Further information

This function returns the version number of Mosel, the Native Interface or BIM file format in numerical form. For instance for the Mosel version 1.2.1, the returned value is 1002001.

1.1.2 Model management

The following functions are required to manipulate models loaded in core memory: loading, running or unloading a model, getting information. Several models may be loaded in a single session of Mosel and used alternatively: each function requires a model (type `XPRMmodel`) as parameter to designate on which of the loaded models the operation is to be performed. This object of type `XPRMmodel` is returned by the function `XPRMloadmod` when a model has been successfully read from a *bim* (= **bi**nary **m**odel) file[1].

`XPRMaddparamstr`	Add a parameter definition to a parameter string.	p. 26
`XPRMfindmod`	Find a model by its name or order number.	p. 37
`XPRMgetannotations`	Retrieve annotations of a model.	p. 31
`XPRMgetbimprefix`	Get the list of prefixes for loading of packages.	p. 32
`XPRMgetmodprop`	Get a property of a model.	p. 33
`XPRMgetnextdep`	Enumerate dependencies of a model or package.	p. 34
`XPRMgetnextmod`	Get the next model.	p. 35
`XPRMgetnextmoddso`	Get the next module loaded for a model.	p. 36
`XPRMisrunmod`	Check if a model is running.	p. 27
`XPRMloadmod, XPRMloadmodsec`	Load a Binary Model file.	p. 20
`XPRMresetmod`	Reset a model.	p. 24
`XPRMrunmod`	Run a model.	p. 25
`XPRMsetbimprefix`	Set the list of prefixes for loading of packages.	p. 22
`XPRMsetdefstream`	Set default input/output streams.	p. 23
`XPRMstoprunmod`	Stop a running model.	p. 28
`XPRMtermrunmod`	Terminate execution of a model.	p. 29
`XPRMunloadmod`	Unload a model.	p. 30

[1] BIM files are produced by the Mosel compiler either by using the command line interpreter or with the Model Compiler Library.

XPRMloadmod, XPRMloadmodsec

Purpose

Load a Binary Model file.

Synopsis

```
XPRMmodel XPRMloadmod(const char *bname, const char *intname);
XPRMmodel XPRMloadmodsec(const char *bname, const char *intname, const char
      *flags, const char *passfile, const char *privkey, const char *keys);
```

Arguments

`bname` Name of a binary model file

`intname` Internal name (may be `NULL`), use `"*"` for automatic name generation

`flags` Loading options (may be `NULL`, options may be separated by spaces or `'-'` symbols):

- `"c"` Check signature (if the file is signed)
- `"V"` If the file is signed, load it only if the signature is valid
- `"T"` Load only signed files with a valid signature
- `"F"` The argument `passfile` is a file name (not the password itself)
- `"l"` Do not load required packages
- `"h"` Load only header of the bim file

`passfile` Password or password file (for encrypted BIM files)

`privkey` Private key file (for encrypted BIM files)

`keys` File of public keys

Return value

Reference to the model that has been loaded or `NULL`.

Further information

1. This function returns the reference of a new model instance created from a binary model file. The second form of the function will be used to load encrypted and/or signed BIM files if additional information has to be provided. While loading a model from a file, Mosel also automatically opens any additional modules and loads packages that are required by this model. If an internal name is provided, it is used in place of the name stored in the BIM file. If a model already existing in core memory (*i.e.* with the same internal name) is loaded a second time, the first instance of this model is deleted and a reference to the newly created model is returned. If model name or provided internal name is `"*"`, a unique name is automatically generated using pattern `"model_#"` where `#` is a hexadecimal number. If the loaded model has no name (empty string) and no internal name is provided (`intname` is `NULL` or an empty string), string `"(none)"` is used as a default.

2. Both BIM files referring to models and packages can be loaded by this function but only a model can be executed with `XPRMrunmod`.

3. The argument `keys` is a list of public key files (*i.e.* each line of the file is a key file name): when a signed BIM file is loaded, its signature is checked with the keys listed in this file. If this argument is not specified, the signing key is searched in the default public keys directory located at `getparam("ssl_dir")+"/pubkeys"`.

4. When the options `"-l"` or `"-h"` are used only the requested file is loaded and the resulting model cannot be executed. These options are typically used to inspect the dependencies of a BIM file (see `XPRMgetnextdep`).

5. If the flags have the special value `"!CLONE!"` and the internal name is `"*"` and the file name consists in a sequence of digits and all other parameters are empty, this routine will interpret the file name as a model number (see `XPRMgetmodprop`, property `XPRM_PROP_ID`) and create a copy of the corresponding model (without using any bim file). The returned model reference can be used just like it had been loaded from a bim file but it will share with its reference model all constant information (like constant strings or the code segment). As a consequence, during a debugging session, setting a breakpoint in a model loaded this way also installs the same breakpoint in all other models coming from the same source (including the reference model).

Related topics

`XPRMrunmod`, `XPRMdbg_runmod`, `XPRMunloadmod`, `XPRMsetbimprefix`.

XPRMsetbimprefix

Purpose

Set the list of prefixes for loading of packages.

Synopsis

```
void XPRMsetbimprefix(const char *pfx);
```

Argument

`pfx` List of prefixes separated by the `||` sequence

Further information

1. This parameter is used to locate packages when compiling (`XPRMcompmod`) or loading a file (`XPRMloadmod`). The name of the file is constructed by using each element of the list as a prefix. For instance if `bimprefix` has the value `"tmp:||./"` and it is required to load package `"mypkg"`, the loader will try first `"tmp:mypkg.bim"`, and then `"./mypkg.bim"`. If none of the files created through this mechanism can be found the same procedure as for modules is used (see `XPRMsetdsopath`).
2. The default value for this parameter is taken from the environment variable `MOSEL_BIM` when Mosel is initialised.

Related topics

`XPRMgetbimprefix`.

XPRMsetdefstream

Purpose

Set default input/output streams.

Synopsis

```
int XPRMsetdefstream(XPRMmodel model, int wmd, const char *filename);
```

Arguments

`model`	Reference to a model or `NULL`
`wmd`	Stream to set. Possible values: `XPRM_F_READ` Default input stream `XPRM_F_WRITE` Default output stream `XPRM_F_ERROR` Default error stream `XPRM_F_LINBUF` Use line buffering
`filename`	Extended file name to be used for the stream.

Return value

0 if successful, 1 otherwise.

Further information

1. This function sets default IO streams to be used by a model or by the entire system. Model streams can be changed only when the model is not running. Each stream is associated to an extended file name (*i.e.* IO drivers can be used). For output streams, `XPRM_F_LINBUF` may be specified (*e.g.*`XPRM_F_WRITE+XPRM_F_LINBUF`) in order to enable line buffering for the corresponding stream (the error stream is always open using line buffering).

2. For input and output streams, the filename is stored and streams are actually open when execution of the model starts: in case of an invalid file name, the error is not reported by this function. The error stream is immediately opened so in the case of an invalid file name is detected by this function. If the first parameter is `NULL`, this function defines the corresponding global stream: it is used as the default when a model is loaded and whenever no model information is available (*e.g.* compilation errors, error on modules, *etc.*). This option can be used only if no model is currently loaded in memory.

3. Using an empty string as the file name implies resetting to the original default stream: for a model this is the corresponding global stream, if no model is provided, this is the operating system stream.

XPRMresetmod

Purpose

Reset a model.

Synopsis

```
int XPRMresetmod(XPRMmodel model);
```

Argument

`model` Reference to a model

Return value

0 if successful, 1 otherwise.

Further information

This function resets a model after its execution: all resources it has allocated are released. The model returns to its state just after it has been loaded into memory. Note that this function is automatically called before a model is unloaded or run.

Related topics

`XPRMrunmod`, `XPRMdbg_runmod`, `XPRMunloadmod`.

XPRMrunmod

Purpose

Run a model.

Synopsis

```
int XPRMrunmod(XPRMmodel model, int *exitcode, const char *parlist);
```

Arguments

`model`	Reference to a model
`exitcode`	Pointer to an area where the model exit code value is returned
`parlist`	String composed of model parameter initializations separated by commas, may be `NULL`

Return value

Execution status. Possible values are:

`XPRM_RT_OK`	Normal termination
`XPRM_RT_ERROR`	An error occured during execution
`XPRM_RT_MATHERR`	Mathematical error (*e.g.* division by zero)
`XPRM_RT_IOERR`	Input/output error (*e.g.* cannot open file)
`XPRM_RT_NULL`	A NULL reference error occurred
`XPRM_RT_LICERR`	Execution could not start because no license was available
`XPRM_RT_STOP`	Bit set if execution has been interrupted
`XPRM_RT_BREAK`	Interruption because of a breakpoint (see Section 1.3)

Further information

1. This function executes the given model. The parameter `parlist` may be used to initialize the model parameters of the program as well as the control parameters of the modules and packages in use (*e.g.* `"PAR1=12,mypkg.par='tutu'"`). The special model parameter `workdir` defines the initial working directory of the model.

2. In the return value, the bit `XPRM_RT_STOP` is set if the execution of the model has been interrupted by a call to the function `XPRMstoprunmod`.

3. The parameter `exitcode` receives the result of the execution (*i.e.* parameter value of the “exit” procedure or `0` if the routine was not called), its value is meaningful only when the execution succeeded (*i.e.* the status is `XPRM_RT_OK`).

Related topics

`XPRMdbg_runmod`, `XPRMisrunmod`, `XPRMstoprunmod`, `XPRMaddparamstr`.

XPRMaddparamstr

Purpose

Add a parameter definition to a parameter string.

Synopsis

```
int XPRMaddparamstr(char **parlist,size_t *maxlen,const char *parm);
```

Arguments

`parlist` Reference to the buffer holding the parameter string (that may be `NULL`)

`maxlen` Reference to the size of the buffer of the parameter string (its value must be 0 if the buffer is `NULL`)

`parm` The parameter to add. It must be of the form `paramname=value`

Return value

0 if successful, 1 if memory cannot be allocated.

Example

```
char *parstr=NULL;
size_t maxlen=0;

XPRMaddparamstr(&parstr,&maxlen,"P1=10");
XPRMaddparamstr(&parstr,&maxlen,"P2=some text");
printf("%s\n",parstr);
free(parstr);
```

Further information

1. This function helps constructing the parameter string to be passed to `XPRMrunmod` by appending to an existing parameter string a new definition with the appropriate quoting.
2. The routine performs memory allocation and expects the provided buffer to have been allocated with the C function `malloc` (it is automatically reallocated as necessary). Note that the size pointed by `maxlen` corresponds to the size of the buffer, not the length of the parameter string (that will always be smaller).

Related topics

`XPRMrunmod`.

XPRMisrunmod

Purpose

Check if a model is running.

Synopsis

```
int XPRMisrunmod(XPRMmodel model);
```

Argument

`model` Reference to a model

Return value

1 if the model is running, 0 otherwise.

Further information

This function checks if the given model is being run.

Related topics

`XPRMrunmod`, `XPRMdbg_runmod`, `XPRMstoprunmod`.

XPRMstoprunmod

Purpose

Stop a running model.

Synopsis

```
void XPRMstoprunmod(XPRMmodel model);
```

Argument

`model` Reference to a model

Further information

This function interrupts the execution of a model.

Related topics

`XPRMisrunmod`, `XPRMrunmod`, `XPRMdbg_runmod`, `XPRMtermrunmod`.

XPRMtermrunmod

Purpose

Terminate execution of a model.

Synopsis

```
void XPRMtermrunmod(XPRMmodel model);
```

Argument

`model` Reference to a model

Further information

This function terminates the execution of a model. The termination is effective even if a debugger is controlling the model.

Related topics

`XPRMisrunmod`, `XPRMrunmod`, `XPRMstoprunmod`, `XPRMdbg_runmod`.

XPRMunloadmod

Purpose

Unload a model.

Synopsis

```
int XPRMunloadmod(XPRMmodel model);
```

Argument

`model` Reference to a model

Return value

0 if successful, 1 otherwise.

Further information

This function unloads the given model. All resources used by this model, including modules, are released. The function fails if the model is being run.

Related topics

`XPRMloadmod`.

XPRMgetannotations

Purpose

Retrieve annotations of a model.

Synopsis

```
int XPRMgetannotations(XPRMmodel model, const char *ident, const char
      *prefix, const char **ann, int maxann);
```

Arguments

`model`	Reference to a model
`ident`	Symbol to be considered or `NULL` for global declarations
`prefix`	Filtering prefix
`ann`	Array of size `maxann` where to store the annotations (can be `NULL`)
`maxann`	Size of `ann` (to get up to `maxann/2` annotations)

Return value

Size of the array required to get all annotations (two times the number of found annotations).

Further information

1. This function retrieves the annotations associated to the given symbol using a prefix as a filter (*e.g.* use `"doc."` to get all the documentation annotations). The result is stored in the provided array: each annotation occupies 2 entries in the array (the first one for the name of the annotation and the following one for its value).

2. The returned value may exceed `maxann` (but no more than `maxann` entries are recorded in the array). To get the required size for `ann` the function may be called with a `NULL` array.

Related topics

`XPRMgetnextanident`, `XPRMgetdsoannotations`.

XPRMgetbimprefix

Purpose

Get the list of prefixes for loading of packages.

Synopsis

```
const char *XPRMgetbimprefix(void);
```

Return value

The list of prefixes as a text string.

Further information

The value returned by this function is valid until the next call to `XPRMsetbimprefix`.

Related topics

`XPRMsetbimprefix`.

XPRMgetmodprop

Purpose

Get a property of a model.

Synopsis

```
int XPRMgetmodprop(XPRMmodel model, int prop, void *value);
```

Arguments

`model` Reference to a model

`prop` Property to retrieve. Possible values:

`XPRM_PROP_NAME`	Model name (*cf.* `model` statement)
`XPRM_PROP_ID`	Order number
`XPRM_PROP_VERSION`	Model version
`XPRM_PROP_SYSCOM`	System comment
`XPRM_PROP_USRCOM`	User comment
`XPRM_PROP_SIZE`	Amount of memory (in bytes) used by the model
`XPRM_PROP_NBBIM`	Number of BIM files loaded for this model
`XPRM_PROP_DATE`	Compilation date
`XPRM_PROP_SECSTAT`	Security status
`XPRM_PROP_SKEYFP`	Key fingerprint (if the BIM file was signed)
`XPRM_PROP_NBTYPES`	Number of types
`XPRM_PROP_UNAME`	Unique model name
`XPRM_PROP_COMPAT`	Smallest compatible version of a package (0 if not available)

`value` Pointer to an area where the model property is returned

Return value

0 if successful, 1 otherwise.

Further information

1. This function returns information about a given model. The type of the property (specified via the `prop` argument) decides how the argument `value` is interpreted: a reference to an integer is used for `ID`, `VERSION`, `SECSTAT`, `NBBIM` and `NBTYPES`; a reference to a `size_t` for `SIZE` and `DATE` (should be casted to the C type `time_t`); and a reference to a `XPRMstring` for the other properties. The returned version number is coded as an integer, for example, `1.2.3` is coded as `1002003`.

2. The *security status* is a bit encoded integer indicating whether the BIM file was encrypted (value `XPRM_SECSTAT_CRYPTED`); signed (value `XPRM_SECSTAT_SIGNED`). If the BIM file was signed, the bits `XPRM_SECSTAT_VERIFIED` and `XPRM_SECSTAT_UNVERIFIED` indicate whether the signature was valid (if none of these bits is set the signature was not checked).

3. The property `NBBIM` returns the number of BIM files loaded for the model: if there is no dynamic package dependency (either the model does not use any package or all packages are imported) this property will be `1`.

4. The function can also be used to retrieve information about packages dynamically loaded for the model (*i.e.* property `NBBIM` is greater than `1`) by using the macro `XPRM_PROP` to generate the `prop` parameter. For instance to retrieve the version number of the first package required by the model use `XPRM_PROP(XPRM_PROP_VERSION,1)`.

XPRMgetnextdep

Purpose

Get the next dependency (module or package) of a model or package.

Synopsis

```
void *XPRMgetnextdep(XPRMmodel model, void *ref, const char **name,
       int *version, int *dso_pkg);
```

Arguments

`model`	Reference to a model
`ref`	Reference pointer or `NULL`
`name`	Returned name of the package/module
`version`	Returned version of the package/module
`dso_pkg`	Returned type of the dependency: `0` for a module, `1` for a required package and `2` for an imported package

Return value

Reference pointer for the next call to `XPRMgetnextdep`.

Further information

1. This function returns the next dependency of a model: model dependencies are the packages it includes and the modules it requires. The second parameter is used to store the current location in the table of dependencies; if this parameter is `NULL`, the first dependency of the table is returned. This function returns `NULL` if it is called with the reference to the last dependency defined by the given model. Otherwise, the returned value can be used as the input parameter `ref` to get the following dependency and so on.

2. The data is returned ordered: the modules are first reported, then the required packages and finally the imported packages.

3. If the model (or package) has dynamic dependencies and it has not been loaded with option `"-l"` (see `XPRMloadmod`) the returned information includes data related to the loaded packages.

Related topics

`XPRMgetnextmoddso`.

XPRMgetnextmod

Purpose

Get the next model.

Synopsis

```
XPRMmodel XPRMgetnextmod(XPRMmodel model);
```

Argument

`model` Reference to a model or `NULL`

Return value

Reference to a model or `NULL` .

Further information

Mosel maintains a list of loaded models. This function returns the next model held in the internal list after the given model. If the input parameter is set to `NULL`, the first model in the list is returned. If the given model is the last in the list, `NULL` is returned.

XPRMgetnextmoddso

Purpose

Get the next module loaded for a model.

Synopsis

```
void *XPRMgetnextmoddso(XPRMmodel model, void *ref, XPRMdsolib *dso);
```

Arguments

`model` Reference to a model

`ref` Reference pointer or `NULL`

`dso` Returned reference to a dynamic shared object

Return value

Reference pointer for the next call to `XPRMgetnextmoddso`.

Further information

When loading a BIM file (see `XPRMloadmod`) additional modules may be loaded: this function can be used to enumerate the modules that have been loaded for a given model. The second parameter is used to store the current location in the table of loaded DSOs; if this parameter is `NULL`, the first module of the table is returned. This function returns `NULL` if it is called with the reference to the last module loaded for the given model. Otherwise, the returned value can be used as the input parameter `ref` to get the following dependency and so on.

Related topics

`XPRMgetnextdep`.

XPRMfindmod

Purpose

Find a model by its name or order number.

Synopsis

```
XPRMmodel XPRMfindmod(const char *name, int number);
```

Arguments

`name` Name of a model or `NULL`

`number` Model order number or `-1`

Return value

Reference to a model or `NULL` if the model does not exist.

Further information

In the list of loaded models, each model is characterised by its internal name (the name stored in the BIM file, not the filename) and an order number (this number is automatically assigned to the model when it is loaded). This function returns a model that is identified either by its name (`number = -1`) or by its order number (`name = NULL`). If both parameters are defined, the function returns a pointer to the model defined by `name`.

Related topics

`XPRMfindmod`.

1.2 Post processing interface

The post processing interface gives easy access to the internal database of Mosel. This database is composed of all model objects that are defined in a BIM file (like constants) or created during the execution of a model (like arrays). Obviously the dynamically created objects are only available after the model has been run.
Note that the dictionary is not available if the model has been compiled with the option "s" (*strip symbols*) and no identifier has been explicitly published (refer to the description of the `public` qualifier in declarations): such a model cannot be accessed through the post processing interface.

`XPRMdsotyptostr`	Get a string representation from an external type reference.	p. 39
`XPRMfindattrdesc`	Find an attribute descriptor from its name.	p. 43
`XPRMfindident`	Find an identifier in the dictionary.	p. 40
`XPRMfindtypecode`	Find the code associated to a type.	p. 44
`XPRMgetattr`	Get an attribute of an entity.	p. 45
`XPRMgetnextanident`	Get the next annotated identifier in the dictionary.	p. 46
`XPRMgetnextattrdesc`	Get the next attribute descriptor.	p. 53
`XPRMgetnextident`	Get the next identifier in the dictionary.	p. 47
`XPRMgetnextparam`	Get the next parameter of the model.	p. 48
`XPRMgetnextpbcomp`	Enumerate components of a problem type.	p. 49
`XPRMgetnextpkgparam`	Enumerate control parameters of a package.	p. 50
`XPRMgetnextproc`	Get the next overloaded version of a procedure or function.	p. 51
`XPRMgetnextreq`	Enumerate requirements of a package.	p. 52
`XPRMgetprocinfo`	Get the procedure/function information.	p. 54
`XPRMgettypeprop`	Get a property of a type.	p. 55

XPRMdsotyptostr

Purpose

Get a string representation from an external type reference.

Synopsis

```
int XPRMdsotyptostr(XPRMmodel model,int type, void *value, char *str,
      int size);
```

Arguments

`model`	Reference to a model
`type`	Code of the external type
`value`	Entity to convert
`str`	Destination buffer
`size`	Size of `str`

Return value

Size of the generated string (excluding the terminating null byte) or -1 in case of error.

Further information

1. This function converts an entity of an external type or a union into its textual representation. If the type does not support this conversion, the function produces a string using the address of the entity.
2. The returned length might be larger than `size-1`. In this case this return value is the minimum buffer size (not including the terminating null byte) required to generate the text representation and the destination string `str` is not necessarily populated.

XPRMfindident

Purpose

Find an identifier in the dictionary.

Synopsis

```
int XPRMfindident(XPRMmodel model, const char *name,
      XPRMalltypes *value);
```

Arguments

`model` Reference to a model

`name` Identifier

`value` Pointer to an area where the dictionary entry is returned

Return value

Aggregated type information of the returned dictionary entry, or 0 if the identifier is not registered.

Further information

1. This function returns the dictionary entry of a given identifier for a given model together with its type. The returned type information is bit encoded and associates a *type code* and a *structure* that can be extracted using the macros `XPRM_TYP(t)` and `XPRM_STR(t)`.

   The possible structures are:
   `XPRM_STR_CONST` the object is a constant scalar
   `XPRM_STR_REF` the object is a reference
   `XPRM_STR_LIST` the object is a list
   `XPRM_STR_SET` the object is a set
   `XPRM_STR_ARR` the object is an array
   `XPRM_STR_PROC` the object is a procedure or function
   `XPRM_STR_MEM` the object is a memory block
   `XPRM_STR_UTYP` the object is a user defined type

   Depending on the structure, the possible type codes are:
   `XPRM_TYP_NOT` no type (procedure or list)
   `XPRM_TYP_INT` integer (constant, reference, list, set, array, function)
   `XPRM_TYP_REAL` real (constant, reference, list, set, array, function)
   `XPRM_TYP_STRING` text string (constant, reference, set, array, function)
   `XPRM_TYP_BOOL` Boolean (constant, reference, list, set, array, function)
   `XPRM_TYP_MPVAR` decision variable (reference, list, set, array)
   `XPRM_TYP_LINCTR` linear constraint (reference, list, set, array)

   Any other value designates an external type (type provided by a module or defined in the model). Moreover, if the structure is `XPRM_STR_UTYP`, the identifier is the name of a user type and the value (an integer) corresponds to the expanded form of this type (see `XPRMgettypeprop`). Otherwise, the function `XPRMgettypeprop` can be used to get the name and the properties of this type.
   The union `XPRMalltypes` groups all possible types and the result of a call to `XPRMfindident` is decoded as follows depending on the structure:
   `value.integer` for constant, reference or user type
   `value.real` for constant or reference
   `value.string` for constant or reference
   `value.boolean` for constant or reference
   `value.mpvar` for reference
   `value.linctr` for reference
   `value.list` for list (to be used as input for list functions)
   `value.set` for set (to be used as input for set functions)
   `value.array` for array (to be used as input for array functions)
   `value.ref` for a reference to an external type (available operations depend on the actual type)
   `value.proc` for procedure and function
   `value.memblk` for memory block

   Memory blocks are generated by the `mem` IO driver when used with a label. Blocks created this way can be found using the label: the name is linked to the following structure describing the block:

   ```
   typedef struct
      {
       void *ref;    /* Base address of the block */
       size_t size;  /* Size of the block */
      } XPRMmemblk;
   ```

   Note that memory blocks allocated by Mosel are managed by the memory manager of the IO driver and must not be explicitly released.

2. When the model is compiled with debug information (flags `"-g"` or `"-G"`) both public and private symbols may be returned by this function. The macro `XPRM_IS_PUBLIC(t)` applied to the encoded type makes it possible to identify public entities.

Related topics

`XPRMgetnextident`, `XPRMdbg_getnextlocal`, `XPRMgettypeprop`.

XPRMfindattrdesc

Purpose

Find an attribute descriptor from its name.

Synopsis

```
XPRMattrdesc XPRMfindattrdesc(XPRMmodel model, int type, const char **name,
      int *atype);
```

Arguments

`model` Reference to a model

`type` Type number for the attribute

`name` Name of the attribute

Return value

Reference pointer to an attribute descriptor or `NULL` if no corresponding function could be found.

Further information

The attribute `att` of an entity of type `T` is obtained by calling the function `get`*`att`* returning an integer, a string, a real or a Boolean and taking as its only argument an entity of type `T`. This routine returns a descriptor of such a function that can be used with `XPRMgetattr` in order to retrieve the corresponding attribute of an entity of the corresponding type.

Related topics

`XPRMgetnextattrdesc`, `XPRMgetattr`.

XPRMfindtypecode

Purpose

Find the code associated to a type.

Synopsis

```
int XPRMfindtypecode(XPRMmodel model, const char *name);
```

Arguments

`model` Reference to a model

`name` Name of a type

Return value

The type code or `-1` if the type cannot be found.

Further information

Each external type (user defined or coming from a module) is identified by a type code. This routine returns the code corresponding to a type name.

Related topics

`XPRMgettypeprop`.

XPRMgetattr

Purpose

Get an attribute of an entity.

Synopsis

```
int XPRMgetattr(XPRMmodel model, XPRMattrdesc attrdesc, void *ref,
      XPRMalltypes *value);
```

Arguments

`model`	Reference to a model
`attrdesc`	An attribute descriptor
`ref`	An entity of the type associated to the attribute
`value`	Pointer to an area where the value of the attribute is returned

Return value

Type number of the returned value or `0` in case of error.

Example

The following example displays the reduced cost of variable `x`.

```
XPRMalltypes x,rcost;
XPRMattrdesc getrcost;
XPRMfindident(model,"x",&x);
getrcost=XPRMfindattrdesc(model,XPRM_TYP_MPVAR,"rcost");
XPRMgetattr(model,getrcost,x.ref,&rcost);
printf("getrcost(x)=%g\n",rcost.real);
```

Further information

1. This function makes it possible to retrieve an attribute of an entity using a type descriptor as returned by `XPRMfindattrdesc`.
2. The internal list of types attributes is deleted each time the model is run or reset: descriptors obtained before a model execution should not be used after the model has been run.

Related topics

`XPRMgetnextattrdesc`, `XPRMfindattrdesc`.

XPRMgetnextanident

Purpose

Get the next annotated identifier in the dictionary.

Synopsis

```
const char *XPRMgetnextanident(XPRMmodel model, void **ref);
```

Arguments

`model` Reference to a model

`ref` Pointer to an area where current location is stored

Return value

An identifier of the symbol table or `NULL` if all identifiers have been returned.

Further information

This function returns the next identifier for which annotations are available. The second parameter is used to store the current location in the table; this reference is updated with every call to this function. If this parameter references a `NULL` pointer, the first identifier of the table is returned. This function returns `NULL` if it is called with the reference to the last identifier in the internal table.

Related topics

`XPRMgetnextident`, `XPRMgetannotations`.

XPRMgetnextident

Purpose

Get the next identifier in the dictionary.

Synopsis

```
const char *XPRMgetnextident(XPRMmodel model, void **ref);
```

Arguments

`model` Reference to a model

`ref` Pointer to an area where current location is stored

Return value

An identifier of the symbol table or `NULL` if all identifiers have been returned.

Further information

1. This function returns the next identifier held in the internal table of symbols. The second parameter is used to store the current location in the table; this reference is updated with every call to this function. If this parameter references a `NULL` pointer, the first identifier of the table is returned. This function returns `NULL` if it is called with the reference to the last identifier in the internal table.

2. The compiler generates automatic names for constant sets (identifiers start with `"@"`) and anonymous types (identifiers start with `"%"`). This function reports only automatic names of sets, however the other symbols can be accessed using `XPRMfindident`.

3. When the model or package is compiled with debug information included, local symbols of imported packages are also available (and listed through this function). In order to avoid name collisions each symbol local to a package is prefixed by the package name and the symbol `~`. For instance the symbol `myctr` defined in the package `mypkg` is stored as `mypkg~myctr`.

4. This function enumerates only globally defined symbols. To get a list of local symbols (like a loop index) during a debugging session use `XPRMdbg_getnextlocal`.

Related topics

`XPRMdbg_getnextlocal`, `XPRMfindident`.

XPRMgetnextparam

Purpose

Get the next parameter of the model.

Synopsis

```
const char *XPRMgetnextparam(XPRMmodel model, void **ref);
```

Arguments

`model` Reference to a model

`ref` Pointer to an area where current location is stored

Return value

The name of the parameter or `NULL` if there is no subsequent parameter.

Further information

This function returns the next parameter of the given model. The second argument is used to store the current location in the list of parameters; this reference is updated with every call to this function. If this argument references a `NULL` pointer, the first parameter of the model is returned. This function returns `NULL` if it is called with the reference to the last parameter in the model as its second argument.

Related topics

`XPRMgetnextpkgparam`.

XPRMgetnextpbcomp

Purpose

Get the next component of a problem type.

Synopsis

```
void *XPRMgetnextpbcomp(XPRMmodel model, void *ref, int typcode,
       int *type);
```

Arguments

`model`	Reference to a model
`ref`	Reference pointer or `NULL`
`typcode`	Type code
`type`	Returned type of the component

Return value

Reference pointer for the next call to `XPRMgetnextpbcomp`.

Further information

1. Problem types are composed by a collection of components (typically one or more main types and the associated extensions) each of which being a native problem type. This function returns the next component of a problem type. The type returned by the function can be decoded in the same way as for a type returned by `XPRMfindident`. The second parameter is used to store the current location in the table of components of the type; if this parameter is `NULL`, the first component of the table is returned. This function returns `NULL` if it is called with the reference to the last component for the given problem type. Otherwise, the returned value can be used as the input parameter `ref` to get the following component and so on.

2. The routine will return a type `0` as the first component of problem types including an `mpproblem` component.

3. A problem type has at least one component: the first component of a native type is the type itself (*i.e.* the parameter `type` receives the value of `typcode`).

XPRMgetnextpkgparam

Purpose

Get the next control parameter in the list of the given package.

Synopsis

```
void *XPRMgetnextpkgparam(XPRMmodel model, void *ref, const char **name,
      const char **desc, int *type);
```

Arguments

`model`	Reference to a model
`ref`	Reference pointer or `NULL`
`name`	Returned name of the control parameter
`desc`	Reserved for future use (should be `NULL`
`type`	Returned type of the control parameter

Return value

Reference pointer for the next call to `XPRMgetnextpkgparam`.

Further information

This function returns the next control parameter of the given package (a model cannot publish control parameters). The second parameter is used to store the current location in the table of control parameters; if this parameter is `NULL`, the first control parameter of the table is returned. This function returns `NULL` if it is called with the reference to the last parameter of the given module. Otherwise, the returned value can be used as the input parameter `ref` to get the following control parameter and so on. The type can be decoded using the macro `XPRM_TYP`. Moreover, the bits `XPRM_CPAR_READ` and `XPRM_CPAR_WRITE` are set to indicate if the parameter can be read or written respectively (using `getparam` and `setparam`).

Related topics

`XPRMgetnextparam`.

XPRMgetnextproc

Purpose

Get the next overloaded version of a procedure or function.

Synopsis

```
XPRMproc XPRMgetnextproc(XPRMproc proc);
```

Argument

`proc` Reference to a procedure or function

Return value

A procedure or function reference or `NULL` if no overloading subroutine is defined.

Further information

This function returns the following overloading defined for the given subroutine. A subroutine may be defined several times in a model with different sets of parameters. This function gives access to all the defined overloaded versions of a subroutine.

Related topics

`XPRMgetprocinfo`.

XPRMgetnextreq

Purpose

Get the next requirement of a package.

Synopsis

```
void *XPRMgetnextreq(XPRMmodel model, void *ref, const char **name,
      int *type,void **data);
```

Arguments

`model` Reference to a model

`ref` Reference pointer or `NULL`

`name` Returned name of the requirement

`type` Returned type

`data` Returned extra data for the type

Return value

Reference pointer for the next call to `XPRMgetnextreq`.

Further information

This function returns the next requirement of a package: requirements of a package are the symbols it declares but that must be defined by the model using it. The type returned by the function can be decoded in the same way as for a type returned by `XPRMfindident`. The information returned via the last argument depends on the type: for a scalar, a set or a list a `NULL` pointer is returned; for an array the list of the names of the indexing sets is returned through a text string (for instance the array `a:array(S1,S2)` has the following data string: `"S1,S2"`). In the case of a subroutine, an `XPRMproc` reference is provided: this can be used with `XPRMgetprocinfo` for getting information on the required routine. The second parameter is used to store the current location in the table of requirements; if this parameter is `NULL`, the first requirement of the table is returned. This function returns `NULL` if it is called with the reference to the last requirement defined by the given model. Otherwise, the returned value can be used as the input parameter `ref` to get the following requirement and so on.

XPRMgetnextattrdesc

Purpose

Get the next attribute descriptor.

Synopsis

```
XPRMattrdesc XPRMgetnextattrdesc(XPRMmodel model,XPRMattrdesc ref,int
      *ntype,const char **name,int *atype);
```

Arguments

`model` Reference to a model

`ref` Reference an attribute descriptor or `NULL`

`ntype` Type number supporting the attribute (may be `NULL`)

`name` Name of the attribute (may be `NULL`)

`atype` Type of the attribute (may be `NULL`)

Return value

Reference pointer for the next attribute descriptor or `NULL` if no further descriptor is available.

Further information

1. This function is used to enumerate attribute descriptors of a model. The second parameter is used to store the current location in the list of descriptors; if this parameter is `NULL`, the first descriptor of the list is returned. This function returns `NULL` if it is called with the reference to the last descriptor. Otherwise, the returned value can be used as the input parameter `ref` to get the following descriptor and so on.

2. The `name`, `ntype` and `atype` are the returned attribute properties associated to the provided descriptor `ref`.

Related topics

`XPRMfindattrdesc`, `XPRMgetattr`.

XPRMgetprocinfo

Purpose

Get the procedure/function information.

Synopsis

```
int XPRMgetprocinfo(XPRMproc proc, const char **partyp, int *nbpar,
      int *type);
```

Arguments

`proc` Reference to a procedure or function

`partyp` Returned string of parameter types

`nbpar` Returned number of parameters

`type` Bit encoded returned type of the subroutine

Return value

0 if successful, 1 otherwise.

Further information

This function provides information about a procedure or function. The type can be decoded like for any other identifier of a model. Note that a procedure has no return type (`XPRM_TYP(type)=XPRM_TYP_NOT`). The string of parameter types is a text string describing which parameters are expected by the function, it is its *signature*. This string is composed with the following characters:

`i` an integer

`r` a real

`s` a text string

`b` a Boolean

`v` a decision variable (type `mpvar`)

`c` a linear constraint (type `linctr`)

`I` a range set

`a` an array (of any kind)

`e` a set (of any type)

`l` a list (of any type)

`|xxx|` external type named '*xxx*'. A type code may also be given as '*%???*' where '*???*' (3 hexadecimal digits) is the code number

`!xxx!` the set named '*xxx*'

`Andx.t` an array indexed by '*ndx*' of the type '*t*'. '*ndx*' is a string describing the type of each indexing set. '*ndx*' may be omitted in which case any array of type '*t*' is a valid parameter.

`Et` a set of type '*t*'

`Lt` a list of type '*t*'

`?` any type

`*` function with variable number of parameters (this character is the last one of the string)

For instance, the procedure:

```
proc(n:integer,
     tab:array(range, set of real, myset) of string,
     flag:boolean)
```

has the signature "iAIr!myset!.sb".

Related topics

`XPRMgetnextproc`.

XPRMgettypeprop

Purpose

Get a property of a type.

Synopsis

```
void *XPRMgettypeprop(XPRMmodel model, int type,
       int prop, void *value);
```

Arguments

`model` Reference to a model

`type` Code of a type

`prop` Property to retrieve. Possible values:

`XPRM_TPROP_NAME`	Name of the type
`XPRM_TPROP_FEAT`	Encoded features
`XPRM_TPROP_EXP`	Expanded type
`XPRM_TPROP_PBID`	Problem index (negative if the type is not a problem)
`XPRM_TPROP_ITYPS`	Array of field types for a constant reference containing a record
`XPRM_TPROP_NBELT`	Type size information (see comments below)
`XPRM_TPROP_SIGN`	Signature of a procedure or function type

`value` Pointer to an area where the type property is returned

Return value

0 if successful, -1 if `type` is not valid and 1 if `prop` is not supported.

Further information

1. This function returns a property of an external type (types provided by modules or user defined). For the properties `XPRM_TPROP_NAME` and `XPRM_TPROP_SIGN`, `value` must be of type `const char**`, for `XPRM_TPROP_ITIPS` it must be `void**` and for the other properties, the result is returned as an integer (`int*`).

2. The type features are bit encoded as follows:
   `XPRM_MTP_CREAT` Creation function available for this type
   `XPRM_MTP_DELET` Deletion function available for this type
   `XPRM_MTP_TOSTR` Type can be converted to a string
   `XPRM_MTP_FRSTR` Type can be initialized from a string
   `XPRM_MTP_PRTBL` Type can be converted to a string after execution
   `XPRM_MTP_RFCNT` Type implements reference count
   `XPRM_MTP_COPY` Type implements copy: it may be used in assignments
   `XPRM_MTP_APPND` The copy function of this type supports appending
   `XPRM_MTP_ORSET` The copy function of this type can only be used to reset an object
   `XPRM_MTP_PROB` Type is a problem
   `XPRM_MTP_CMP` Test of equality is supported by this type
   `XPRM_MTP_SHARE` An entity of this type can be declared as shared
   `XPRM_MTP_TFBIN` Type supports export/import in binary format
   `XPRM_MTP_ORD` Type supports comparison
   `XPRM_MTP_CONST` Type supports constant definition
   `XPRM_MTP_ANDX` Type is an array indexer
   `XPRM_MTP_NAMED` Type supports name association

3. The expanded type is available for user defined types only: it corresponds to the actual type (including structure information) associated to a user defined type code. For instance, assuming the type `myset` is defined as a set of integer, getting the type expansion for the code associated to `myset` will give `XPRM_STR_SET|XPRM_TYP_INT` indicating that a reference to an entity of type `myset` has to be handled with functions for sets.

4. Trying to get the expanded type of a module type or the features of a user defined type is an error: the function returns 1. This can be used to identify module types.

5. A user type which expanded type has structure `XPRM_STR_REC` is a record type. The public fields of a record type may be enumerated with `XPRMgetnextfield`.

6. A user type which expanded type has structure `XPRM_STR_UNION` is a union type. The compatible types of a union type may be enumerated with `XPRMgetnextuncomptype`.

7. A user type which expanded type has structure `XPRM_STR_PROB` is a problem type. The components of a problem type may be enumerated with `XPRMgetnextpbcomp`. Note that problem types are also implemented as native types. In this case, the flag `XPRM_MTP_PROB` will be set in the type features.

8. If a type refers to a native or record constant the expanded type has structure `XPRM_STR_CSREF`. The function `XPRMcsrtoref` might be used to retrieve the actual object associated to a constant.

9. The information returned for the property `XPRM_TPROP_NBELT` depends on the kind of type considered: for an array this is the number of dimensions (`XPRMgetarrdim`), for a subroutine this corresponds to the number of parameters it requires (`XPRMgetprocinfo`), for a union this is the number of compatible types of this union (`XPRMgetnextuncomptype`), for a record this is the number of fields it contains (`XPRMgetnextfield`) and for a problem this gives the number of components (`XPRMgetnextpbcomp`).

Related topics

`XPRMgetnextfield`, `XPRMgetnextpbcomp`, `XPRMfindtypecode`.

XPRMcsrtoref

Purpose

Get the constant object associated to a CSREF structure.

Synopsis

```
void *XPRMcsrtoref(XPRMmodel model, void *csr);
```

Arguments

`ctx` Mosel's execution context

`csr` A reference of an entity of structure `XPRM_STR_CSREF`

Return value

A reference to the constant object associated to the CSREF entity.

Further information

Mosel saves references to constant objects of native or record types in entities of structure `XPRM_STR_CSREF`. This function makes it possible to access the actual object associated to a constant of this kind.

Related topics

`XPRMgettypeprop`.

1.2.1 Lists

Lists are an ordered collection of objects. The functions available here allows to get properties of a list (size and type) as well as enumerate all elements it contains.

`XPRMgetlistsize`	Get the size of a list.	p. 60
`XPRMgetlisttype`	Get the type of a list.	p. 61
`XPRMgetnextlistelt`	Get the next element of a list.	p. 62
`XPRMgetprevlistelt`	Get the previous element of a list.	p. 63

XPRMgetlistsize

Purpose

Get the size of a list.

Synopsis

```
unsigned int XPRMgetlistsize(XPRMlist list);
```

Argument

`list` Reference to a list

Return value

Size (=number of elements) of the list.

Further information

This function returns the size, that is the number of elements, of a given list.

Related topics

`XPRMgetlisttype`.

XPRMgetlisttype

Purpose

Get the type of a list.

Synopsis

```
int XPRMgetlisttype(XPRMlist list);
```

Argument

`list` Reference to a list

Return value

List type.

Further information

The type of a list is both the type of all elements of the list (as a type code) and the storage class used for the list. The element type code can be extracted using the macro `XPRM_TYP(type)`. Note that a list with no type (`XPRM_TYP_NOT`) contains elements of different types. In this case the type of each element has to be checked when enumerating the content of the list with `XPRMgetnextlistelt`. The storage class can be extracted using the macro `XPRM_GRP(type)`. If the bit `XPRM_GRP_DYN` is set, the list is dynamic and may be modified.

Related topics

`XPRMgetlistsize`,`XPRMgetnextlistelt`.

XPRMgetnextlistelt

Purpose

Get the next element of a list.

Synopsis

```
void *XPRMgetnextlistelt(XPRMlist list, void *ref, int *type, XPRMalltypes
      *value);
```

Arguments

`list` Reference to a list

`ref` Reference pointer or `NULL`

`type` Returned type

`value` Pointer to an area where the result is returned

Return value

Reference pointer for the next call to `XPRMgetnextlistelt`.

Further information

This function is used to enumerate elements of a list. The second parameter is used to store the current location in the list; if this parameter is `NULL`, the first element of the list is returned. This function returns `NULL` if it is called with the reference to the last element. Otherwise, the returned value can be used as the input parameter `ref` to get the following element and so on. The function returns in the third argument the type of the object stored in `value` (as a type code): this corresponds to the value returned by `XPRMgetlisttype` if all elements have the same type.

Related topics

`XPRMgetlisttype`,`XPRMgetprevlistelt`.

XPRMgetprevlistelt

Purpose

Get the previous element of a list.

Synopsis

```
void *XPRMgetprevlistelt(XPRMlist list, void *ref, int *type, XPRMalltypes
      *value);
```

Arguments

`list` Reference to a list

`ref` Reference pointer or `NULL`

`type` Returned type

`value` Pointer to an area where the result is returned

Return value

Reference pointer for the next call to `XPRMgetnextlistelt`.

Further information

This function is used to enumerate elements of a list in reverse order. The second parameter is used to store the current location in the list; if this parameter is `NULL`, the last element of the list is returned. This function returns `NULL` if it is called with the reference to the first element. Otherwise, the returned value can be used as the input parameter `ref` to get the following element and so on. The function returns in the third argument the type of the object stored in `value`: this correspond to the value returned by `XPRMgetlisttype` if all elements have the same type.

Related topics

`XPRMgetlisttype`,`XPRMgetnextlistelt`.

1.2.2 Sets

Sets are used to index arrays: any model using arrays also uses sets even if no set has been defined explicitly. Note that a *range* is a special case of a set of integers which contains all consecutive integers in a given interval.

`XPRMgetelsetndx`	Get the index of a set element.	p. 68
`XPRMgetelsetval`	Get the value of an element of a set.	p. 67
`XPRMgetfirstsetndx`	Get the first index of a set.	p. 69
`XPRMgetlastsetndx`	Get the last index of a set.	p. 70
`XPRMgetsetsize`	Get the size of a set.	p. 65
`XPRMgetsettype`	Get the type of a set.	p. 66

XPRMgetsetsize

Purpose

Get the size of a set.

Synopsis

```
unsigned int XPRMgetsetsize(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Size (=number of elements) of the set.

Further information

This function returns the size, that is the number of elements, of a given set.

Related topics

`XPRMgetsettype`.

XPRMgetsettype

Purpose

Get the type of a set.

Synopsis

```
int XPRMgetsettype(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Set type.

Further information

The type of a set is both the type of all elements of the set (as a type code) and the storage class used for the set. The element type code can be extracted using the macro `XPRM_TYP(type)`. The storage class can be extracted using the macro `XPRM_GRP(type)`. If the bit `XPRM_GRP_GEN` is set then the set is a general set as opposed to a range set. If the bit `XPRM_GRP_DYN` is set, the set is dynamic and may be extended.

Related topics

`XPRMgetsetsize`.

XPRMgetelsetval

Purpose

Get the value of an element of a set.

Synopsis

```
XPRMalltypes *XPRMgetelsetval(XPRMset set, int ind, XPRMalltypes *value);
```

Arguments

`set` Reference to a set

`ind` Index number

`value` Pointer to an area where the result is returned

Return value

The third argument or `NULL`.

Further information

This function returns the value of the element of a given set denoted by the given index number. The result is copied to the argument `value`.

Related topics

`XPRMgetelsetndx`.

XPRMgetelsetndx

Purpose

Get the index of a set element.

Synopsis

```
int XPRMgetelsetndx(XPRMmodel model, XPRMset set, XPRMalltypes *elt);
```

Arguments

`model` Reference to a model

`set` Reference to a set

`elt` Reference to the element

Return value

Index of a set element or a negative value if the element is not contained in the set.

Further information

This function returns the index of a given element of a set.

Related topics

`XPRMgetfirstsetndx`, `XPRMgetlastsetndx`, `XPRMgetelsetndx`.

XPRMgetfirstsetndx

Purpose

Get the first index of a set.

Synopsis

```
int XPRMgetfirstsetndx(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Index of the first element in the set.

Further information

This function returns the index of the first element of a given set.

In a range set, the lowest value (lower range bound) is returned. In a set of strings, the first element always has the index (= order number) 1. It is recommended to test whether the set is not empty (using function `XPRMgetsetsize`) before calling this function.

Related topics

`XPRMgetlastsetndx`, `XPRMgetsetsize`.

XPRMgetlastsetndx

Purpose

Get the last index of a set.

Synopsis

```
int XPRMgetlastsetndx(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Index of the last element in the set.

Further information

This function returns the index of the last element of a given set.

In a range set the highest value (upper range bound) is returned. In a set of strings the index of the last element always corresponds to the number of elements in the set. It is recommended to test whether the set is not empty (using function `XPRMgetsetsize`) before calling this function.

Related topics

`XPRMgetfirstsetndx`, `XPRMgetsetsize`.

1.2.3 Arrays

In Mosel, arrays are used to store any kind of object, including other arrays or sets. The *type* of the array is also the type of the collected objects. The *storage class* indicates how these objects are stored in memory. In most cases this information can be ignored as all functions accessing arrays automatically handle each special case.

The storage class is encoded in two bits:

`XPRM_GRP_DYN`	The array is a dynamic array: there is no range defined for its indexing sets (*i.e.* there cannot be any "out of range error" for this array as the indexing sets may grow on demand).
`XPRM_GRP_GEN`	The array is a general (= dynamic bounded) array: the number of elements may be augmented up to the range limits specified at its creation.

Typically a "sparse table" uses a storage class of `XPRM_GRP_DYN` or `XPRM_GRP_DYN | XPRM_GRP_GEN` (dynamic or fixed ranges). The Mosel compiler may decide which storage class should be used for each array: even a "dense table" may be created using a storage class of `XPRM_GRP_DYN` if the model does not provide enough information for deciding the actual size of the array at compile time.

For dynamic arrays one may distinguish between *logical* and *true entries*. Assuming an array has been created with the range 1..5, but only entry 3 has been defined, this array has 5 logical entries but only a single true entry. This difference is mainly noticeable in the functions provided for enumerating arrays.

Note that at the library level all arrays are indexed by integers (negative value are allowed). To use text index values, the conversion from the text to the order number must be performed using the function `XPRMgetelsetndx`.

`XPRMchkarrind`	Check whether an index tuple of an array is valid.	p. 81
`XPRMcmpindices`	Compare two index tuples.	p. 82
`XPRMgetarrdim`	Get the number of dimensions of an array.	p. 72
`XPRMgetarrsets`	Get the index sets of an array.	p. 75
`XPRMgetarrsize`	Get the size of an array.	p. 74
`XPRMgetarrtype`	Get the type of an array.	p. 73
`XPRMgetarrval`	Get the value of an array entry.	p. 83
`XPRMgetfirstarrentry`	Get the list of indices of the first entry of an array.	p. 76
`XPRMgetfirstarrtruentry`	Get the list of indices of the first true entry of an array.	p. 79
`XPRMgetlastarrentry`	Get the list of indices of the last entry of an array.	p. 77
`XPRMgetnextarrentry`	Get the list of indices of the next entry of an array.	p. 78
`XPRMgetnextarrtruentry`	Get the list of indices of the next true entry of an array.	p. 80

XPRMgetarrdim

Purpose

Get the number of dimensions of an array.

Synopsis

```
int XPRMgetarrdim(XPRMarray array);
```

Argument

`array` Reference to an array

Return value

Number of dimensions of the array.

Further information

This function returns the number of dimensions of a given array.

Related topics

`XPRMgetarrsets`, `XPRMgetarrsize`, `XPRMgetarrtype`.

XPRMgetarrtype

Purpose

Get the type of an array.

Synopsis

```
int XPRMgetarrtype(XPRMarray array);
```

Argument

`array` Reference to an array

Return value

Type of the array.

Further information

This function returns the type of a given array. The type of an array designates both the type of all entries of the array (as a type code) and the storage class used for that array. The entry's type code can be extracted using the macro `XPRM_TYP(type)`. The storage class can be extracted using the macro `XPRM_GRP(type)` that indicates the internal representation of the array. Possible values are:

`XPRM_ARR_FIX` array is dense and all its indexing sets are constant

`XPRM_ARR_DYFIX` array is dense but at least one of its indexing sets is not constant

`XPRM_ARR_DYN` array is sparse and has been declared as dynamic

`XPRM_ARR_HMAP` array is sparse and has been declared as hashmap

Sparse arrays can be identified using the macro `XPRM_ARR_IS_SPARSE(type)`.

Related topics

`XPRMgetarrdim`, `XPRMgetarrsets`, `XPRMgetarrsize`.

XPRMgetarrsize

Purpose

Get the size of an array.

Synopsis

```
unsigned int XPRMgetarrsize(XPRMarray array);
```

Argument

`array` Reference to an array

Return value

Size (= total number of true entries) of the array.

Further information

This function returns the total number of true entries contained in the array.

Related topics

`XPRMgetarrdim`, `XPRMgetarrsets`, `XPRMgetarrtype`.

XPRMgetarrsets

Purpose

Get the index sets of an array.

Synopsis

```
void XPRMgetarrsets(XPRMarray array, XPRMset sets[]);
```

Arguments

`array` Reference to an array

`sets` `n`-tuple of set references where `n` is the number of dimensions of the array `array`

Further information

This function returns in the parameter `sets` the list of sets that index the array `array`. Each set corresponds to one dimension of the array.

Related topics

`XPRMgetarrdim`, `XPRMgetarrsize`, `XPRMgetarrtype`.

XPRMgetfirstarrentry

Purpose

Get the list of indices of the first entry of an array.

Synopsis

```
int XPRMgetfirstarrentry(XPRMarray array, int indices[]);
```

Arguments

`array` Reference to an array

`indices` `n`-tuple (`n` is the dimension of array `array`) where the index values of the first logical element in the array are returned

Return value

0 if executed succesfully, a positive value otherwise.

Further information

This function returns the index tuple of the first entry of a given array.

Related topics

`XPRMgetfirstarrtruentry`, `XPRMgetlastarrentry`, `XPRMgetnextarrentry`.

XPRMgetlastarrentry

Purpose

Get the list of indices of the last entry of an array.

Synopsis

```
int XPRMgetlastarrentry(XPRMarray array, int indices[]);
```

Arguments

`array` Reference to an array

`indices` `n`-tuple (`n` is the dimension of array `array`) where the index values of the last logical element in the array are returned

Return value

0 if executed succesfully, a positive value otherwise.

Further information

This function returns the index tuple of the last entry in the given array.

Related topics

`XPRMgetfirstarrentry`, `XPRMgetfirstarrtruentry`.

XPRMgetnextarrentry

Purpose

Get the list of indices of the next entry of an array.

Synopsis

```
int XPRMgetnextarrentry(XPRMarray array, int indices[]);
```

Arguments

`array` Reference to an array

`indices` `n`-tuple (`n` is the dimension of array `array`); the input values denote the tuple for which the next (logical) array entry is required; the returned values are the next array entry

Return value

0 if executed succesfully, a positive value otherwise (end of array).

Further information

This function returns the index tuple of the entry following the given tuple in the given array. The next entry in an array is determined by enumerating the last index of the tuple first. The parameter `indices` serves for input and return values at the same time. It is modified by the function to return the tuple corresponding to the next array entry after the tuple that has been input.

Related topics

`XPRMgetfirstarrentry`, `XPRMgetfirstarrtruentry`, `XPRMgetnextarrtruentry`.

XPRMgetfirstarrtruentry

Purpose

Get the list of indices of the first true entry of an array.

Synopsis

```
int XPRMgetfirstarrtruentry(XPRMarray array, int indices[]);
```

Arguments

`array` Reference to an array

`indices` `n`-tuple (`n` is the dimension of array `array`) where the index values of the first defined element in the array are returned

Return value

0 if executed succesfully, a positive value otherwise.

Further information

If the given array has a fixed size (dense array), this function behaves like `XPRMgetfirstarrentry`. With a dynamic array, this function returns the index tuple of the first true entry.

Related topics

`XPRMgetfirstarrentry`, `XPRMgetlastarrentry`, `XPRMgetnextarrentry`.

XPRMgetnextarrtruentry

Purpose

Get the list of indices of the next true entry of an array.

Synopsis

```
int XPRMgetnextarrtruentry(XPRMarray array, int indices[]);
```

Arguments

`array` Reference to an array

`indices` `n`-tuple (`n` is the dimension of array `array`), the input values denote the tuple for which the next true array entry is required; the returned values are the next array entry

Return value

0 if executed succesfully, a positive value otherwise (end of array) .

Further information

If the given array has a fixed size (dense array), this function behaves like `XPRMgetnextarrentry`. With a dynamic array, this function returns the index tuple of the next true entry.

Related topics

`XPRMgetfirstarrentry`, `XPRMgetfirstarrtruentry`, `XPRMgetnextarrentry`.

XPRMchkarrind

Purpose

Check whether an index tuple of an array is valid.

Synopsis

```
int XPRMchkarrind(XPRMarray array, int indices[]);
```

Arguments

`array` Reference to an array

`indices` `n`-tuple of indices where `n` is the dimension of array `array`

Return value

0 if the index tuple lies within the ranges for which the array is defined, a positive value otherwise.

Further information

This function checks whether the given index tuple lies within the range bounds of an array.

Related topics

`XPRMcmpindices`.

XPRMcmpindices

Purpose

Compare two index tuples.

Synopsis

```
int XPRMcmpindices(int nbdim, int ind1[], int ind2[]);
```

Arguments

`nbdim`	number of dimensions (= size of tuples `ind1` and `ind2`)
`ind1, ind2`	Index tuples of size `nbdim`

Return value

`-1`	Tuple `ind1` comes before tuple `ind2`
`0`	Tuples are identical
`1`	Tuple `ind2` comes before tuple `ind1`

Further information

This function compares two index tuples.

Related topics

`XPRMchkarrind`.

XPRMgetarrval

Purpose

Get the value of an array entry.

Synopsis

```
int XPRMgetarrval(XPRMarray array, int indices[], void *adr);
```

Arguments

`array`	Reference to an array
`indices`	`n`-tuple of indices where `n` is the number of dimensions of the array `array`
`adr`	Pointer to the area where the value of the array entry denoted by the index-tuple is returned.

Return value

0 if executed successfully, a positive value otherwise.

Further information

1. This function returns the value of an array entry that corresponds to a given tuple of indices for a given array. The address passed must reference an area large enough to receive data of the array's type: for instance, for an array of reals (type = `XPRM_TYP_REAL`) the `adr` parameter must be of type `double*`.
2. The returned value is 0 (integer, real or Boolean) or `NULL` (other types) if the requested entry does not exist when referencing a dynamic array.

Related topics

`XPRMgetfirstarrentry`, `XPRMgetfirstarrtruentry`, `XPRMgetnextarrentry`, `XPRMgetnextarrtruentry`.

1.2.4 Records

Records are a special kind of user defined types that associate to a an entity a collection of fields. Thanks to the following functions one can enumerate these fields and get the value of a specific field of given record.

`XPRMgetfieldval`	Get the value of a field of a record.	p. 86
`XPRMgetnextfield`	Get the next field of a record type.	p. 85

XPRMgetnextfield

Purpose

Get the next field of a record type.

Synopsis

```
void *XPRMgetnextfield(XPRMmodel model, void *ref, int typcode, const char
      **name, int *type, int *number);
```

Arguments

`model`	Reference to a model
`ref`	Reference pointer or `NULL`
`typcode`	Type code of the record
`name`	Field name
`type`	Field type code
`number`	Field number (in the record)

Return value

Reference pointer for the next call to `XPRMgetnextfield`.

Further information

1. This function is used to enumerate fields of a record type. The second parameter is used to store the current location in the list of fields; if this parameter is `NULL`, the first field of the record is returned. This function returns `NULL` if it is called with the reference to the last field. Otherwise, the returned value can be used as the input parameter `ref` to get the following field and so on.

2. The name, type and number are the returned field properties. The field number is used by the function `XPRMgetfieldval` to retrieve the value of the corresponding field in an object of this record type.

3. All public fields of the record are enumerated by this function but if the model or package has been compiled with debugging information the private fields will also be returned. The macro `XPRM_IS_PUBLIC(t)` applied to the field type may be used to identify public fields.

Related topics

`XPRMgetfieldval`.

XPRMgetfieldval

Purpose

Get the value of a field of a record.

Synopsis

```
void XPRMgetfieldval(XPRMmodel model, int typcode, void *ref, int number,
       XPRMalltypes *value);
```

Arguments

`model`	Reference to a model
`ref`	Reference to the record
`typcode`	Type code of the record
`number`	Field number (in the record)
`value`	Pointer to an area where the field value is returned

Further information

The field number must be obtained from the function `XPRMgetnextfield`. Its value is valid as long as the model is loaded in memory.

Related topics

`XPRMgetnextfield`.

1.2.5 Unions

Unions are a special kind of user defined types that can hold values of different types. Thanks to the following functions one can enumerate the compatible types of a union and get the value of such an entity.

`XPRMgetnextuncomptype`	Get the next compatible type of a union type.	p. 88
`XPRMgetuntype`	Get the type and structure of the value stored in a union.	p. 89
`XPRMgetuntypeid`	Get the type ID of the value stored in a union.	p. 90
`XPRMgetunvalue`	Get the value stored in a union.	p. 91

XPRMgetnextuncomptype

Purpose

Get the next compatible type of a union type.

Synopsis

```
void *XPRMgetnextuncomptype(XPRMmodel model, void *ref, int typcode, int
      *type);
```

Arguments

`model`	Reference to a model
`ref`	Reference pointer or `NULL`
`typcode`	Type code of the union
`type`	Returned type ID

Return value

Reference pointer for the next call to `getnextuncomptype`.

Further information

1. This function is used to enumerate compatible types of a union type. The second parameter is used to store the current location in the list of types; if this parameter is `NULL`, the first compatible type of the union is returned. This function returns `NULL` if it is called with the reference to the last type. Otherwise, the returned value can be used as the input parameter `ref` to get the following type and so on.

2. A union has at least 2 compatible types: the first one is the type used for initialization of the union from a textual representation. An 'any' type is characterized by a 0 value returned for the second (and last) returned type ID.

Related topics

`XPRMgetuntype`, `XPRMgetuntypeid`.

XPRMgetuntype

Purpose

Get the type and structure of the value stored in a union.

Synopsis

```
int XPRMgetuntype(XPRMunion un);
```

Argument

`un` Reference to a union entity

Return value

An encoded type or `-1` if the union is undefined.

Further information

The returned type information is bit encoded and associates a *type code* and a *structure* that can be extracted using the macros `XPRM_TYP(t)` and `XPRM_STR(t)` (see `XPRMfindident`). If the union value is not associated with a type ID (see `XPRMgetuntypeid`) the returned type will report only a structure (`XPRM_STR_ARR`, `XPRM_STR_SET` or `XPRM_STR_LIST`). Otherwise, in the case of a reference (`XPRM_STR_REF`) to a non basic type it will be necessary to check the type properties with `XPRMgettypeprop` in order to identify a structured type.

Related topics

`XPRMgetunvalue`, `XPRMgetuntypeid`.

XPRMgetuntypeid

Purpose

Get the type ID of the value stored in a union.

Synopsis

```
int XPRMgetuntypeid(XPRMunion un);
```

Argument

`un` Reference to a union entity

Return value

A type ID, `0` if the information is not available or `-1` if the union is undefined.

Further information

1. This function retrieves the type ID of the value stored in a union. This information is always defined if the value is a scalar (*i.e.* a constant or a reference) but it will be available for a structured type only if the value comes from an entity defined as an instance of a defined user type.

2. When the function returns 0, the properties of the value stored in the union can still be obtained from `XPRMgetuntype`.

Related topics

`XPRMgetunvalue`, `XPRMgetuntype`.

XPRMgetunvalue

Purpose

Get the value stored in a union.

Synopsis

```
int XPRMgetunvalue(XPRMmodel model, XPRMunion un, XPRMalltypes *value);
```

Arguments

`model` Reference to a model

`un` Reference to a union entity

`value` Pointer to an area where the value is returned

Return value

An encoded type, `0` if the union is undefined or `−1` in case of error.

Further information

The returned type information is either a type ID (see `XPRMgetuntypeid`) or a structure code (`XPRM_STR_ARR`, `XPRM_STR_SET` or `XPRM_STR_LIST`). In the case of a reference to a non basic type it will be necessary to check the type properties with `XPRMgettypeprop` in order to identify a structured type.

Related topics

`XPRMgetuntypeid`, `XPRMgetuntype`.

1.2.6 Problems

Like all statements of a model, the routines presented in this section are executed in the context of an active problem. By default, at the beginning of the processing of a model an initial problem is created: the "main problem". After the end of the execution of a model, this particular problem is *active* but a different problem can be selected using the routine `XPRMselectprob`. The following functions enable the user to access various information related to linear constraints and decision variables created or used in the context of the active problem. With the exception of the `XPRMexportprob` function, all operations in this section require the problem to be loaded into an optimizer either explicitly (*e.g.* procedure 'loadprob' of the module "mmxprs") or implicitly by using an optimization operation (*e.g.* procedure 'maximize' of the module "mmxprs") in the model. If no problem is available (model not run, no constraint created by the model or problem not loaded in an optimizer) a specific default value is returned by each function.

`XPRMexportprob`	Export the active problem to a file.	p. 94
`XPRMgetact`	Get the activity value of a linear constraint.	p. 101
`XPRMgetcsol`	Get the solution value of a linear constraint.	p. 97
`XPRMgetctrnum`	Get the row number of a linear constraint.	p. 103
`XPRMgetdual`	Get the dual value of a linear constraint.	p. 99
`XPRMgetobjval`	Get the objective function value.	p. 95
`XPRMgetprobstat`	Get the problem status of a model.	p. 93
`XPRMgetrcost`	Get the reduced cost value of a variable.	p. 98
`XPRMgetslack`	Get the slack value of a linear constraint.	p. 100
`XPRMgetvarnum`	Get the column number of a decision variable.	p. 102
`XPRMgetvsol`	Get the solution value of a variable.	p. 96
`XPRMselectprob`	Select the active problem.	p. 104

XPRMgetprobstat

Purpose

Get the problem status of a model.

Synopsis

```
int XPRMgetprobstat(XPRMmodel model);
```

Argument

`model` Reference to a model

Return value

Problem status or `-1` if the model has not been run.

Further information

This function returns the status of the active problem of the given model, or 0 if no problem is available.

The problem status is bit encoded as follows:

`XPRM_PBCHG` Problem loaded in the optimizer (if any) is not valid

`XPRM_PBSOL` A solution is available

The solution status can be obtained by checking the `XPRM_PBRES` bits of the problem status. Possible values are:

`XPRM_PBOPT` optimal solution found

`XPRM_PBUNF` optimization unfinished

`XPRM_PBINF` problem is infeasible

`XPRM_PBUNB` problem is unbounded

`XPRM_PBOTH` optimization failed (any other cause)

Related topics

`XPRMgetobjval`.

XPRMexportprob

Purpose

Export the active problem to a file.

Synopsis

```
int XPRMexportprob(XPRMmodel model, const char *options,
      const char *fname, XPRMlinctr obj);
```

Arguments

`model` Reference to a model

`options` Format of the output. Possible value are:

`""`	LP output format, minimization (default)
`"m"`	MPS output format
`"p"`	Maximization (default is minimization)
`"s"`	Use scrambled names
`"x"`	Hexadecimal numbers for MPS

`fname` File name, may be `NULL`

`obj` Objective to use for optimization, or `NULL` (no objective) or `XPRM_KEEPOBJ` (last objective used)

Return value

0 if executed successfully, `XPRM_RT_ERROR` if no problem is available or `XPRM_RT_IOERR` in case of IO error.

Further information

This function exports the active problem to an MPS or LP format matrix file. If the filename is set to `NULL`, the output is printed to the console. If the filename is given without an extension, the extension `.mps` for MPS files or `.lp` for LP format files is added. The output format options can be combined in a single string (*e.g.* `"sp"`). This function is disabled (*i.e.* it succeeds but performs no operation) when Mosel is running in trial mode.

When exporting matrices in MPS format any possibly specified lower bounds on semi-continuous or semi-continuous integer variables are lost. LP format matrices maintain the complete information.

XPRMgetobjval

Purpose

Get the objective function value.

Synopsis

```
double XPRMgetobjval(XPRMmodel model);
```

Argument

`model` Reference to a model

Return value

Objective function value.

Further information

This function returns the value of the objective function if the problem has been solved successfully.

Related topics

`XPRMgetprobstat`.

XPRMgetvsol

Purpose

Get the solution value of a variable.

Synopsis

```
double XPRMgetvsol(XPRMmodel model, XPRMmpvar var);
```

Arguments

`model` Reference to a model

`var` Reference to a decision variable

Return value

Solution value or `0`.

Further information

This function returns the value of a given variable if the problem has been solved successfully (LP: optimal LP solution or 0, global: last integer solution or 0).

Related topics

`XPRMgetrcost`.

XPRMgetcsol

Purpose

Get the solution value of a linear constraint.

Synopsis

```
double XPRMgetcsol(XPRMmodel model, XPRMlinctr ctr);
```

Arguments

`model` Reference to a model

`ctr` Reference to a linear constraint

Return value

Solution value.

Further information

This function returns the evaluation of the given constraint using the current solution (this corresponds to the Mosel `getsol` function applied to a linear constraint).

Related topics

`XPRMgetdual`, `XPRMgetslack`.

XPRMgetrcost

Purpose

Get the reduced cost value of a variable.

Synopsis

```
double XPRMgetrcost(XPRMmodel model, XPRMmpvar var);
```

Arguments

`model` Reference to a model

`var` Reference to a decision variable

Return value

Reduced cost value or `0`.

Further information

This function returns the reduced cost value of a given variable if the problem has been solved successfully (otherwise 0).

Related topics

`XPRMgetvsol`.

XPRMgetdual

Purpose

Get the dual value of a linear constraint.

Synopsis

```
double XPRMgetdual(XPRMmodel model, XPRMlinctr ctr);
```

Arguments

`model` Reference to a model

`ctr` Reference to a linear constraint

Return value

Dual value or `0`.

Further information

This function returns the dual value of a given linear constraint if the problem has been solved successfully and the constraint is contained in the problem (otherwise 0).

Related topics

`XPRMgetact`, `XPRMgetcsol`, `XPRMgetslack`.

XPRMgetslack

Purpose

Get the slack value of a linear constraint.

Synopsis

```
double XPRMgetslack(XPRMmodel model, XPRMlinctr ctr);
```

Arguments

`model` Reference to a model

`ctr` Reference to a linear constraint

Return value

Slack value or `0`.

Further information

This function returns the slack value of a given linear constraint if the problem has been solved successfully (otherwise 0).

Related topics

`XPRMgetcsol`, `XPRMgetdual`.

XPRMgetact

Purpose

Get the activity value of a linear constraint.

Synopsis

```
double XPRMgetact(XPRMmodel model, XPRMlinctr ctr);
```

Arguments

`model` Reference to a model

`ctr` Reference to a linear constraint

Return value

Activity value.

Further information

This function returns the activity value of a given linear constraint if the problem has been solved successfully.

Related topics

`XPRMgetcsol`, `XPRMgetslack`.

XPRMgetvarnum

Purpose

Get the column number of a decision variable.

Synopsis

```
int XPRMgetvarnum(XPRMmodel model, XPRMmpvar var);
```

Arguments

`model` Reference to a model

`var` Reference to a variable

Return value

The column number ($\geqslant 0$) of the decision variable, or a negative value.

Further information

This function returns the column number of a decision variable. A negative value is returned if no problem is available or if the variable does not belong to the active problem.

Related topics

`XPRMgetctrnum`.

XPRMgetctrnum

Purpose

Get the row number of a linear constraint.

Synopsis

```
int XPRMgetctrnum(XPRMmodel model, XPRMlinctr ctr);
```

Arguments

`model` Reference to a model

`ctr` Reference to a linear constraint

Return value

The row number ($\geqslant 0$) of the linear constraint, or a negative value.

Further information

This function returns the row number of a linear constraint. A negative value is returned if no problem is available or if the constraint does not belong to the active problem.

Related topics

`XPRMgetvarnum`.

XPRMselectprob

Purpose

Select the active problem.

Synopsis

```
int XPRMselectprob(XPRMmodel model, int typcode, void *prob);
```

Arguments

`model`	Reference to a model
`typcode`	Type code of the object problem (`0` to activate the main problem)
`prob`	Reference to the object problem (ignored if `typcode` is `0`)

Return value

`0` if successful, `1` otherwise.

Further information

1. This function *activates* a problem: after a successful call, all functions accessing problem information will refer to the selected problem. The function will fail if the requested problem has not been initialised or is empty (for instance before execution of the model).

2. This function cannot be used while the model is running, in particular it will fail if called from the debugger interface (see Section 1.3).

Related topics

`XPRMfindident`.

1.2.7 Miscellaneous

`XPRMcb_sendctrl`	Send a control character to an initialization stream.	p. 114
`XPRMcb_sendint`	Send an integer value to an initialization stream.	p. 111
`XPRMcb_sendreal`	Send a real value to an initialization stream.	p. 112
`XPRMcb_sendstring`	Send a text string value to an initialization stream.	p. 113
`XPRMdate2jdn`	Convert a date into a Julian Day Number (JDN).	p. 107
`XPRMfreememblk`	Release a memory block allocated by the `mem:` IO driver.	p. 106
`XPRMjdn2date`	Convert a Julian Day Number (JDN) into a calendar date.	p. 108
`XPRMpathcheck`	Expand a path name and check whether it can be accessed.	p. 110
`XPRMtime`	Get the current date and time.	p. 109

XPRMfreememblk

Purpose

Release a memory block allocated by the `mem:` IO driver.

Synopsis

```
int XPRMfreememblk(XPRMmemblk *memblk);
```

Argument

`memblk` Reference to a memory block

Return value

`0` if successful, `1` otherwise.

Further information

This function can be used to release a memory block allocated by the `"mem:"` IO driver used with an `XPRMmemblk` object (see Section 1.5.3)

XPRMdate2jdn

Purpose

Convert a date into a Julian Day Number (JDN).

Synopsis

```
int XPRMdate2jdn(int year,int month, int day);
```

Arguments

`year`	Year number
`month`	Month number (1-12)
`day`	Day number (1-31)

Return value

The JDN corresponding to the provided date.

Further information

The value returned by this function corresponds to the number of days elapsed since 1/1/1970.

Related topics

`XPRMjdn2date`,`XPRMtime`.

XPRMjdn2date

Purpose

Convert a Julian Day Number (JDN) into a calendar date.

Synopsis

```
void XPRMjdn2date(int jdn, int *year,int *month, int *day);
```

Arguments

`jdn`	The Julian Day Number to decode
`year`	Returned year number
`month`	Returned month number (1-12)
`day`	Returned day number (1-31)

Further information

This function decodes a date represented using a JDN as returned by the functions `XPRMdate2jdn` or `XPRMtime`.

Related topics

`XPRMdate2jdn`,`XPRMtime`.

XPRMtime

Purpose

Get the current date and time.

Synopsis

```
void XPRMtime(int *jdn, int *t,int *tz);
```

Arguments

`jdn`	Returned Julian Day Number
`t`	Returned current time (in milliseconds)
`tz`	Time zone. Possible values are: `XPRM_TIME_LOCAL` Time is expressed in local time `XPRM_TIME_UTC` Time is expressed in Coordinated Universal Time (UTC)

Return value

`0` if successful, `1` otherwise.

Further information

1. This function returns the current date as a JDN (number of days since 1/1/1970) and a number of milliseconds since midnight. The JDN may be decoded using the function `XPRMjdn2date`.

2. The date returned by this function can be converted to a Unix time (type `time_t`) using the expression: `jdn*86400+t/1000`. Similarly a Windows file time (type `FILETIME`) can be obtained using: `((__int64)jdn+134774)*864000000000i64+(__int64)t*10000i64)`.

Related topics

`XPRMjdn2date`,`XPRMdate2jdn`.

XPRMpathcheck

Purpose

Expand a path name and check whether it can be accessed.

Synopsis

```
int XPRMpathcheck(const char *path,char *fullpath,int maxlen,int acc);
```

Arguments

`path`	Path (or file name) to be processed
`fullpath`	Buffer to return the expanded path
`maxlen`	Size of `fullpath`
`acc`	Operation to perform. Possible values: `XPRM_RCHK_READ` Check whether path or file can be read `XPRM_RCHK_WRITE` Check whether path or file can be written `XPRM_RCHK_NOCHK` Only expand the path without testing access

Return value

0 if successful, 1 if access is denied and a negative value in case of error (*e.g.* buffer too small).

Further information

1. This routine returns an absolute path to the file name it gets as input and optionally checks whether access is allowed according to the current restrictions.
2. The string `"tmp:"` beginning a path name will be replaced by the path to the Mosel temporary directory.

Related topics

`XPRMsetdefworkdir`, `XPRMsetrestrictions`.

XPRMcb_sendint

Purpose

Send an integer value to an initialization stream.

Synopsis

```
int XPRMcb_sendint(XPRMcbinit cbinit,int i,int flush);
```

Arguments

`cbinit`	Stream context
`i`	Integer value to send
`flush`	If `0`, the data will be processed when the queue of tokens is full, otherwise processing occurs immediately.

Return value

`0`	Normal termination
`1`	All tokens have been processed and no more data is expected
`2`	No more data is expected but the queue of tokens is not empty
`-1`	An error occurred

Further information

1. A boolean value can also be sent using this routine: `1` for `true` and `0` for false.
2. This function can only be used from a callback stream (see Section 1.5.2) during the execution of an `initializations from` block.
3. If the `flush` argument is `0`, the information sent is queued and the function returns `0`. The queue is flushed when it is full or if the `flush` argument is non-zero.

Related topics

`XPRMcb_sendreal`,`XPRMcb_sendstring`,`XPRMcb_sendctrl`.

XPRMcb_sendreal

Purpose

Send a real value to an initialization stream.

Synopsis

```
int XPRMcb_sendreal(XPRMcbinit cbinit,double r,int flush);
```

Arguments

`cbinit`	Stream context
`r`	Real value to send
`flush`	If `0`, the data will be processed when the queue of tokens is full, otherwise processing occurs immediately.

Return value

`0`	Normal termination
`1`	All tokens have been processed and no more data is expected
`2`	No more data is expected but the queue of tokens is not empty
`-1`	An error occurred

Further information

1. This function can only be used from a callback stream (see Section 1.5.2) during the execution of an `initializations from` block.
2. If the `flush` argument is `0`, the information sent is queued and the function returns `0`. The queue is flushed when it is full or if the `flush` argument is non-zero.

Related topics

`XPRMcb_sendint`,`XPRMcb_sendstring`,`XPRMcb_sendctrl`.

XPRMcb_sendstring

Purpose

Send a text string value to an initialization stream.

Synopsis

```
int XPRMcb_sendstring(XPRMcbinit cbinit,const char *text, int len,int
      flush);
```

Arguments

`cbinit`	Stream context
`text`	Text string value to send
`len`	Length of the string (or `-1`)
`flush`	If `0`, the data will be processed when the queue of tokens is full, otherwise processing occurs immediately.

Return value

`0`	Normal termination
`1`	All tokens have been processed and no more data is expected
`2`	No more data is expected but the queue of tokens is not empty
`-1`	An error occurred

Further information

1. This function can only be used from a callback stream (see Section 1.5.2) during the execution of an `initializations from` block.
2. If the `flush` argument is `0`, the information sent is queued and the function returns `0`. The queue is flushed when it is full or if the `flush` argument is non-zero.

Related topics

`XPRMcb_sendint`,`XPRMcb_sendreal`,`XPRMcb_sendctrl`.

XPRMcb_sendctrl

Purpose

Send a control character to an initialization stream.

Synopsis

```
int XPRMcb_sendctrl(XPRMcbinit cbinit,int ctrl,int flush);
```

Arguments

`cbinit` Stream context

`ctrl` Control character code. Possible values:

`XPRM_CBC_SKIP`	Skip the next record (same as `*` in an ASCII file)
`XPRM_CBC_OPENLST`	Begin a list of values (same as `[` in an ASCII file)
`XPRM_CBC_CLOSELST`	End a list of values (same as `]` in an ASCII file)
`XPRM_CBC_OPENNDX`	Begin a list of indices (same as `(` in an ASCII file)
`XPRM_CBC_CLOSENDX`	End a list of indices (same as `)` in an ASCII file)

`flush` If `0`, the data will be processed when the queue of tokens is full, otherwise processing occurs immediately.

Return value

`0`	Normal termination
`1`	All tokens have been processed and no more data is expected
`2`	No more data is expected but the queue of tokens is not empty
`-1`	An error occurred

Further information

1. This function can only be used from a callback stream (see Section 1.5.2) during the execution of an `initializations from` block.
2. If the `flush` argument is `0`, the information sent is queued and the function returns `0`. The queue is flushed when it is full or if the `flush` argument is non-zero.

Related topics

`XPRMcb_sendint`,`XPRMcb_sendreal`,`XPRMcb_sendstring`.

1.3 Debugger interface

The Mosel debugger interface provides the necessary functionality for controling the execution of a program (execution step by step, breakpoints, access to local symbols, stack frame change) that may be used, for instance, to implement an interactive debugger. This interface relies on debugging information stored in the BIM file which is generated at compile time depending on compilation options (see Section 2.1):

- correspondence between a global symbol and its value: this information is available as long as the source is not compiled with option `"s"`;
- correspondence between a local symbol (*e.g.* index of a loop or variable local to a function) and its value: this information is generated when model is compiled with option `"g"`;
- correspondence between source code and compiled code: the *source location* information is also constructed if option `"g"` was used for compilation;
- tracing facility to enable the Mosel virtual machine to suspend execution at a specified location (*breakpoint*) or execute one statement at a time: as opposed to the previous features, this information requires insertion of instructions in the compiled code (and may alter the execution speed of a model). To enable this extension, option `"G"` has to be used when compiling the source model.

A model to be run through the debugger interface should be compiled with flag `"g"` or `"G"`.

For the functions described below, the source location is indicated by means of *line indices*: each of these indices is associated to a statement, a data structure declaration or an end of subroutine (just before it returns). The function `XPRMdbg_getlocation` makes the correspondence between a line index and an actual source location (*i.e.* file name and line number). The first statement of the program has always index 0 and the total number of indices can be obtained using `XPRMdbg_getnblndx`. It is also possible to retrieve all indices at once using `XPRMdbg_getlndx`. The indices of the first and last statements of a function are returned by `XPRMdbg_findprocblklndx`.

The execution of a program normally terminates when an error occurs or simply when all instructions have been run. Using the function `XPRMdbg_setbrkp`, it is possible to specify locations in the program where execution must be suspended. From these *breakpoints*, one can examine current value of variables, install new breakpoints then continue or cancel execution for instance.

Before procedures (or functions) are called during execution of a program, the execution context of the system (mainly local symbols and a reference to the next instruction) is saved on top of a stack. This way, after the routine returns, the state of the machine can be restored and the execution resumed. When the execution of the program is suspended, it may be interesting to change the current position in the stack, or *stack frame*, in order to view variables that are not defined at the current level because they are declared by the calling procedure. This can be achieved using function `XPRMdbg_setstacklev`.

In order to use the debugger interface, the program has to be run with the function `XPRMdbg_runmod`: this special version of `XPRMrunmod` requires an extra parameter specifying a function reference, Mosel calls this function whenever the program has to be interrupted. If there is no error condition, the return value of the function decides whether execution should continue or not. During the interruption, most functions listed in this manual can be used to retrieve information about the current state of the program. Moreover, `XPRMfindident` returns references to locally defined symbols when called from the debugger interface.

`XPRMdbg_clearbrkp`	Clear a breakpoint at the given line index.	p. 125
`XPRMdbg_findprocblklndx`	Find the line index of a procedure or function.	p. 123
`XPRMdbg_getlndx`	Retrieve all line indices.	p. 120

`XPRMdbg_getlocation`	Get a source file location associated to a given line index.	p. 122
`XPRMdbg_getnblndx`	Get the number of line indices.	p. 121
`XPRMdbg_getnextlocal`	Get the next local identifier in the dictionary.	p. 119
`XPRMdbg_runmod`	Run a model through the debugger interface.	p. 117
`XPRMdbg_setbrkp`	Set a breakpoint at the given line index.	p. 124
`XPRMdbg_setstacklev`	Set the current stack frame to the specified level.	p. 126

XPRMdbg_runmod

Purpose

Run a model through the debugger interface.

Synopsis

```
int XPRMdbg_runmod(XPRMmodel model, int *exitcode, const char *parlist,
      int (MM_RTC *dbgcb)(void *dctx, int vmstat, int lndx), void *dbgctx);
```

Arguments

`model`	Reference to a model
`exitcode`	Pointer to an area where the model exit code value is returned
`parlist`	String composed of model parameter initializations separated by commas, may be `NULL`
`dbgcb`	user defined debugger callback
`dbgctx`	debug context: it is used as the first argument of `dbgcb`

Return value

Execution status. Possible values are:

`XPRM_RT_OK`	Normal termination
`XPRM_RT_ERROR`	An error occured during execution
`XPRM_RT_MATHERR`	Mathematical error (*e.g.* division by zero)
`XPRM_RT_IOERR`	Input/output error (*e.g.* cannot open file)
`XPRM_RT_STOP`	Bit set if execution has been interrupted

Further information

1. The parameter `parlist` may be used to initialize the model parameters of the model/program (*e.g.* `"PAR1=12,PAR2='tutu'"`). The bit `XPRM_RT_STOP` is set if the execution of the model has been interrupted by a call to the function `XPRMstoprunmod`.

2. The parameter `exitcode` receives the result of the execution (*i.e.* parameter value of the "exit" procedure or `0` if the routine was not called), its value is meaningful only when the execution succeeded (*i.e.* the status is `XPRM_RT_OK`).

3. If the function pointer `dbgcb` if `NULL` `XPRMdbg_runmod` behaves like `XPRMisrunmod`; otherwise function `dbgcb` is called whenever the model is interrupted (breakpoint, error or function `XPRMstoprunmod` called). The first argument, `dctx`, is the value of `dbgctx`; the second, `vmstat`, is the virtual machine status (*i.e.* `XPRM_RT_*`) and the last argument, `lndx`, is the line index corresponding to the statement being executed (asynchronous interruption) or to be executed (breakpoint). In this context the virtual machine status may take value `XPRM_RT_BREAK` if interruption is due to a breakpoint and value `XPRM_RT_NIFCT` if the program was executing a native function when interruption occurred.

4. If a breakpoint at line `-1` has been set (see `XPRMdbg_setbrkp`) the debugger callback `dbgcb` is called just before program termination when no error has occurred. In this case the virtual machine status takes value `XPRM_RT_ENDING` for a normal termination and value `XPRM_RT_EXIT` if the last statement is a call to procedure `exit`.

5. If the program is interrupted because of an error, the return value of `dbgcb` is ignored, otherwise it indicates how to continue execution. If `vmstat` is not `XPRM_RT_NIFCT`, the following values can be returned:

`XPRM_DBG_STOP`	terminate execution
`XPRM_DBG_NEXT`	stop before the next statement skipping function calls
`XPRM_DBG_STEP`	stop before the next statement stepping into function calls
`XPRM_DBG_CONT`	continue execution
`j>0`	stop before the statement at line index `j`

6. If the interruption occurs during the execution of a native function (for instance when the optimizer is solving a problem), `vmstat` is `XPRM_RT_NIFCT` and execution of the function can be canceled (execution continues after the NI call) by returning `XPRM_DBG_STOP` (in this case the debugger callback is called again just after the native function call completes). Other values returned by `dbgcb` imply the continuation of the execution.

Related topics

`XPRMrunmod`, `XPRMisrunmod`, `XPRMstoprunmod`.

XPRMdbg_getnextlocal

Purpose

Get the next local identifier in the dictionary.

Synopsis

```
const char *XPRMdbg_getnextlocal(XPRMmodel model, void **ref);
```

Arguments

`model` Reference to a model

`ref` Pointer to an area where current location is stored

Return value

An identifier of the symbol table or `NULL` if all local identifiers have been returned.

Further information

This function is used in the same way as `XPRMgetnextident` except that it returns the identifiers locally defined in the current context. These symbols are not reported by `XPRMgetnextident`.

Related topics

`XPRMgetnextident`, `XPRMfindident`.

XPRMdbg_getlndx

Purpose

Retrieve all line indices.

Synopsis

```
void XPRMdbg_getlndx(XPRMmodel model, int *nbl, int *lines, int *nbf, const
       char **fnames);
```

Arguments

`model`	Reference to a model
`nbl`	Pointer to an area where the number of indices is returned (can be `NULL`)
`lines`	Pointer to an area where the encoded line numbers are returned (can be `NULL`)
`nbf`	Pointer to an area where the number of file names is returned (can be `NULL`)
`fnames`	Pointer to an area where the file names are returned (can be `NULL`)

Further information

1. If no debugging information is available (*e.g.* program has not been compiled with option `"G"`) `nbl` takes value `-1` and `nbf` is set to `0`.
2. `lines` and `fnames` can be `NULL`: in this case the function still returns the sizes of these arrays.
3. Each entry of the `lines` array represents at the same time a line number (coded on 24 bits) and a file number (coded on 8bits) that is a reference in the `fnames` array. Macros `XPRM_DBGL_LINE` and `XPRM_DBGL_FILE` can be used to extract these references from the encoded index.

Related topics

`XPRMdbg_getnblndx`, `XPRMdbg_findprocblklndx`.

XPRMdbg_getnblndx

Purpose

Get the number of line indices.

Synopsis

```
int XPRMdbg_getnblndx(XPRMmodel model);
```

Argument

`model` Reference to a model

Return value

Number of line indices or -1.

Further information

1. When a program is compiled with option `"g"` or `"G"`, each statement in the source code is associated with a line index in the BIM file. This function returns the total number of line indices stored: a line index ranges between 0 and `XPRMdbg_getnblndx()-1`.

2. If no debugging information is included in the BIM file, this function return -1.

Related topics

`XPRMdbg_getlndx`,`XPRMdbg_getlocation`, `XPRMdbg_findprocblklndx`.

XPRMdbg_getlocation

Purpose

Get a source file location associated to a given line index.

Synopsis

```
int XPRMdbg_getlocation(XPRMmodel model, int lndx, int *line,
      const char **fname);
```

Arguments

`model` Reference to a model

`lndx` Line index, -1 for current location or -2 for the last location of the model

`line` Pointer to an area where the line number is returned

`fname` Pointer to an area where the file name is returned

Return value

0 if successful, 1 otherwise (invalid parameters)

Further information

1. This funtion returns the source location (file name and line number) corresponding to a given line index. If the provided index is -1 and an execution context is available, the function returns information related to the statement being executed. If this value is -2, the location of the last statement is returned.

2. If parameter `fname` is `NULL`, the function returns in `line` the current line index (*i.e.* the value of `lndx` or its updated value if it was given as a negative number).

3. If the returned line number is `0`, the machine is currently executing a portion of the code for which there is no debugging information (*i.e.* a package compiled without option `-g` or `-G`). In this case the `fname` information corresponds to the package name.

Related topics

`XPRMdbg_getnblndx`, `XPRMdbg_getlndx`, `XPRMdbg_findprocblklndx`.

XPRMdbg_findprocblklndx

Purpose

Find the line index of a procedure or function.

Synopsis

```
int XPRMdbg_findprocblklndx(XPRMmodel model, XPRMproc proc, int *elndx);
```

Arguments

`model` Reference to a model

`proc` Reference to a procedure or function

`elndx` Pointer to an area where the index of the last statement of the routine is returned

Return value

Line index of the first statement of the routine, 0 if `proc` is `NULL` or -1 in case of error.

Further information

1. This funtion returns the line index corresponding to the first statement of the provided procedure or function (as returned by `XPRMfindident`).
2. The index of the last statement can be retrieved only if the bim file includes tracing information. A value of `-1` is set to `elndx` if the information is not available.

Related topics

`XPRMdbg_getnblndx`, `XPRMdbg_getlndx`, `XPRMdbg_getlocation`.

XPRMdbg_setbrkp

Purpose

Set a breakpoint at the given line index.

Synopsis

```
int XPRMdbg_setbrkp(XPRMmodel model, int lndx);
```

Arguments

`model` Reference to a model

`lndx` Line index

Return value

0 if successful, 1 otherwise (invalid parameters)

Further information

1. After a breakpoint has been established, execution of the program is interrupted just before the specified location. A breakpoint remains active as long as it is not removed.
2. Breakpoints can be set before execution of the program but are automatically deleted after the execution terminates. A breakpoint may be explicitly removed by calling the function `XPRMdbg_clearbrkp`.
3. The line index value `-1` is a special code to establish a breakpoint just before the execution terminates (when there is no error). The debugger function is therefore called on the last statement of the program or at a call to the `exit` function.

Related topics

`XPRMdbg_clearbrkp`, `XPRMdbg_getnblndx`.

XPRMdbg_clearbrkp

Purpose

Clear a breakpoint at the given line index.

Synopsis

```
int XPRMdbg_clearbrkp(XPRMmodel model, int lndx);
```

Arguments

`model` Reference to a model

`lndx` Line index or -1 for all breakpoints

Return value

0 if successful, 1 otherwise (invalid parameters)

Further information

This function deletes a breakpoint previously set using `XPRMdbg_setbrkp`. If no breakpoint was installed at the given location, the function has no effect; if the line index is -1, all defined breakpoints are cleared.

Related topics

`XPRMdbg_setbrkp`, `XPRMdbg_getnblndx`.

XPRMdbg_setstacklev

Purpose

Set the current stack frame to the specified level.

Synopsis

```
int XPRMdbg_setstacklev(XPRMmodel model, int level);
```

Arguments

`model` Reference to a model

`level` Stack level

Return value

Line index or -1 if the level does not exist

Further information

This function changes the current stack frame of the program: the initial level is 0, positive values indicate higher levels. The line index returned corresponds to the location of the function call or the current location if the level is 0. If no location can be found for the selected stack frame the value `INT_MAX` is returned. Changing the stack frame modifies the behaviour of `XPRMfindident` regarding local symbols: symbols returned are those of the specified stack level and not those of the interruption (level 0).

Related topics

`XPRMdbg_runmod`, `XPRMfindident`.

1.4 Handling of modules

The functionalities of Mosel may be extended by using *native libraries* or *modules* implemented as ***d****ynamic* ***s****hared* ***o****bjects* (DSO). The module manager of Mosel keeps a list of all loaded modules and maintains a list of references for each of them. Using the following functions it is possible to know which modules are currently loaded and what are the provided features, and to access the values of their control parameters.

`XPRMautounloaddso`	Disable/enable automatic unloading of modules.	p. 131
`XPRMfinddso`	Find a DSO descriptor from a module name.	p. 132
`XPRMflushdso`	Unload unused dynamic shared objects.	p. 133
`XPRMgetdsoannotations`	Retrieve annotations defined by a module.	p. 134
`XPRMgetdsoparam`	Get the current value of a control parameter.	p. 135
`XPRMgetdsopath`	Get the directory list where DSO files are searched for.	p. 129
`XPRMgetdsoprop`	Get a property of a dynamic shared object.	p. 142
`XPRMgetnextdso`	Get next dynamic shared object.	p. 136
`XPRMgetnextdsoconst`	Enumerate constants of a module.	p. 137
`XPRMgetnextdsodep`	Enumerate dependencies of a module.	p. 138
`XPRMgetnextdsoparam`	Enumerate control parameters of a module.	p. 140
`XPRMgetnextdsoproc`	Enumerate procedures and functions of a module.	p. 141
`XPRMgetnextdsotype`	Enumerate native types of a module.	p. 139
`XPRMgetnextiodrv`	Get the next IO driver in the list of available drivers.	p. 143
`XPRMpreloaddso`	Explicitly load the named module.	p. 144
`XPRMregstatdso`	Declare a module as static.	p. 130
`XPRMsetdsopath`	Set the directory list where DSO files are stored.	p. 128

XPRMsetdsopath

Purpose

Set the directory list where DSO files are stored.

Synopsis

```
void XPRMsetdsopath(const char *paths);
```

Argument

`paths` List of directories

Further information

By default, Mosel looks for its modules in the directories defined by the environment variable `MOSEL_DSO` and then in a path deduced from the location (*rtdir*) of the Mosel runtime library:

`"rtdir\..\dso"` Under Windows if *rtdir* terminates by `"\bin"` and `"rtdir\..\dso"` exists or

`"rtdir/../dso"` On Unix compatible systems if *rtdir* terminates by `"/lib"` and `"rtdir/../dso"` exists or

`"rtdir/dso"` if this directory exists or

`"rtdir"` if none of the above rules apply

This function may be used to replace the directory list defined by `MOSEL_DSO`. Note that the directory separator is ':' under Unix (for example, `"/opt/Mosel/dso:/tmp"`) and ';' under Win32 (for example, `"E:\Mosel\Dso;C:\Temp"`).

Related topics

`XPRMgetlibpath`, `XPRMsetbimprefix`, `XPRMgetdsopath`.

XPRMgetdsopath

Purpose

Get the directory list where DSO files are searched for.

Synopsis

```
int XPRMgetdsopath(char *path, int len);
```

Arguments

`path` Array of chars where the path is returned

`len` The size of the array `path`

Return value

0 if successful, 1 if path is truncated, -1 in case of error.

Further information

This function returns the path currently used by Mosel for searching modules. Note that the returned path includes both the default search path (deduced from the location of the Mosel runtime library) and the path set up either via the environment variable `MOSEL_DSO` or the function `XPRMsetdsopath`.

Related topics

`XPRMsetdsopath`.

XPRMregstatdso

Purpose

Declare a module as static.

Synopsis

```
int XPRMregstatdso(const char *name, int (*dsoinit)(XPRMnifct, int *,
      int *, XPRMdsointer **));
```

Arguments

`name`	Name of the module
`dsoinit`	Address of the module initialization function

Return value

`0`	Registration is successful
`1`	The module was already registered as a static module
`2`	Module initialisation function failed
`3`	Module not compatible with the running version of Mosel
`4`	Loaded library is not a Mosel module

Further information

This function declares a module as static. If parameter `dsoinit` is `NULL`, the module is loaded and will not be unloaded until the termination of the program. Otherwise the module is implemented in the current program (instead of being an external library) and `dsoinit` is the initialization function of the module (see Mosel Native Interface Reference Manual).

XPRMautounloaddso

Purpose

Disable or enable automatic unloading of dynamic shared objects.

Synopsis

```
void XPRMautounloaddso(int yesno);
```

Argument

`yesno` Disable if 0, enable otherwise

Further information

Modules are loaded by the system whenever they are required. By default, each unused module is automatically unloaded after a fixed period of time. Using this function it is possible to disable this automatic unloading; in which case, unused modules have to be unloaded explicitly using `XPRMflushdso`.

Related topics

`XPRMflushdso`.

XPRMfinddso

Purpose

Find a DSO descriptor from a module name.

Synopsis

```
XPRMdsolib XPRMfinddso(const char *libname);
```

Argument

`libname` Name of the module to find

Return value

A reference to a DSO descriptor or `NULL` if the requested module has not been loaded.

Further information

This function returns the DSO pointer of a module that has been loaded previously.

Related topics

`XPRMgetnextdso`.

XPRMflushdso

Purpose

Unload unused dynamic shared objects.

Synopsis

```
void XPRMflushdso(void);
```

Further information

Each unused module is automatically unloaded after a fixed period of time. This function forces the manager to unload all unused modules.

Related topics

`XPRMautounloaddso`.

XPRMgetdsoannotations

Purpose

Retrieve annotations defined by a module.

Synopsis

```
int XPRMgetdsoannotations(XPRMdsolib dso, const char *prefix, const char
      **ann, int maxann);
```

Arguments

`dso`	Reference to a dynamic shared object loaded by Mosel
`prefix`	Filtering prefix
`ann`	Array of size `maxann` where to store the annotations (can be `NULL`)
`maxann`	Size of `ann` (to get up to `maxann/2` annotations)

Return value

Size of the array required to get all annotations (two times the number of found annotations).

Further information

1. This function retrieves the annotations defined by a module using a prefix as a filter (*e.g.* use `"doc."` to get all the documentation annotations). The result is stored in the provided array: each annotation occupies 2 entries in the array (the first one for the name of the annotation and the following one for its value).

2. The returned value may exceed `maxann` (but no more than `maxann` entries are recorded in the array). To get the required size for `ann` the function may be called with a `NULL` array.

Related topics

`XPRMgetannotations`.

XPRMgetdsoparam

Purpose

Get the current value of a control parameter.

Synopsis

```
int XPRMgetdsoparam(XPRMmodel model, XPRMdsolib dso, const char *name,
      int *type, XPRMalltypes *value);
```

Arguments

`model` Reference to a model

`dso` Reference to a dynamic shared object loaded by Mosel or `NULL`

`name` Name of the control parameter (lower case only)

`type` Returned type of the control parameter

`value` Returned value of the control parameter

Return value

0 if successful, 1 otherwise.

Further information

1. This function returns the current value of a control parameter of the given module in the context of the given model. This function requires that the model has been executed and uses the requested module.
2. If the argument `dso` is `NULL`, the function looks for the value of a Mosel parameter (like `"realfmt"`).
3. If both arguments `dso` and `model` are `NULL`, the function looks for the default value of a Mosel parameter (like `"realfmt"`).
4. The type can be decoded using the macro `XPRM_TYP`. Moreover, the bits `XPRM_CPAR_READ` and `XPRM_CPAR_WRITE` are set to indicate if the parameter can be read or written respectively (using `getparam` and `setparam`).

XPRMgetnextdso

Purpose

Get next dynamic shared object.

Synopsis

```
XPRMdsolib XPRMgetnextdso(XPRMdsolib dso);
```

Argument

`dso` Reference to a dynamic shared object loaded by Mosel or `NULL`

Return value

Next dynamic shared object loaded by Mosel or `NULL`.

Further information

This function returns the next module held in the list of modules loaded by Mosel. If the given module is at the end of the list, the function returns `NULL`, if the input parameter is set to `NULL`, the function returns the first module in the list.

Related topics

`XPRMfinddso`.

XPRMgetnextdsoconst

Purpose

Get the next constant in the list of constants defined by the given module.

Synopsis

```
void *XPRMgetnextdsoconst(XPRMdsolib dso, void *ref, const char **name,
      int *type, XPRMalltypes *value);
```

Arguments

`dso` Reference to a dynamic shared object loaded by Mosel

`ref` Reference pointer or `NULL`

`name` Returned name of the constant

`type` Returned type of the constant

`value` Returned value of the constant

Return value

Reference pointer for the next call to `XPRMgetnextdsoconst`.

Further information

1. This function returns the next constant defined by the given module. The second parameter is used to store the current location in the table of constants; if this parameter is `NULL`, the first constant of the table is returned. This function returns `NULL` if it is called with the reference to the last constant defined by the given module. Otherwise, the returned value can be used as the input parameter `ref` to get the following constant and so on. The returned information about type and value of the constant can be decoded in the same way as for the model identifiers (see `XPRMfindident`).

2. If parameters `ref` and `name` are `NULL` the function returns `NULL` and stores in `value` the total number of constants defined by the module.

Related topics

`XPRMgetnextdsoparam`, `XPRMgetnextdsoproc`, `XPRMgetnextdsotype`, `XPRMgetnextiodrv`.

XPRMgetnextdsodep

Purpose

Get the next required module of a module.

Synopsis

```
void *XPRMgetnextdsodep(XPRMdsolib dso, void *ref, const char **name);
```

Arguments

`dso` Reference to a dynamic shared object loaded by Mosel

`ref` Reference pointer or `NULL`

`name` Returned name of the module

Return value

Reference pointer for the next call to `XPRMgetnextdsodep`.

Further information

This function returns the next dependency of a module: module dependencies are the other modules it requires. The second parameter is used to store the current location in the table of dependencies; if this parameter is `NULL`, the first dependency of the table is returned. This function returns `NULL` if it is called with the reference to the last dependency defined by the given module. Otherwise, the returned value can be used as the input parameter `ref` to get the following dependency and so on.

XPRMgetnextdsotype

Purpose

Get the next type in the list of types defined by the given module.

Synopsis

```
void *XPRMgetnextdsotype(XPRMdsolib dso, void *ref, const char **name,
      unsigned int *props);
```

Arguments

`dso` Reference to a dynamic shared object loaded by Mosel

`ref` Reference pointer or `NULL`

`name` Returned name of the type

`props` Returned properties of the type (may be `NULL`)

Return value

Reference pointer for the next call to `XPRMgetnextdsotype`.

Further information

This function returns the name and properties of the next type defined by the given module. The type properties corresponds to the information returned by function `XPRMgettypeprop`. The second parameter is used to store the current location in the table of types; if this parameter is `NULL`, the first type of the table is returned. This function returns `NULL` if it is called with the reference to the last type defined by the given module. Otherwise, the returned value can be used as the input parameter `ref` to get the following type and so on.

Related topics

`XPRMgetnextdsoconst`, `XPRMgetnextdsoparam`, `XPRMgetnextdsoproc`, `XPRMgetnextiodrv`, `XPRMgettypeprop`.

XPRMgetnextdsoparam

Purpose

Get the next control parameter in the list of the given module.

Synopsis

```
void *XPRMgetnextdsoparam(XPRMdsolib dso, void *ref, const char **name,
      const char **desc, int *type);
```

Arguments

`dso` Reference to a dynamic shared object loaded by Mosel or `NULL`

`ref` Reference pointer or `NULL`

`name` Returned name of the control parameter

`desc` Returned description of the control parameter

`type` Returned type of the control parameter

Return value

Reference pointer for the next call to `XPRMgetnextdsoparam`.

Further information

This function returns the next control parameter of the given module. If the argument `dso` is `NULL`, the function returns Mosel control parameters. The second parameter is used to store the current location in the table of control parameters; if this parameter is `NULL`, the first control parameter of the table is returned. This function returns `NULL` if it is called with the reference to the last parameter of the given module. Otherwise, the returned value can be used as the input parameter `ref` to get the following control parameter and so on. The type can be decoded using the macro `XPRM_TYP`. Moreover, the bits `XPRM_CPAR_READ` and `XPRM_CPAR_WRITE` are set to indicate if the parameter can be read or written respectively (using `getparam` and `setparam`). The parameter `desc` is a textual description of the function of the parameter — this information is not necessarily available (that is, it may be `NULL` or an empty string). Note that not all modules implement the required functionality for enumerating control parameters.

Related topics

`XPRMgetnextdsoconst`, `XPRMgetnextdsoproc`, `XPRMgetnextdsotype` `XPRMgetnextiodrv`.

XPRMgetnextdsoproc

Purpose

Get the next subroutine in the list of the given module.

Synopsis

```
void *XPRMgetnextdsoproc(XPRMdsolib dso, void *ref, const char **name,
      const char **partyp, int *nbpar, int *type);
```

Arguments

`dso`	Reference to a dynamic shared object loaded by Mosel
`ref`	Reference pointer or `NULL`
`name`	Returned name of the routine (procedure or function)
`partyp`	Returned string describing the parameters of the routine
`nbpar`	Returned number of parameters expected by the routine
`type`	Returned type of the result of the routine

Return value

Reference pointer for the next call to `XPRMgetnextdsoproc`.

Further information

This function returns the next subroutine defined by the given module. The second parameter is used to store the current location in the table of subroutines; if this parameter is `NULL`, the first subroutine of the table is returned. This function returns `NULL` if it is called with the reference to the last subroutine defined by the given module. Otherwise, the returned value can be used as the input parameter `ref` to get the following subroutine and so on. The type and parameter string can be decoded in the same way as for the model procedures and functions (see `XPRMgetprocinfo`) except that native functions may return objects of native type. In this case, the function type is `XPRM_TYP_EXTN` and the parameter string `partyp` begins with the name of the function type followed by ':' (*e.g.* `"mytype:|mytype|"` is the signature of a function of type 'mytype' expecting an object of type 'mytype' as parameter. Note that the same subroutine name may be returned several times if a subroutine has been defined with different types of parameters (overloading).

Related topics

`XPRMgetnextdsoconst`, `XPRMgetnextdsoparam`, `XPRMgetnextdsotype`, `XPRMgetnextiodrv`.

XPRMgetdsoprop

Purpose

Get a property of a dynamic shared object.

Synopsis

```
int XPRMgetdsoprop(XPRMdsolib dso, int prop, void *value);
```

Arguments

`dso` Reference to a module loaded by Mosel

`prop` Property to retrieve. Possible values:

`XPRM_PROP_NAME`	Module name
`XPRM_PROP_ID`	Internal number of the module
`XPRM_PROP_VERSION`	Version number
`XPRM_PROP_SYSCOM`	Identity of the provider
`XPRM_PROP_NBREF`	Number of loaded models that use the module
`XPRM_PROP_PATH`	Path to the actual module file
`XPRM_PROP_PRIORITY`	Priority of the module
`XPRM_PROP_NBTYPES`	Number of types
`XPRM_PROP_COMPAT`	Smallest compatible version (0 if not available)

`value` Pointer to an area where the model property is returned

Further information

This function returns information about a given module. The type of the property (specified via the `prop` argument) decides how the argument `value` is interpreted: it is a reference to a constant string for `NAME`, `SYSCOM` and `PATH`; and a reference to an integer for the other properties. The returned version number is coded as an integer, for example, `1.2.3` is coded as `1002003`. The module is currently not in use if the property `NBREF` is `0`.

XPRMgetnextiodrv

Purpose

Get the next IO driver in the list of available drivers.

Synopsis

```
void *XPRMgetnextiodrv(void *ref, const char **name,
      const char **module, const char **info);
```

Arguments

`ref`	Reference pointer or `NULL`
`name`	Name of the driver (may be `NULL`)
`module`	Name of the module publishing the driver (may be `NULL`)
`info`	Information about the driver (may be `NULL`)

Return value

Reference pointer for the next call to `XPRMgetnextiodrv`.

Further information

This function returns the next IO driver in the table of currently available drivers. The first parameter is used to store the current location in the table; if this parameter is `NULL`, the first driver of the table is returned. This function returns `NULL` if it is called with the reference to the last driver available. Otherwise, the returned value can be used as the input parameter `ref` to get the following driver and so on.

Note that internal drivers have a `NULL` module name and the default driver has no name (*i.e.* `name` is an empty string). Information returned via `info` parameter corresponds to the string stored as the `XPRM_IOCTRL_INFO` operation for the driver. If this operation is not defined, return value is `NULL`.

Related topics

`XPRMgetnextdsoconst`, `XPRMgetnextdsoparam`, `XPRMgetnextdsoproc`, `XPRMgetnextdsotype`.

XPRMpreloaddso

Purpose

Explicitly load the named module.

Synopsis

```
XPRMdsolib XPRMpreloaddso(const char *libname);
```

Argument

`libname` Name of the module to load

Return value

A reference to a DSO descriptor if the module has been loaded successfully or `NULL`.

Further information

Mosel loads modules on demand when they are required by the models in core memory. However, it is possible to force the system to load a module using this function. If the module is already in memory, no action is performed and the corresponding DSO pointer is returned.

Related topics

`XPRMisrunmod`, `XPRMrunmod`.

1.5 Using IO drivers for data exchange

Mosel comes with a default set of IO drivers which are used as data source/destination. The selection of the driver is achieved via the file name in use: for instance file name `"myfile"` is a physical file handled by the operating system but `"mem:myfile"` is a block of memory managed by the `mem` driver. IO drivers are mainly used to interface specific data sources with Mosel (like `odbc` from the mmodbc module). In this context, each data source may require a dedicated driver that can be implemented in a user module through the Mosel NI (refer to the *Mosel NI Reference Manual* for further explanation). Drivers may also be employed to easily exchange information between the application running the Mosel Libraries and a model. In particular the predefined drivers `cb`, `mem` and `raw` are specifically designed for this purpose.

1.5.1 `sysfd` driver

Thanks to this driver, a file descriptor provided by the operating system may be used in place of a file. The general syntax of a file name for the `sysfd` driver is:

```
sysfd:OSfd
```

where `OSfd` is a numerical file descriptor (Posix) or a file handle (Windows). File descriptors are usually returned by C functions `open` or `fileno` (from a C-stream obtained with `fopen`) on Posix systems. Under Windows, file handles can be created using `CreateFile` or obtained with `_get_osfhandle` (from a C file descriptor) for instance. When a program starts, 3 files are automatically opened for input, output and errors; they are respectively associated to file numbers 0,1 and 2 (this applies to both Posix systems and Windows). Mosel uses these file decriptors as default streams.

Example:

```
XPRMsetdefstream(NULL,XPRM_F_ERROR,"sysfd:1"); /* redirect error to output stream */
```

1.5.2 `cb` driver

This driver allows using a function as a file. The general syntax of a file name for the `cb` driver is:

```
cb:funcaddr[/refval]
```

where `funcaddr` is the address of the *callback* function and the optional parameter `refval` is a pointer (both references must be expressed in hexadecimal). Depending on the type of stream to manage (*i.e.* a general stream or a for an `initializations` block) a specific function type as to be provided.

1.5.2.1 Handling of general streams

The expected function must have the following prototype:

```
long XPRM_RTC func(XPRMmodel model, void *ref, char *buf, unsigned long size);
```

Whenever data needs to be transferred, Mosel calls this function indicating the location (`buf`) and the size (`size`) of the buffer to use. The parameter `ref` is the information provided to Mosel during the opening of the file (`refval` above). The model reference may be `NULL` if the stream is used directly by Mosel (for instance for compilation). When the stream is open for writing, the return value of the function is ignored. If the corresponding output stream is open in text mode, the function is called at each end of line and the buffer can be seen as a `NULL` terminated character string (the size does not include the terminating character). When used for reading, the function should return the number of bytes actually copied into the buffer (0 means end of file).

Example:

```
long XPRM_RTC simpleout(XPRMmodel model, void *ref, char *buf,
              unsigned long size)
{
```

```
 printf("OUT: %.*s",(int)size,buf);
 return 0;
}

 ...

char fname[32];

sprintf(fname, "cb:%p", simpleout);
XPRMsetdefstream(NULL, XPRM_F_ERROR, fname);  /* redirect error str. to 'simpleout' */
 ...
```

1.5.2.2 Handling of initializations blocks

In the case of an initializations block, the expected function must be of the following form:

```
int XPRM_RTC func(XPRMcbinit cbinit, void *ref,const char *label, int type, XPRMalltypes *obj);
```

When executing an `initializations from` block, the function is called once for each label with the label to be initialized (`label`), its type (`type`) and a reference to the object (`obj`). The parameter `ref` is the information provided to Mosel during the opening of the file (`refval` above). The function must then send to Mosel the data to be used for the initialization of the object using routines `XPRMcb_sendint`, `XPRMcb_sendreal`, `XPRMcb_sendstring` and `XPRMcb_sendctrl`. The 3 first functions provide the basic type values while the last one is used to structure the data stream (*i.e.* delimit a list of array indices or a collection of values) in a similar fashion as in an ASCII initialization file.

Example:

```
int XPRM_RTC initfrom(XPRMcbinit cbinit, void *ref,const char *label,
              int type, XPRMalltypes *ref)
{
 int i;

 if(strcmp(label,"I")==0)      /* I:10 */
 {
  XPRMcb_sendint(cbinit,10,0)
 }
 else if(strcmp(label,"S")==0) /* S:[1 2 3] */
 {
  XPRMcb_sendctrl(cbinit,XPRM_CBC_OPENLST,0);
   for(i=1;i<=3;i++)
    XPRMcb_sendreal(cbinit,(double)i,0);
  XPRMcb_sendctrl(cbinit,XPRM_CBC_CLOSELST,0);
 }
 else if(strcmp(label,"A")==0) /* A:[(1) "a"] */
 {
  XPRMcb_sendctrl(cbinit,XPRM_CBC_OPENLST,0);
   XPRMcb_sendctrl(cbinit,XPRM_CBC_OPENNDX,0);
    XPRMcb_sendint(cbinit,1,0);
   XPRMcb_sendctrl(cbinit,XPRM_CBC_CLOSENDX,0);
   XPRMcb_sendstring(cbinit,"a",1,0);
  XPRMcb_sendctrl(cbinit,XPRM_CBC_CLOSELST,0);
 }
}
/* The associated Mosel code:
 declarations
  I:integer
  S:set of real
  A:array(range) of string
 end-declarations
 initializations from INITFILE
  I S A
 end-initialisations
*/
```

Similarly, when executing an `initializations to` block, the function is called once for each label

with the object reference, its type and associated label (in this case the first parameter is NULL). The user function can then inspect the object using the usual routines of the Mosel Runtime Library.

1.5.3 mem driver

With this driver, a block of memory is used as data source. Three different types of blocks are supported: named blocks can be used only from a model during its execution, are identified by a label and their allocation is dynamic. The second type uses a block of memory already allocated: it is characterized by an address and a size. With the third form the file name corresponds to a reference to a dedicated data structure to hold the properties of a memory block managed by Mosel.

The general syntax of a file name for the mem driver accessing a named block is:

```
mem:label[/minsize[/incstep]]
```

where label is an identifier whose first character is a letter and minsize an optional initial amount of memory to be reserved (size is expressed in bytes, in kilobytes with suffix "k" or in megabytes with suffix "m"). The memory block is allocated dynamically and resized as necessary. By default the size of the memory block is increased by pages of 4 kilobytes: the optional parameter incstep may be used to change this page size (*i.e.* the default setting is "label/0/4k"). The special value 0 modifies the allocation policy: instead of being increased of a fixed amount, the block size is doubled. In all cases unused memory is released when the file is closed.

When a named memory block is used in a model, it is possible to access the block of memory allocated by the driver by searching for the label in the model's dictionary: the function XPRMfindident returns a reference to an object of structure XPRM_STR_MEM that describes the location and size of the memory block.

The general syntax of a file name for the mem driver accessing a fixed block is:

```
mem:addr/size[/actualsize]
```

where addr and size identify the memory block (the pointer must be expressed in hexadecimal). Optionally a pointer to a size_t value may be provided (actualsize expressed in hexadecimal): when the stream is open for writing, this variable receives the size actually used by the operation (its value thus ranges between 0 and size). Moreover, if the stream is open in append mode, writing starts after the location indicated by this value. When the stream is open for reading, the value is used in place of size if it is smaller than this upper limit.

Example:

```
char blk[2048];
char fname[40];
size_t actualsize;

sprintf(fname, "mem:%p/%u/%p", blk, (int)sizeof(blk), &actualsize);
XPRMcompmod(NULL, "mymodel", fname, NULL);   /* compile model to memory */
printf("BIM data uses %u bytes.\n", actualsize);
mod=XPRMloadmod(fname, NULL);                /* load BIM file from memory */
```

The last form is similar to first one except that the name of the block is replaced by an address prefixed by the & symbol:

```
mem:&addr[/minsize[/incstep]]
```

where addr is a reference (expressed in hexadecimal) to an XPRMmemblk data structure:

```
typedef struct
   {
    void *ref;    /* Base address of the block */
    size_t size;  /* Size of the block */
   } XPRMmemblk;
```

When the block is used for the first time, fields of the structure must be cleared: Mosel handles (re)allocation of the memory block when writing to the corresponding memory file. The function XPRMfreememblk can be used to release the memory allocated through this IO driver (deleting the file

from the Mosel code has the same effect).

Example:

```
XPRMmemblk memblk;
char fname[40];

sprintf(fname, "mem:&%p", &memblk);
memset(&memblk,0,sizeof(XPRMmemblk));
XPRMcompmod(NULL, "mymodel", fname, NULL);   /* compile model to memory */
mod=XPRMloadmod(fname, NULL);                /* load BIM file from memory */
XPRMfreememblk(&memblk)                      /* release BIM file memory */
```

1.5.4 rawdriver

The raw driver provides an implementation of the "initializations blocks" in binary mode: instead of translating information from/to text format, data is kept in its raw representation. Typically this driver will be combined with the mem driver in order to exchange arrays of data between the model and an application through memory without translation. The general syntax of a file name for the raw driver is:

```
raw:[noindex,align,noalign,append,all,slength=#]
```

When using the raw driver as a file for an initializations block, no actual data location is provided at the beginning of the block. The driver uses each label as a file name for locating data.

Example:

```
initializations from "raw:noindex"
 t as "datafile.bin"
 r as "mem:0x1234/456"
end-initializations
```

Data transfer is achieved without conversion: 4 bytes for an integer, 8 bytes for a real, 1 byte for a Boolean, strings are of fixed size or just an address, external types are translated to strings (if "tostr" is available for the type) and anything else has the size of an address that is 4 or 8 bytes depending on the architecture. The option slength specifies the fixed length of strings, default value for this parameter is 16 (shorter strings are padded with 0 characters, longer strings are cut). The special value 0 implies that the address of the string is used.

If option append is specified, files open for writing are open in append mode.

Transfer of scalar is straightforward and sets are treated as a collection of consecutive scalars. The handling of arrays varies depending on the options: by default, each array element is preceded by its indices (for instance t(1,2) is stored or read as 1,2,t(1,2)). If option noindex is in use, only values are listed and if option all has been given, all elements of dynamic arrays are listed (by default: only existing elements).

The driver aligns data according to the processor architecture requirements assuming the starting address provided is aligned properly (for instance on Sparc processors real values [or doubles] are aligned on 8 bytes boundaries). Thanks to this property, it is safe to map data exchanged using this driver with the corresponding structure in the C language.

Example:

```
declarations
 a: array(integer,boolean) of real
end-declarations
! the above declaration can be mapped to the following C-structure:
! struct {
!    int ndx1
!    char ndx2
!    double a_ndx1_ndx2 };
! This structure uses 13 bytes with an Intel processor and 16 on a Sparc
```

This behavior may be changed by using the align and noalign options (for instance for saving binary

data to physical files, alignment is not necessary and uses more memory).

Options may be specified for each label individually: they have to be given as a list preceding the actual filename.

Example: the following model:

```
parameters
 DAT=""
 RES=""
end-parameters
declarations
 d:array(string) of real
 r:array(1..10) of real
end-declarations
initializations from "raw:"
 d as "slength=0,mem:"+DAT  ! load data from memory location defined by DAT
end-initializations
...
initializations to "raw:"
 r as "noindex,mem:"+RES  ! save results in memory location defined by RES
end-initializations
```

can be used with the following C-source:

```
char params[128];
struct { const char *ndx; double v; } d[]={{"one",10}, {"two",0.5}};
double r[10];

sprintf(params, "DAT='%p/%u', RES='%p/%u'", d, sizeof(d), r, sizeof(r));
XPRMrunmod(mod, &result, params);
```

1.5.5 `bin` driver

Like the `raw` driver, the `bin` driver provides an implementation of the “initializations blocks” in binary mode. However, thanks to a structured and architecture independent data format the `bin` driver can handle cross platform files containing all records of an initialisations block. An application can generate and decode files using this format with the help of the *bindrv library*. Refer to the documentation of this library for further explanation.

CHAPTER 2

Mosel Model Compiler Library

2.1 Compilation

The Mosel Model Compiler (`xprm_mc`) Library contains the compiler of Mosel. The main function provided performs the compilation of a source model file into the corresponding binary model (BIM) file. Note that `xprm_mc` requires the library `xprm_rt` to be present and even a program using only the `XPRMcompmod` function must initialize Mosel with the function `XPRMinit`.

Programs using the Model Compiler Library must include the header file `xprm_mc.h`.

Note that all text strings handled by functions of this library are encoded in UTF-8. It is therefore required to convert text strings to alternate encodings when exchanging data with other libraries not working with UTF-8. In particular the C library supports either wide characters (`wchar_t` type) or the default system encoding (that depends on the localisation of the system). These encoding conversions can be achieved with the help of the XPRNLS library (please refer to the XPRNLS Reference Manual for further details).

`XPRMcompmod, XPRMcompmodsec` Compile a model source file. p. 151

`XPRMexecmod` Compile, load then run a model source file. p. 153

XPRMcompmod, XPRMcompmodsec

Purpose

Compile a model source file.

Synopsis

```
int XPRMcompmod(const char *options, const char *srcfile, const char
      *dstfile, const char *userc);
int XPRMcompmodsec(const char *options, const char *srcfile, const char
      *dstfile, const char *userc, const char *passfile, const char
      *privkey, const char *kfile);
```

Arguments

`options` Compilation options (may be `NULL`, options may be separated by spaces or `'-'` symbols):

`"g"`	Include debugging information: in the case of a run time error during the execution of the model the location of the error in the source file may be indicated and private symbols are included in the BIM file (they are by default not saved)
`"G"`	Include tracing information: with this option the model can be run through the debugger for an execution step by step
`"p"`	parse only: stop after the syntax analysis of the source file, do not compile (no file generated)
`"x"`	extract messages for translation (generation of a POT file in place of a BIM file)
`"ni"`	Disable implicit declarations
`"na"`	Disable assertions in debug mode
`"wi"`	Emit a warning message each time a symbol is implicitly declared
`"wu"`	Report unused public symbols (except subroutines in packages)
`"we"`	Handle warnings like errors
`"D"`	Generate and record documentation annotations
`"bx=prefix"`	Package prefix (can be quoted with single or double quotes)
`"ix=prefix"`	Include source prefix (can be quoted with single or double quotes)
`"I"`	Enable xbim extensions
`"S"`	Sign the BIM file
`"E"`	Encrypt the BIM file
`"F"`	The argument `pass` is a file name (not the password itself)
`"V"`	Accept to load signed packages only if their signature can be verified
`"T"`	Accept to load only signed packages with a valid signature

`scrfile` Name of the source file

`dstfile` Name of the destination file (may be `NULL`)

`userc` Commentary text that will be saved as is at the beginning of the output file (may be `NULL`)

`passfile` Password or password file (for encryption with a password)

`privkey` Private key file (for BIM file signing)

`kfile` File of public keys (for encryption with public keys)

Return value

`0`	Function executed sucessfully
`1`	Parsing phase has failed (syntax error or file access error)
`2`	Error in compilation phase (a semantic error has been detected)
`3`	Error writing the output file
`4`	License error (compiler not authorized)

Further information

1. This function compiles a given model source file into a binary model file (BIM file) that is required as input to function `XPRMloadmod` for executing the model. The second form of the function will be used to generate encrypted and/or signed BIM files.

2. The source file name may contain environment variable references using the notation `${varname}` (for example, '`${XPRESSDIR}/examples/mymodel`') that are expanded to generate the actual name. If no destination file name is provided, the output file takes the same name as the source file with the extension `.bim`. Note that the empty string (*i.e.* `""`) is interpreted as the standard input for `srcfile` and as the standard output for `dstfile`.

3. The argument `kfile` is a list of public key files (*i.e.* each line of the file is a key file name): when encrypting a file, the encryption is performed for each of the listed public keys such that the BIM file can be decrypted by any of the corresponding private keys.

4. When prefixes provided via `bx` or `ix` are quoted with double quotes, backslashes are interpreted such that special characters can be included in the string. It is therefore required to double this symbol when it has to be included (*e.g.* `'bx="C:\\mydir"'`).

Related topics

`XPRMloadmod`, `XPRMrunmod`, `XPRMdbg_runmod`, `XPRMsetbimprefix`.

XPRMexecmod

Purpose

Compile, load then run a model source file.

Synopsis

```
int XPRMexecmod(const char *options, const char *srcfile, const char
      *parlist, int *returned, XPRMmodel *rtmod);
```

Arguments

`options`	Compilation options (may be `NULL`)
`scrfile`	Name of the source file
`parlist`	String composed of model parameter initializations separated by commas, may be `NULL`
`returned`	Pointer to an area where the result value is returned
`rtmod`	Pointer to an area where the model pointer is returned (may be `NULL`)

Return value

`<0`	Compilation failed
`0`	Function executed sucessfully
`>0`	An error occured during model execution

Further information

This function calls in sequence `XPRMcompmod`, `XPRMloadmod`, and then `XPRMrunmod` (no BIM file is generated). If parameter `rtmod` is not `NULL`, this pointer is initialized with the model reference. Otherwise, the model is unloaded after execution.

Related topics

`XPRMcompmod`, `XPRMloadmod`, `XPRMrunmod`, `XPRMunloadmod`.

Appendix A

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.

Index

A
activity
get, 101
annotations
get, 31, 134
array, 2, 41
check indices, 81
dynamic, 71
dynamic bounded, 71
general, 71
get dimensions, 72
get entry, 83
get first entry, 76
get first true entry, 79
get indices, 75
get last entry, 77
get next entry, 78
get next true entry, 80
get size, 74
get type, 73
indexing set, 64
logical entry, 71
storage class, 71
true entry, 71
types, 71

B
BIM, 19, 152
binary
model file, 152
binary model file, 19, 150
Boolean, 2, 41
breakpoint, 115
clearing, 125
setting, 124

C
check
array indices, 81
running model, 27
column
get number, 102
comment
user, 151
compare
indices, 82
compile
model, 151, 153
constant, 41
next, 137
constraint
get activity, 101
get dual, 99
get number, 103
get slack, 100
linear, 2, 41
control parameter, 135
next, 50, 140

D
date
convert to JDN, 107, 108
current, 109
debugging, 151
decision variable, *see* variable
dependency
next, 34, 138
dimension
get, 72
DSO, *see* dynamic shared object, 127
dual
get, 99
dynamic array, 71
dynamic bounded array, 71
dynamic library, *see* dynamic shared object
dynamic shared object, 2, 127
descriptor, 132
get next, 136
get property, 142
load, 144
next constant, 137
next control parameter, 140
next subroutine, 141
next type, 139
parameter, 135
unload, 133

E
element
get value, 67
entry
get first, 76
get first true, 79
get last, 77
get next, 78
get next true, 80
logical, 71
true, 71
execute
model, 25, 117
export
problem, 94

F
file

output, 94
find
identifier, 40
model, 37
type, 44
finish, 6
function, 41
information, 54
next, 141
next overloading, 51

G
get
activity, 101
array dimensions, 72
array entry, 83
array indices, 75
array size, 74
array type, 73
column number, 102
dual, 99
dynamic shared object, 132
dynamic shared object property, 142
element value, 67
first array entry, 76
first index, 69
first true array entry, 79
index, 68
last array entry, 77
last index, 70
list size, 60
list type, 61
model, 37
model property, 33
next array entry, 78
next constant, 137
next control parameter, 50, 140
next dynamic shared object, 136
next identifier, 47
next model, 35
next overloading, 51
next parameter, 48
next subroutine, 141
next true array entry, 80
next type, 139
objective, 95
parameter, 135
problem status, 93
reduced cost, 98
row number, 103
set size, 65
set type, 66
signature, 54
slack, 100
solution value, 96, 97
version, 17, 18

I
I/O driver, 145
bin:, 149
cb:, 145
mem:, 147
raw:, 148
sysfd:, 145
identifier
find, 40
next, 47
index
get, 68
get first, 69
get last, 70
index set, 64
indices
check, 81
compare, 82
infeasible problem, 93
initialization, 3, 4
integer, 2, 41
interrupt
model, 28

J
JDN, 107

L
library
Model Compiler, 1, 150
native, 127
Run Time, 1, 2
line index, 115, 120
get number of, 121
get source location, 122
of a procedure/function, 123
linear constraint, 2, 41
list, 2, 41, 59
get elements, 62, 63
get size, 60
get type, 61
storage class, 61
load
dynamic shared object, 144
model, 20
logical entry, 71
LP format, 94

M
main problem, 92
maximization, 94
memory block, 41
minimization, 94
model, 2
check running, 27
compile, 151, 153
debug, 117
find, 37
get next, 35
get next dso, 36
get problem status, 93
get property, 33
load, 20

reset, 24
run, 25, 117
stop, 28
unload, 30
Model Compiler Library, 1, 150
model file, 150
binary, 19, 152
model property
get, 33
module, *see* dynamic shared object, 127
Mosel
finish, 6
initialization, 4
initialize, 5
version, 17
MPS format, 94

N
names
scramble, 94
native library, 127
next
identifier, 47
local identifier, 119
overloading, 51
parameter, 48

O
objective
get value, 95
optimal solution, 93
optimization
failed, 93
unfinished, 93
output, 94
output format, 94
overloading, 51, 141

P
package
next control parameter, 50
parameter, 135
next, 48
post processing interface, 38
problem
active, 92
export, 94
get status, 93
infeasible, 93
select, 104
unbounded, 93
problem component
next, 49
procedure, 41
information, 54
next, 141
next overloading, 51

R
range, 64
range set, 64, 69, 70
real, 2, 41
record, 84
get field value, 86
get fields, 85
reduced cost
get, 98
reference, 41
requirement
next, 52
resetting model, 24
row
get number, 103
run
model, 25, 117
Run Time Library, 1, 2

S
scrambled names, 94
set, 2, 41, 64
get element value, 67
get first index, 69
get index, 68
get last index, 70
get size, 65
get type, 66
storage class, 66
signature, 54
size
get, 60, 65, 74
slack
get, 100
solution
get, 96, 97
get value, 95
optimal, 93
status, 93
source file, 151, 153
source model file, 150
stack frame, 115
changing, 126
stop
model, 28
storage class
array, 71
list, 61
set, 66
string, 2, 41
subroutine
next, 141

T
table, *see* array
terminate, 6
termination, 3, 6
tracing, 151
true entry, 71
type
array, 71
basic, 2
code, 41

find, 44
get, 61, 66, 73
get property, 55
next, 139
structure, 41
types
all, 41

U
unbounded problem, 93
unfinished
optimization, 93
union, 87
get compatible types, 88
get value, 91
type ID of value, 90
type of value, 89
unload
dynamic shared object, 133
model, 30
user comment, 151
user type, 41

V
variable, 2, 41
get number, 102
get reduced cost, 98
get solution, 96, 97

X
XPRM_GRP, 61, 66, 73
XPRM_GRP_DYN, 61, 66, 71
XPRM_GRP_GEN, 66, 71
XPRM_IS_PUBLIC(t), 41, 85
XPRM_PBRES, 93
XPRM_STR, 41, 89
XPRM_TYP, 41, 61, 66, 73, 89
XPRM_DBG_CONT, 118
XPRM_DBG_NEXT, 118
XPRM_DBG_STEP, 118
XPRM_DBG_STOP, 118
xprm_mc, 150
XPRM_MTP_ANDX, 56
XPRM_MTP_APPND, 56
XPRM_MTP_CMP, 56
XPRM_MTP_CONST, 56
XPRM_MTP_COPY, 56
XPRM_MTP_CREAT, 56
XPRM_MTP_DELET, 56
XPRM_MTP_FRSTR, 56
XPRM_MTP_NAMED, 56
XPRM_MTP_ORD, 56
XPRM_MTP_ORSET, 56
XPRM_MTP_PROB, 56
XPRM_MTP_PRTBL, 56
XPRM_MTP_RFCNT, 56
XPRM_MTP_SHARE, 56
XPRM_MTP_TFBIN, 56
XPRM_MTP_TOSTR, 56
XPRM_PBCHG, 93
XPRM_PBINF, 93
XPRM_PBOPT, 93
XPRM_PBOTH, 93
XPRM_PBSOL, 93
XPRM_PBUNB, 93
XPRM_PBUNF, 93
xprm_rt, 2
XPRM_RT_BREAK, 25, 118
XPRM_RT_ENDING, 118
XPRM_RT_ERROR, 25, 117
XPRM_RT_EXIT, 118
XPRM_RT_IOERR, 25, 117
XPRM_RT_MATHERR, 25, 117
XPRM_RT_NIFCT, 118
XPRM_RT_OK, 25, 117
XPRM_RT_STOP, 25, 117
XPRM_STR_ARR, 41
XPRM_STR_CONST, 41
XPRM_STR_LIST, 41
XPRM_STR_MEM, 41
XPRM_STR_PROB, 56
XPRM_STR_PROC, 41
XPRM_STR_REC, 56
XPRM_STR_REF, 41
XPRM_STR_SET, 41
XPRM_STR_UNION, 56
XPRM_STR_UTYP, 41
XPRM_TYP_BOOL, 41
XPRM_TYP_INT, 41
XPRM_TYP_LINCTR, 41
XPRM_TYP_MPVAR, 41
XPRM_TYP_NOT, 41
XPRM_TYP_REAL, 41
XPRM_TYP_STRING, 41
XPRMaddparamstr, 26
XPRMalltypes, 41
XPRMarray, 2
XPRMattrdesc, 2
XPRMautounloaddso, 131
XPRMboolean, 2
XPRMcb_sendctrl, 114
XPRMcb_sendint, 111
XPRMcb_sendreal, 112
XPRMcb_sendstring, 113
XPRMchkarrind, 81
XPRMcmpindices, 82
XPRMcompmod, 151
XPRMcompmodsec, 151
XPRMcsrtoref, 58
XPRMdate2jdn, 107
XPRMdbg_clearbrkp, 125
XPRMdbg_findprocblklndx, 123
XPRMdbg_getlndx, 120
XPRMdbg_getlocation, 122
XPRMdbg_getnblndx, 121
XPRMdbg_getnextlocal, 119
XPRMdbg_runmod, 117
XPRMdbg_setbrkp, 124
XPRMdbg_setstacklev, 126
XPRMdsolib, 2

XPRMdsotyptostr, 39
XPRMexecmod, 153
XPRMexportprob, 92, 94
XPRMfindattrdesc, 43
XPRMfinddso, 132
XPRMfindident, 40
XPRMfindmod, 37
XPRMfindtypecode, 44
XPRMfinish, 3, 6
XPRMflushdso, 133
XPRMfree, 6
XPRMfreelibpath, 9
XPRMfreememblk, 106
XPRMgetact, 101
XPRMgetannotations, 31
XPRMgetarrdim, 72
XPRMgetarrsets, 75
XPRMgetarrsize, 74
XPRMgetarrtype, 73
XPRMgetarrval, 83
XPRMgetattr, 45
XPRMgetbimprefix, 32
XPRMgetcsol, 97
XPRMgetctrnum, 103
XPRMgetdefworkdir, 7
XPRMgetdsoannotations, 134
XPRMgetdsoparam, 135
XPRMgetdsopath, 129
XPRMgetdsoprop, 142
XPRMgetdual, 99
XPRMgetelsetndx, 68
XPRMgetelsetval, 67
XPRMgetfieldval, 86
XPRMgetfirstarrentry, 76
XPRMgetfirstarrtruentry, 79
XPRMgetfirstsetndx, 69
XPRMgetlastarrentry, 77
XPRMgetlastsetndx, 70
XPRMgetlibpath, 8
XPRMgetlicerrmsg, 5
XPRMgetlistsize, 60
XPRMgetlisttype, 61
XPRMgetlocaledir, 15
XPRMgetmodprop, 33
XPRMgetnextanident, 46
XPRMgetnextarrentry, 78
XPRMgetnextarrtruentry, 80
XPRMgetnextattrdesc, 53
XPRMgetnextdep, 34
XPRMgetnextdso, 136
XPRMgetnextdsoconst, 137
XPRMgetnextdsodep, 138
XPRMgetnextdsoparam, 140
XPRMgetnextdsoproc, 141
XPRMgetnextdsotype, 139
XPRMgetnextfield, 85
XPRMgetnextident, 47
XPRMgetnextiodrv, 143
XPRMgetnextlistelt, 62
XPRMgetnextmod, 35
XPRMgetnextmoddso, 36
XPRMgetnextparam, 48
XPRMgetnextpbcomp, 49
XPRMgetnextpkgparam, 50
XPRMgetnextproc, 51
XPRMgetnextreq, 52
XPRMgetnextuncomptype, 88
XPRMgetobjval, 95
XPRMgetprevlistelt, 63
XPRMgetprobstat, 93
XPRMgetprocinfo, 54
XPRMgetrcost, 98
XPRMgetsdmax, 16
XPRMgetsetsize, 65
XPRMgetsettype, 66
XPRMgetslack, 100
XPRMgettypeprop, 55
XPRMgetuntype, 89
XPRMgetuntypeid, 90
XPRMgetunvalue, 91
XPRMgetvarnum, 102
XPRMgetversion, 17
XPRMgetversions, 18
XPRMgetvsol, 96
XPRMinit, 3, 4
XPRMinteger, 2
XPRMisrunmod, 27
XPRMjdn2date, 108
XPRMlinctr, 2
XPRMlist, 2
XPRMloadmod, 19, 20
XPRMloadmodsec, 20
XPRMmodel, 2, 19
XPRMmpvar, 2
XPRMpathcheck, 110
XPRMpreloaddso, 144
XPRMproc, 2
XPRMreal, 2
XPRMregstatdso, 130
XPRMremovetmpdir, 14
XPRMresetmod, 24
XPRMrunmod, 25
XPRMselectprob, 104
XPRMset, 2
XPRMsetbimprefix, 22
XPRMsetdefstream, 23
XPRMsetdefworkdir, 10
XPRMsetdsopath, 128
XPRMsetlocaledir, 11
XPRMsetrestrictions, 12
XPRMsetsdmax, 13
XPRMstoprunmod, 28
XPRMstring, 2
XPRMtermrunmod, 29
XPRMtime, 109
XPRMunloadmod, 30