FICO

Xpress-Mosel Libraries

Reference manual

Release 3.0

Last update April 2009

Published by Fair Isaac Corporation

How to Contact the Xpress Team

Information, Sales and Licensing

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 1926 315862
Fax: +44 1926 315854

FICO, Xpress team
Leam House, 64 Trinity Street
Leamington Spa
Warwickshire CV32 5YN
UK

Product Support

Email: Support@fico.com
(Please include *'Xpress'* in the subject line)

Telephone:

NORTH AMERICA
Tel (toll free): +1 (877) 4FI-SUPP
Fax: +1 (402) 496-2224

EUROPE, MIDDLE EAST, AFRICA
Tel: +44 (0) 870-420-3777
UK (toll free): 0800-0152-153
South Africa (toll free): 0800-996-153
Fax: +44 (0) 870-420-3778

ASIA-PACIFIC, LATIN AMERICA, CARIBBEAN
Tel: +1 (415) 446-6185
Brazil (toll free): 0800-891-6146

For the latest news and Xpress software and documentation updates, please visit the Xpress website at http://www.fico.com/xpress or subscribe to our mailing list.

Contents

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
`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 *`)

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

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 and completely frees the memory used by Mosel.

Related topics

XPRMinit.

XPRMgetdllpath

Purpose

Get the location of the xprm_rt dynamic library (Windows OS only).

Synopsis

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

Return value

The directory the file xprm_rt.dll is stored.

Further information

This function returns the location of the xprm_rt dynamic library. Note that this function is available only for the Windows version of Mosel.

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.

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

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].

`XPRMfindmod`	Find a model by its name or order number.	p. 22
`XPRMgetmodprop`	Get a property of a model.	p. 19
`XPRMgetnextdep`	Enumerate dependencies of a model.	p. 20
`XPRMgetnextmod`	Get the next model.	p. 21
`XPRMisrunmod`	Check if a model is running.	p. 16
`XPRMloadmod`	Load a Binary Model file.	p. 12
`XPRMresetmod`	Reset a model.	p. 14
`XPRMrunmod`	Run a model.	p. 15
`XPRMsetdefstream`	Set default input/output streams.	p. 13
`XPRMstoprunmod`	Stop a running model.	p. 17
`XPRMunloadmod`	Unload a model.	p. 18

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

XPRMloadmod

Purpose

Load a Binary Model file.

Synopsis

```
XPRMmodel XPRMloadmod(const char *bname, const char *intname);
```

Arguments

`bname` Name of a binary model file

`intname` Internal name (may be `NULL`)

Return value

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

Further information

This function returns the reference of a new model instance created from a binary model file. While loading a model from a file, Mosel also automatically opens any additional modules 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, string `"(none)"` is used as a default.

Related topics

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

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

```
void XPRMresetmod(XPRMmodel model);
```

Argument

`model` Reference to a model

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 *returned, const char *parlist);
```

Arguments

`model` Reference to a model

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

`parlist` String composed of model parameter initializations separated by commas, may be `NULL`

Return value

`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

`XPRM_RT_BREAK` Interruption because of a breakpoint (see Section 1.3)

Further information

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

Related topics

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

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`.

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`.

XPRMgetmodprop

Purpose

Get a property of a model.

Synopsis

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

Arguments

`model` Reference to a model

`prop` Property to retrieve. Possible values:

- `XPRM_PROP_NAME` Model name
- `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_DATE` Compilation date

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

Return value

0 if successful, 1 otherwise.

Further information

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: the field `integer` is used for `ID` and `VERSION`; `size` for `SIZE` and `DATE` (should be casted to the C type `time_t`); and `string` for the other properties. The returned version number is coded as an integer, for example, `1.2.3` is coded as `1002003`.

XPRMgetnextdep

Purpose

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

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: `1` for a package and `0` for a module

Return value

Reference pointer for the next call to `XPRMgetnextdep`.

Further information

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. Note that this function allocates memory when it is called for the first time and releases the allocated data when all items have been returned (*i.e.* the function returns `NULL`).

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.

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

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 string

`size` Maximum length of the string

Return value

Size of the generated string or -1 in case of error.

Further information

This function converts an entity of an external type into its textual representation. If the type does not support this convertion, the function produces a string using the address of the entity.

XPRMfindident

Purpose

Find an identifier in the dictionary.

Synopsis

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

Arguments

`model` Reference to a model

`text` Identifier

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

Return value

Type and structure of the returned dictionary entry, or 0 if the identifier is not registered.

Further information

This function returns the dictionary entry of a given identifier for a given model, together with the type and structure of the entry. Type and structure are bit encoded and 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

`XPRM_STR_REF` the object is a reference to a scalar

`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 types 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 code 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 */
    unsigned long 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.

Related topics

`XPRMgetnextident`, `XPRMgettypeprop`.

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`.

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`.

Related topics

`XPRMfindident`.

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.

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.

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`.

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` Returned type of the function or `XPRM_TYP_NOT` for a procedure

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 (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*'

`*` 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, XPRMalltypes *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 code
- `XPRM_TPROP_PBID` Problem index (`-1` if the type is not a problem)

`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 property `XPRM_TPROP_NAME`, the type name is returned in `value->string`, for the 3 other properties, the result is returned in `value->integer`.

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

3. The expanded code is available for user defined types only: it corresponds to the actual type code associated to a user defined type. 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 code 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 code is `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 code is `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.

Related topics

XPRMgetnextfield, XPRMgetnextpbcomp, XPRMfindtypecode.

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`).

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

Purpose

Get the size of a list.

Synopsis

```
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 and the storage class used for the list. The element type 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`: this correspond 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.

XPRMgetsetsize

Purpose

Get the size of a set.

Synopsis

```
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 and the storage class used for the set. The element type 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. 58
`XPRMcmpindices`	Compare two index tuples.	p. 59
`XPRMgetarrdim`	Get the number of dimensions of an array.	p. 49
`XPRMgetarrsets`	Get the index sets of an array.	p. 52
`XPRMgetarrsize`	Get the size of an array.	p. 51
`XPRMgetarrtype`	Get the type of an array.	p. 50
`XPRMgetarrval`	Get the value of an array entry.	p. 60
`XPRMgetfirstarrentry`	Get the list of indices of the first entry of an array.	p. 53
`XPRMgetfirstarrtruentry`	Get the list of indices of the first true entry of an array.	p. 56
`XPRMgetlastarrentry`	Get the list of indices of the last entry of an array.	p. 54
`XPRMgetnextarrentry`	Get the list of indices of the next entry of an array.	p. 55
`XPRMgetnextarrtruentry`	Get the list of indices of the next true entry of an array.	p. 57

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 and the storage class used for that array. The entry's type can be extracted using the macro `XPRM_TYP(type)`. The storage class can be extracted using the macro `XPRM_GRP(type)`. The macro `XPRM_ARR_DENSE` can be used to characterize a "dense table" (*e.g.* `XPRM_GRP(type) == XPRM_ARR_DENSE`).

Related topics

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

XPRMgetarrsize

Purpose

Get the size of an array.

Synopsis

```
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

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. 63
`XPRMgetnextfield`	Get the next field of a record type.	p. 62

XPRMgetnextfield

Purpose

Get the next field of a record type.

Synopsis

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

Arguments

`model` Reference to a model

`ref` Reference pointer or `NULL`

`code` Code of the record type

`name` Field name

`type` Field type

`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.

Related topics

`XPRMgetfieldval`.

XPRMgetfieldval

Purpose

Get the value of a field of a record.

Synopsis

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

Arguments

`model` Reference to a model

`ref` Reference to the record

`code` 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 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.

XPRMgetprobstat

Purpose

Get the problem status of a model.

Synopsis

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

Argument

`model` Reference to a model

Return value

Problem status.

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

`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 `.mat` 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 contraint 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 (≥ 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 (≥ 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

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).

Related topics

`XPRMfindident`.

1.2.6 Miscellaneous

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)

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 `jdn2date`.

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)*86400000000i64+(__int64)t*10000i64)`.

Related topics

`XPRMjdn2date`,`XPRMdate2jdn`.

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 has to 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`. The index of the first statement of a function is returned by `XPRMdbg_findproclndx`.

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. 89

XPRMdbg_runmod

Purpose

Run a model through the debugger interface.

Synopsis

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

Arguments

`model` Reference to a model

`returned` Pointer to an area where the result 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

`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 parameter `returned` receives the result of the execution (*e.g.* parameter value of the "exit" procedure). The bit `XPRM_RT_STOP` is set if the execution of the model has been interrupted by a call to the function `XPRMstoprunmod`.

2. 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.

3. 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`

4. 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_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_getlocation`, `XPRMdbg_findproclndx`.

XPRMdbg_getlocation

Purpose

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

Synopsis

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

Arguments

`model` Reference to a model

`lndx` Line index or -1 for current location

`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

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.

Related topics

`XPRMdbg_getnblndx`, `XPRMdbg_findproclndx`.

XPRMdbg_findproclndx

Purpose

Find the line index of a procedure or function.

Synopsis

```
int XPRMdbg_findproclndx(XPRMmodel model, XPRMproc proc);
```

Arguments

`model` Reference to a model

`proc` Reference to a procedure or function

Return value

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

Further information

This funtion returns the line index corresponding to the first statement of the provided procedure or function (as returned by `XPRMfindident`).

Related topics

`XPRMdbg_getnblndx`, `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`.

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. 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. 95
XPRMfinddso	Find a DSO descriptor from a module name.	p. 96
XPRMflushdso	Unload unused dynamic shared objects.	p. 97
XPRMgetdsoparam	Get the current value of a control parameter.	p. 98
XPRMgetdsopath	Get the directory list where DSO files are searched for.	p. 93
XPRMgetdsoprop	Get a property of a dynamic shared object.	p. 104
XPRMgetnextdso	Get next dynamic shared object.	p. 99
XPRMgetnextdsoconst	Enumerate constants of a module.	p. 100
XPRMgetnextdsoparam	Enumerate control parameters of a module.	p. 102
XPRMgetnextdsoproc	Enumerate procedures and functions of a module.	p. 103
XPRMgetnextdsotype	Enumerate native types of a module.	p. 101
XPRMgetnextiodrv	Get the next IO driver in the list of available drivers.	p. 105
XPRMpreloaddso	Explicitly load the named module.	p. 106
XPRMregstatdso	Declare a module as static.	p. 94
XPRMsetdsopath	Set the directory list where DSO files are stored.	p. 92

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` then in `MOSEL/dso`. 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

`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 (`MOSEL/dso`) 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 if successful, 1 otherwise.

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.

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 Mosel parameter (like `"realfmt"`).

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

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`).

Related topics

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

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 `getparm` 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, XPRMalltypes *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 if the module is certified
- `XPRM_PROP_NBREF` Number of loaded models that use the module
- `XPRM_PROP_PATH` Path to the actual module file

`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: the field `string` is used for `NAME`, `SYSCOM` and `PATH`; and `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. 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:%#lx", (unsigned long)simpleout);
XPRMsetdefstream(NULL, XPRM_F_ERROR, fname);  /* redirect error str. to 'simpleout' */
 ...
```

1.5.3 `mem` driver

With this driver, a block of memory is used as data source. Two 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.

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

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

where `label` is an identifier whose first character is a letter and `minsize` an initial amount of memory (in bytes) to be allocated. When this kind of 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 (see `XPRMfindident`).

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. Optionally a pointer to a long integer value may be provided (`actualsize`): 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];
unsigned long actualsize;

sprintf(fname, "mem:%#lx/%u/%#lx", (unsigned long)blk, sizeof(blk),
        (unsigned long)&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 */
```

1.5.4 `raw` driver

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='%#lx/%u', RES='%#lx/%u'", (unsigned long)d, sizeof(d),
                                                (unsigned long)r, sizeof(r));
XPRMrunmod(mod, &result, params);
```

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.

XPRMcompmod

Purpose

Compile a model source file.

Synopsis

```
int XPRMcompmod(const char *options, const char *srcfile, const char
      *dstfile, const char *userc);
```

Arguments

`options` Compilation options (may be `NULL`). Possible values:

- `"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
- `"G"` Include tracing information: with this option the model can be run through the debugger for an execution step by step
- `"s"` Strip symbols: secure the bim file by removing all private symbol names used in the source model
- `"p"` parse only: stop after the syntax analysis of the source file, do not compile (no file generated)

`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`)

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.
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`.

Related topics

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

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`.

Index