FICO® Xpress Mosel Native Interface

6.10

REFERENCE MANUAL

FICO® Xpress Optimization

FICO

Xpress Mosel 6.10 (FICO® Xpress 9.7)

Deliverable Version: A

Last Revised: April 2024

Contents

Introduction

The Mosel environment may be extended by means of *modules*. A module may bring into the Mosel Language:

- constant symbols
- functions and procedures
- types
- operators (like '+') to be applied to the types introduced by the module
- control parameters
- IO drivers

Mosel comes with a default set of modules (like *mmsystem* or *mmsheet*) but a user can implement his own modules which are a special kind of Dynamic Shared Object (DSO) written in the C (or C++) programming language.

The Mosel Native Interface is a set of conventions that a DSO must respect to be accepted as a module by Mosel. This document describes these conventions and also gives a comprehensive list of the Mosel functions that can be accessed from a module.

CHAPTER 1

The Native Interface

1.1 Module management in Mosel

In Mosel, all basic module operations are performed by a *Module Manager*. This manager is in charge of loading and unloading the modules as well as maintaining various pieces of information about the modules (version number, features provided, reference counting). Whenever a module is requested either by the Model Compiler or to run a previously compiled model, the Module Manager looks for the module and, if it is not yet in core memory, loads it from the disk and initializes it by calling its *initialization function* (see Section 1.2). After a period of inactivity or on request, the Module Manager may unload unused modules.

1.1.1 Use of modules for compiling a model

The directive `uses "modname"` informs the Model Compiler that the module *modname* is required by the model being compiled. For every module that appears in a `uses` directive, the compiler queries the Module Manager in order to extend its dictionary with the symbols and operators defined by the module. The following information is therefore recorded for every symbol:

- identifier (*i.e.* name of the subroutine or type)
- origin: the identity of the module that defines the symbol
- depending on the type of the symbol:
  - constant: the value associated to the identifier
  - procedure/function/operator: internal code in the module and types of the result and parameters
  - type: internal code in the module and properties (*e.g.* can this type be translated into a string?)

During the compilation, each time an external symbol is encountered, assuming it is used appropriately (type of result as required, valid parameters for routines, *etc.*), the following operation is performed depending on the category of the symbol:

- constant: the symbol is replaced by its value
- procedure, function or operator: a function call is prepared using the internal code of the symbol
- type: the function call that corresponds to the required operation (*e.g.* creation) is prepared using the internal code of the symbol

In the BIM file that is generated during the compilation of a model, the compiler saves the table of modules it requires together with their version numbers. Only the modules that are effectively required (those from which functions are to be called) are stored in this table.

1.1.2 Use of modules for running a model

When the BIM file is loaded, Mosel queries the Module Manager for the modules required by the model. The model is ready for execution only if all the modules with their specified versions are available. The modules are considered to be “in use” as long as the model is in core memory.
During the execution of the model, the function calls prepared during the compilation phase are executed.

1.2 The initialization function

Once a module is loaded in core memory, the Module Manager calls a special function: the *initialization function*. Through this function, the constant symbols (Section 1.2.1), subroutines (Section 1.2.2), types (Section 1.2.3), and services (Section 1.2.4) that are provided by the module are passed on to Mosel. The control parameters of the module are not communicated in this way, they require the definition of a dedicated service function (Section 1.5.5) and the implementation of two specific library functions (Section 1.2.2).

In order for Mosel to find this initialization function, it must be named *modulename*_init (where *modulename* is the name given to the module) and have the following signature:

```
DSO_INIT modulename_init(XPRMnifct nifct, int *interver, int *libver,
                         XPRMdsointer **interf)
```

The parameters are used to exchange information about version numbers and the functionality provided by the module.

- nifct is a table of functions provided by Mosel that can be called from the module during its processing. They are used to access the data of the running model (see Section 2) and is usually saved in a global variable for later use
- interver is used to tell Mosel which version of the Native Interface is employed for the implementation of this module. This parameter must always be assigned the value XPRM_NIVERS. By default, the interface version is the one of the current Mosel version but it is possible to select an older release (such that the generated module can be used with this release) by defining the macro XPRM_NICOMPAT before including the "xprm_ni.h" header file. This macro must refer to the Mosel target version. For instance:

  ```
  #define XPRM_NICOMPAT 3000000
  #include "xprm_ni.h"
  ```

  will use the API version of Mosel 3.0.0 (functions introduced after this release are disabled)
- libver is the version number of the module: it is generated using the macro XPRM_MKVER(M,n,r) where (M,n,r) stands for *(major version number, minor version number, release number)*. Each number must be an integer between 0 and 999.
- interf is a structure composed of 4 tables (and their respective sizes) describing the constants, the functions, the types and the services implemented by the module

The format of the main interface structure XPRMdsointer is the following:

```
{
 int sizec; XPRMdsoconst *tabconst;
 int sizef; XPRMdsofct *tabfct;
 int sizet; XPRMdsotyp *tabtyp;
 int sizes; XPRMdsoserv *tabserv;
}
```

Every table in this structure (constants, subroutines, types, and services) is preceded by its size. The four tables are described in detail in the following sections.

Example:

```
static XPRMnifct mm;

DSO_INIT mymodule_init(XPRMnifct nifct, int *interver, int *libver,
                       XPRMdsointer **interf)
{
 mm=nifct;                  /* Save the table of NI functions */
 *interver=XPRM_NIVERS;     /* Mosel NI version */
 *libver=XPRM_MKVER(0,0,1); /* Module version */
 *interf=&dsointer;         /* Pass info about module contents to Mosel */
 return 0;
}
```

1.2.1 Table of constants

Each entry of the table of constants is a pair (*constant name*, *constant value*). If a constant is to be declared as part of a namespace its name must be fully qualified (*e.g.* `"mynamspc~mycst"`). A module can define integer, real, Boolean and string constants. The initialization of the table can be done using the following macros:

```
XPRM_CST_INT(char *name, int value)
XPRM_CST_BOOL(char *name, int value)
XPRM_CST_STRING(char *name, char *value)
XPRM_CST_REAL(char *name, static const double value)
```

Note that for real constants, a static variable has to be provided instead of a constant number.

```
static const double myreal=12.456;

static XPRMdsoconst tabconst[]=
    {
     XPRM_CST_INT("MYINT", 10),
     XPRM_CST_BOOL("MYBOOL", XPRM_TRUE),
     XPRM_CST_STRING("MYSTR", "text"),
     XPRM_CST_REAL("MYREAL", myreal)
    };
```

The information provided by the table of constants is used only during the compilation phase of the model: constants are immediatly replaced by their values. As a consequence, a module that only defines constants is only required for the compilation of a model using it; at execution time it is not loaded again.

1.2.2 Table of functions

The table of functions describes the functions, procedures, and operators that will be available in the Mosel language. Each entry of the function table is of the following structure:

```
{
 char *name;
 int code;
 int type;
 int nbpar;
 char *parstr;
 int (*fct)(XPRMcontext ctx, void *libctx);
}
```

name: The name that will be used in the Mosel language. If the subroutine is to be declared as part of a namespace its name must be fully qualified (*e.g.* `"mynamspc~myfct"`). Note that different subroutines (or operators) may have the same name as long as they are not expecting the same parameters (*overloading*). It is also possible to overload a predefined function or procedure with the same restriction as for user defined symbols.

Overloading cannot apply between function and procedure (*i.e.* a procedure cannot overload a function and vice versa).

code: An internal code for the subroutine or operator.
This code must be either an integer value $\geqslant 1000$ or a predefined code. Note that the entries in the table of functions must be sorted in ascending order of their internal code.

type: The type returned by the routine.
For a function, the possible values are: `XPRM_TYP_INT`, `XPRM_TYP_REAL`, `XPRM_TYP_STRING`, `XPRM_TYP_BOOL` to indicate an integer, real, string or Boolean return value respectively. The type must be `XPRM_TYP_EXTN` if the function returns an entity of a type managed by the module (*i.e.* the function is a constructor) — in this case, the first word of the parameter string is the name of this type followed by a colon. If the function returns a set or a list the type must also be `XPRM_TYP_EXTN` and the first word of the parameter string must start with `"&{"` for a set or `"&["` for a list followed by the element type and a colon. The element type is specified by a letter for basic types (same convention as for the `parst`) or the letter `n` and the type name for a native type. If the subroutine is a procedure the type must be `XPRM_TYP_NOT`.
By default a function returning a non-basic type is considered to be a constructor (*i.e.* the returned value is a newly created entity). If such a function returns a reference to an existing entity it must increase the reference count of this object (see Section 1.4.2) and have the flag `XPRM_FTYP_PTR` as part of its type.
Functions the name of which starts with the string `"get"`, returning a basic type (*i.e.* integer, real, string or Boolean) and taking as their only argument a native type, `linctr` or `mpvar` are identified as *attribute accessors* (see `findattrdesc`). For instance the function `"getsol(mpvar):real"` returns the *sol* attribute of a decision variable. Adding `XPRM_FTYP_NOATTR` to the type of a function of this kind prevents Mosel from recording it as an attribute accessor.

nbpar: The number of parameters required by the routine

parstr: The parameter string is used to describe the type of each parameter.
This string is composed with the following characters:

`i`	an integer
`r`	a real
`s`	a registered text string (see `regstring`)
`S`	a text string (it might be deleted after the end of execution of the routine)
`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)
`u`	a union (of any kind)
`f`	a procedure or function (of any kind)
`|`*xxx*`|`	external type named '*xxx*'
`!`*xxx*`!`	the set named '*xxx*'
`A`*ndx*`.`*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.
`E`*t*	a set of type '*t*'
`L`*t*	a list of type '*t*'
`F(`*sig*`)`	a procedure with signature '*sig*' (this signature cannot include the tag `F`)
`F`*t*`(`*sig*`)`	a function of type '*t*' with signature '*sig*' (this signature cannot include the tag `F`)
`?`	any type, the routine will receive actually 2 parameters for this marker: the first one, an integer, is the type of the following one (the effective value of the argument)
`*`	must be the last character: the function has a variable number of arguments

If the last character of the parameter string is `*`, the function accepts a variable number of arguments: the corresponding parameter is a list (possibly empty) containing the supplementary parameters.
Moreover, if the function is of type `XPRM_TYP_EXTN`, the string starts with the name of the type followed by a colon.
Example: `"mytype:ir|mytype|s*"` is the signature of a function of type 'mytype' expecting at least 4 parameters (integer, real, mytype and string).
The signature for a function that returns a list of text and take a real as only argument will be: `"&[ntext:r"`.

fct: The function Mosel has to call to perform the operation.
The prototype of all functions must be (see Section 1.3):

```
int functionname(XPRMcontext ctx, void *libctx);
```

Example:

```
static int my_getsol(XPRMcontext ctx, void *libctx);
static int my_getname(XPRMcontext ctx, void *libctx);
static int my_eql(XPRMcontext ctx, void *libctx);
static int my_new(XPRMcontext ctx, void *libctx);

static XPRMdsofct tabfct[]=
    {
```

```
    {"getsolarray", 1000, XPRM_TYP_NOT, 2, "aa", my_getsol},
    {"getname", 1005, XPRM_TYP_STRING, 1, "|mytype|", my_getname},
    {"@=", 1011, XPRM_TYP_BOOL, 2, "|mytype||mytype|", my_eql},
    {"@&", 1015, XPRM_TYP_EXTN, 3, "mytype:sri", my_new}
   }
```

For details on the definition of operators (such as the third and fourth entries in this example) the reader is referred to Section 1.4.4.

1.2.3 Table of types

Types introduced by modules are handled by Mosel just like any other standard type (integer, real...). To define a type, some specific functions have to be provided by the module. Each entry of the table of types contains the following items:

name (`char *`): The name of the type that will be used in the Mosel language. If the type corresponds to a problem extension, the name must have the form `mainpbtyp.extn` (see Section 1.4.3). If the type is to be declared as part of a namespace its name must be fully qualified (*e.g.* `"mynamspc~mytype"`).

code (`int`): An internal code for the given type. This code is an integer value not larger than 65535. Note that types must be sorted in ascending order of their internal code.

props (`int`): A bit coded set of properties. The supported properties are:

- `XPRM_DTYP_PNCTX`: If this flag is set, the function `tostring` (see below) can be called with a `NULL` context.
- `XPRM_DTYP_RFCNT`: If this flag is set, the module handles reference count for this type. As a consequence Mosel may call the function `create` (see below) with a reference to a previously created object for increasing its reference count. The function `delete` (which is mandatory in this case) is then called as many times as the `create` function has been used for a given object before this object is effectively released. When this property is not available for a type, Mosel handles itself the reference counting: this is in general less efficient except if the type is a problem extension (reference counting is not used in this case).
- `XPRM_DTYP_APPND`: If this flag is set, the function `copy` supports appending.
- `XPRM_DTYP_ORSET`: If this flag is set, the function `copy` can be called only for resetting an object.
- `XPRM_DTYP_PROB`: This flag must be set if the type corresponds to a problem.
- `XPRM_DTYP_SHARE`: If this flag is set, the `create` function of this type supports creation of shared entities.
- `XPRM_DTYP_TFBIN`: If this flag is set, the `tostring` and `fromstring` functions of this type support exportation and exportation in binary format.
- `XPRM_DTYP_ORD`: If this flag is set, the `compare` function implements all comparison operators (*i.e.* objects of this type are ordered).
- `XPRM_DTYP_CONST`: If this flag is set, the `create` function supports creation of constants and the `copy` function can compute hash values.
- `XPRM_DTYP_ANDX`: The type is an *array indexer* when this flag is set, and the service `XPRM_SRV_ARRIND` must be defined (see Section 1.5.21).
- `XPRM_DTYP_NAMED`: If this flag is set, the `create` function of this type supports name association (see `setentname`).

create function (`void *`): The function Mosel has to call for creating an object of this type. The function must return a pointer to this new object or `NULL` in case of failure. The prototype of create is:

```
void *(*create)(XPRMcontext ctx, void *libctx, void *ref,int tnop)
```

This function is mandatory. If the module does not support reference count for this type, the parameter `ref` is always `NULL` and can be ignored. Otherwise, the flag `XPRM_DTYP_RFCNT` has to be set (see above) and whenever this function is called with a valid pointer as the third parameter, the reference count for the corresponding object must be incremented (no new object has to be created). The value returned should be the provided reference. The last parameter is the order number associated to this type for the running model.
If the type has property `XPRM_DTYP_SHARE`, `XPRM_DTYP_CONST` or `XPRM_DTYP_NAMED` the semantic of the last parameter changes: it stores both the type order number (which can be extracted using the macro `XPRM_TYP(tnop)`) and a code indicating what operation to perform. The macro `XPRM_CREATE(tnop)` returns the following operation codes:

- `XPRM_CREATE_NEW`: The routine behaves as described above (*i.e.* handling of an ordinary entity).
- `XPRM_CREATE_SHR`: The routine has to return a shared entity: if the pointer `ref` is `NULL` a newly created entitiy must be returned. Otherwise the provided reference `ref` points to the initial object that is to be shared (that has been created by a preceding call to this routine from another model). The function does not have to return this reference (this can be another datastructure) but it is expected that both pointers refer to the same entity. Note that several models may call this function with the same reference object at the same time. This option will be used if the type has the property `XPRM_DTYP_SHARE`.
- `XPRM_CREATE_CST`: The function has to return a constant copy of the provided object. The module must make sure that any attempt at modifying such a constant (*e.g.* by using an assignment) will cause a runtime error. This option will be used if the type has the property `XPRM_DTYP_CONST`.
- `XPRM_CREATE_NAMED`: The entity `ref` has been associated with a name with `setentname` (this identifier can be retrieved with `getentname`), the function must return `ref` without updating its reference count. It is recommended to release this association when the entity is deleted from the *delete function* using `setentname`.

delete function (`void *`): The function Mosel has to call for deleting an object previously allocated using the `create` function.

```
void (*fdelete)(XPRMcontext ctx, void *libctx, void *todel,int typnum)
```

This function is optional (the entry may be `NULL`) when reference count is not handled by the module, if defined, it is used to delete local and temporary objects. Note that if reference count is implemented, this function will be called as many times as the `create` function has been called for a given reference, the object must be deleted only the last time the function is used. Note that global objects are not explicitly deleted: it is the responsibility of the module to release the resources associated to these objects using the reset and onexit services (see Section 1.5.1 and 1.5.10). The last parameter is the order number associated to this type for the running model.

tostring function (`void *`): This function has to be called by Mosel for getting a textual representation of an object.

```
int (*tostring)(XPRMcontext ctx, void *libctx, void *obj, char *dest,
                int maxsize,int typnum)
```

The textual representation of `obj` encoded in UTF-8 has to be copied into `dest` the maximum length of which is `maxsize`. The reference `obj` might be `NULL`: in this case the function is expected to return the textual representation of an entity in its initial state (*i.e.* just after having been created). The function must return the length of the generated string (excluding the terminating null byte) or a negative value in case of error. If the

string that is to be returned in `dest` exceeds the given maximum length, function `tostring` returns the required length but not the string itself : it is then called a second time with a sufficiently large maximum size.
If the type has property `XPRM_DTYP_TFBIN` the system may use this routine to generate a binary representation of the entity: in this case the last parameter is bit encoded (type number can still be retrieved using `XPRM_TYP(typnum)`) and the bit `XPRM_TFSTR_BIN` is set when the binary fomat is requested. The result of the call is expected to be a platform independent sequence of bytes (instead of the default textual representation) that can be decoded by the function `fromstring`.
This function is optional (the entry may be `NULL`), if defined, it is used for displaying values (procedures `write`/`writeln`) and by the `initializations to` procedure.

fromstring function (`void *`): This function has to be called by Mosel for initializing an object from a textual representation.

```
int (*fromstring)(XPRMcontext ctx, void *libctx, void *obj,
                  const char *src,int typnum, const char **end)
```

The object `obj` is initialized with the content of the string `src` encoded in UTF-8. When the `end` parameter is not `NULL`, the pointer to the first character not used by the conversion has to be returned via this parameter. It must receive a copy of `src` in case of failure. If successful, the function must return 0; any other value is interpreted as a failure.
If the type has property `XPRM_DTYP_TFBIN` the system may use this routine to initialize the entity from a binary representation produced by the `tostring` function: in this case the last parameter is bit encoded (type number can still be retrieved using `XPRM_TYP(typnum)`) and the bit `XPRM_TFSTR_BIN` is set when the input buffer is in binary fomat. In this mode the parameter `end` is initialized with a reference to the first byte after the data (*i.e.* the length of the input buffer is `*end-src`).
This function is optional (the entry may be `NULL`), if defined, it is used by the `initializations from` procedure.

copy function (`void *`): This function is required for assignments not explicitly stated (*e.g.* in array initialization or when assigning records) and may be used to generate some assignments (as a replacement for the `"@:"` and `"@P"` operators, see section 1.4.4).

```
int (*copy)(XPRMcontext ctx, void *libctx, void *dest,void *src,int tnop)
```

The parameter `tnop` is bit encoded: it stores both the type order number (which can be extracted using the macro `XPRM_TYP(tnop)`) and a code indicating what operation to perform. The macro `XPRM_CPY(tnop)` returns the following operation codes:

- `XPRM_CPY_COPY`: The object `dest` receives the content (or becomes a copy) of `src` (which may be `NULL`). This operation is not used if the type has property `XPRM_DTYP_ORSET`.
- `XPRM_CPY_RESET`: The object `dest` is reset (*i.e.* it returns to its initial state). This operation is essentially used to implement the `reset` function of the Mosel language.
- `XPRM_CPY_APPEND`: The object `dest` is extended with a copy of `src` (which may be `NULL`). This operation is used if the type property `XPRM_DTYP_APPND` is set and property `XPRM_DTYP_ORSET` is not set.
- `XPRM_CPY_HASH`: The pointer `dest` references an unsigned integer that must be populated with a hash value computed from the object `src` (which may be `NULL`), the implementation may use `hashmix` for this calculation. This operation is used if the type property `XPRM_DTYP_CONST` is set and property `XPRM_DTYP_ORSET` is not set.

If successful, the function must return 0; any other value is interpreted as a failure.
This function is optional (the entry may be NULL) but if it is missing, the operations where it is necessary are disabled by the compiler for the corresponding type.

compare function (`void *`): This function is required for comparison of aggregated objects (*e.g.* when testing equality of records including fields of this type), it may also be used to implement comparators if the corresponding functions are not defined.

```
int (*compare)(XPRMcontext ctx, void *libctx, void *obj1,void *obj2,int tnop)
```

The parameter `tnop` is bit encoded: it stores both the type order number (which can be extracted using the macro `XPRM_TYP(tnop)`) and a code indicating what operation to perform. The macro `XPRM_COMPARE(tnop)` returns the following operation codes:

- `XPRM_COMPARE_EQ`: Test whether `obj1` and `obj2` are equal.
- `XPRM_COMPARE_NEQ`: Test whether `obj1` and `obj2` are different.
- `XPRM_COMPARE_LTH`: Test whether `obj1` is less than `obj2`.
- `XPRM_COMPARE_LEQ`: Test whether `obj1` is less or equal than `obj2`.
- `XPRM_COMPARE_GEQ`: Test whether `obj1` is greater or equal than `obj2`.
- `XPRM_COMPARE_GTH`: Test whether `obj1` is greater than `obj2`.
- `XPRM_COMPARE_CMP`: Return 0 if `obj1` and `obj2` are the same, `-1` if `obj1` is less than `obj2` and `1` otherwise.

If successful, the function must return `1` if the comparison is true and `0` otherwise (except for the `XPRM_COMPARE_CMP` comparison that may also return `-1`). Error conditions are reported using the special return value `XPRM_COMPARE_ERROR`.
This function is optional (the entry may be NULL) but if it is missing, the operations where it is necessary are disabled by the compiler for the corresponding type. By default the compiler expects that only the 2 first operations are defined, the type property `XPRM_DTYP_ORD` indicates that the function supports all operations.

Example:

```
static XPRMdsotyp tabtyp[]=
{
 {"firsttype", 1, 0, createfirst, del1, tostr1, fromstr1,copy1,cmp},
 {"secondtype", 2, XPRM_DTYP_RFCNT, create2, delete2, NULL, NULL, NULL,NULL},
 {"mpproblem.mypb", 3, XPRM_DTYP_PROB, newpb, delpb, NULL, NULL, cppb,NULL}
};
```

1.2.4 Table of services

Services are special tasks that are not directly linked to the Mosel language itself (that is, they are not visible to the user of a module). Under some particular circumstances, Mosel looks for a service. If this service is implemented by the module, the corresponding function is called. Each entry of the table of services is the pair (*service code, function to call*).

Example:

```
static XPRMdsoserv tabserv[]=
{
 {XPRM_SRV_RESET,(void *)my_reset},
 {XPRM_SRV_PRIORITY,XPRM_MKPRIORITY(-1)},
 {XPRM_SRV_UNLOAD,(void *)my_unload}
};
```

Note that all services are optional. See section 1.5 for a comprehensive list of services.

1.3 Defining subroutines

All library functions that implement subroutines (or operators, see Section 1.4.4) have the same prototype

```
int functionname(XPRMcontext ctx, void *libctx);
```

The first parameter of such a function is the *Mosel execution context* under which the function is called. This context describes the state of the Mosel Virtual Machine plus various pieces of information related to the model that is executed. It is required by most functions of the Native Interface. The second parameter is the module context defined by the reset service (see Section 1.5.1). If this service is not implemented, the value of this parameter is `NULL`.
The return value of a library function must be `XPRM_RT_OK` if the function succeeded, `XPRM_RT_ERROR` if it failed (in which case the execution terminates with an error) or `XPRM_RT_STOP` to interrupt the execution of the model. It is also possible to terminate the execution in the same way as `exit(code)` does by pushing onto the stack the exit code then returning `XPRM_RT_EXIT`.
If the execution of a routine is not immediate (it performs a solution algorithm that requires several seconds for instance), it is recommanded to implement *cancelation points*: from time to time, the routine should call the NI function `chkinterrupt` to check whether execution is to be continued. If the return value of this function is not 0, the execution is expected to terminate and the routine has to interrupt its processing as soon as possible then return.

During the execution of the function, the actual parameters of the subroutine (in the Mosel language) have to be taken from the Mosel stack and if the routine is a function, the return value must be put onto this stack before the termination of the function (this is not required if the routine terminates with `XPRM_RT_ERROR` that results in the interruption of the execution). The following macros are provided for accessing the Mosel stack.

Macros for taking objects from the stack:

```
int XPRM_POP_INT(XPRMcontext ctx)
double XPRM_POP_REAL(XPRMcontext ctx)
XPRMstring XPRM_POP_STRING(XPRMcontext ctx)
void *XPRM_POP_REF(XPRMcontext ctx)
XPRMalltypes XPRM_POP_ANY(XPRMcontext ctx)
```

Macros for putting objects onto the stack:

```
XPRM_PUSH_INT(XPRMcontext ctx, int i)
XPRM_PUSH_REAL(XPRMcontext ctx, double r)
XPRM_PUSH_STRING(XPRMcontext ctx, XPRMstring s)
XPRM_PUSH_REF(XPRMcontext ctx, void *r)
XPRM_PUSH_ANY(XPRMcontext ctx, XPRMalltypes a)
```

Macro for accessing the top of the stack:

```
XPRMalltypes *XPRM_TOP_ST(XPRMcontext ctx)
```

The macro `XPRM_FREE_ST(XPRMcontext ctx)` returns the number of free elements on the stack (*i.e.* that can be used for pushing values). Note that at the beginning of the execution of a native call there are always at least 4 free positions on the stack.

Note that the Mosel stack manipulates only three basic types: integer, real, and string. Boolean values are handled as integers (0 is `false`, 1 is `true`); all other types (including arrays, sets and external types) are passed by reference (the macros `XPRM_POP_REF` and `XPRM_PUSH_REF` have to be used for those types). Routines taking references as parameters must have appropriate handling for the `NULL` pointer: this is the representation of an empty string and the value returned when accessing an uninitialized object (set, array or non existent dynamic array entry).

Text strings manipulated by Mosel are stored in a dictionary. As a consequence, strings produced by a native routine (for instance as the result of a concatenation) have to be registered using the function

`regstring` before being sent to Mosel either as a return value (macro `XPRM_PUSH_STRING`) or stored in a Mosel object (as an identifier, set element or array entry).

Mosel maintains a reference count of all referenced objects (decision variables, linear constraints, lists, sets, arrays and external types): an object is released when its last reference is deleted. If a native routine needs to keep a reference to an object after its termination (to be used later in a subsequent call for instance) it should save the reference using `newref` in order to make sure the object will not be deleted. The native code should then use `delref` after it no longer requires the saved reference.

Example:

Assume the routine `myfct` takes an integer, a real and a set of decision variables as parameters and returns a string. If the C function providing the implementation of this routine is `c_myfct` and its internal code is 1010, the declaration in the table of functions is:

```
{"myfct",1010,XPRM_TYP_STRING,3,"irEv",c_myfct}
```

The function `c_myfct` has to get from the Mosel stack the three parameters and before its termination, to put back the result value. The skeleton of this function is therefore:

```
int c_myfct(XPRMcontext ctx, void *libctx)
{
 int int_param;
 char *result;
 double real_param;
 XPRMset set_param;

 int_param =XPRM_POP_INT(ctx);   /* Get the first parameter */
 real_param=XPRM_POP_REAL(ctx);  /* Get the second parameter */
 set_param =XPRM_POP_REF(ctx);   /* Get the third parameter */

 /*
    Body of the function: assigns the result variable
 */
                                    /* Put the result onto the stack */
 XPRM_PUSH_STRING(ctx,mm->regstring(ctx,result));
 return XPRM_RT_OK;              /* The function succeeded */
}
```

If a module defines control parameters, it needs to implement the special routines `getparam` and `setparam` for its parameters. Function `getparam` takes as its only argument the code of the control parameter in the module (obtained at compilation through the service “find parameter”, see Section 1.5.5) and returns the current value of the parameter. The procedure `setparam` has two arguments, the code of the parameter and its new value.
At the beginning of the table of functions, the following two lines must be added in the order shown here (assuming that `my_getpar` and `my_setpar` are the implementations of the two subroutines):

```
{"", XPRM_FCT_GETPAR, XPRM_TYP_NOT, 0, NULL, my_getpar},
{"", XPRM_FCT_SETPAR, XPRM_TYP_NOT, 0, NULL, my_setpar},
```

1.4 Defining types

A module defines a type via an entry in the table of types (see Section 1.2.3) that specifies the type creation function and optionally, functions for type deletion and transformation to/from string and copy. Besides these five functions, a module needs to implement certain other library functions when it defines a type.

1.4.1 Memory management

Mosel does not guarantee that it will call the delete function for each object created with the create

function. It is therefore required to either use `memalloc`/`memfree` for allocation of entities or implement a module context in order to keep track of all created objects and delete them all at once when the context has to be released (using the reset service, see Section 1.5.1).

1.4.2 Reference counting

The Mosel language makes possible for a given object to be referenced several times in a model. If such an object has to be deleted, Mosel must make sure that there is no remaining reference to this object before releasing it. A typical example of this situation is when an object is defined in a subroutine and added to a set defined globally. In this case, when the subroutine terminates, the object can be deleted only if it is not included in any set. The same remark applies to decision variables `mpvar`: locally defined variables can be deleted only if they do not appear in any constraint.

In order to know when a referenced object (basically all external types as well as `mpvar` and `linctr`) can be deleted, Mosel maintains a counter of references for each object: whenever a new reference is created for an object its counter is increased and each time this object should be deleted its counter is decreased. Objects are created with a counter initialised to 1 and effectively deleted when their counter reaches 0.

Although Mosel can handle *reference counting* for external types, it is recommended for modules to provide support for this mechanism for the types they publish. The interface relies on the `create` and `delete` functions that are used respectively to increase and decrease the reference counts of the associated objects (see Section 1.2.3).

1.4.3 Problem types

Problem types are implemented as special module types: the property `XPRM_DTYP_PROB` is set; both `create` and `delete` routines are provided; `tostring` and `fromstring` functions are not used. The creation and deletion functions are used by the system to manage *problem contexts*: the current context of each problem type is stored in the `pbctx` table of the Mosel execution context which is passed to all NI routines. The index of a particular problem type in this table can be obtained using the function `gettypeprop` to get the `XPRM_TPROP_PBID` property (use `findtypecode` for getting the type code required by this routine). Since this index is not changed during the execution of the model, it can be retrieved and saved when creating the module context (see Section 1.5.1). The table of problem contexts is automatically updated whenever a `with` construct is open or closed: native routines have to refer to this table to get their active context when processing operations related to problems.

A problem type may be extended by means of *problem extensions*. When a problem is created/deleted, a corresponding context of each of its extensions is also created/deleted. Similarly, when a problem is activated (through a `with` construct in the model), all of its associated extensions are also made active. A problem extension is identified by its name that must have the form `mainpb.extn` where `mainpb` is the name of the problem to be extended and `extn` the extension name. For instance, the Xpress Optimizer is declared as an extension to `mpproblem` - it is named `mpproblem.xprs`. Since it is not referenced directly, reference counting is not required for a type representing a problem extension. Note also that extensions may restrict the capabilities of a problem type: for instance, the `copy` operator of a problem type will be disabled if one of its extensions does not support the operation.

1.4.4 Special functions/operators

For handling the types introduced by the module, it is possible to define operators that are declared as functions (*cf* function table, Section 1.2.2) with a special name: all operators have a two-character name, the first character of which is always '@'. Operators can be defined for any type and return any type; however, they cannot replace a predefined operator. For instance the addition of reals `@+(r,r):r` cannot be re-defined in a module. Furthermore, it is not possible to re-define directly any aggregate operators (`prod`, `sum`, `and`, `or`) or set operators (`inter`, `union`, `in`, `max`, `min`).

1.4.4.1 Basic constructors

Operator	Operation	Return value
@&(C):C	duplication (cloning)	new object
@&(params):C	construction	new object
@0:C	identity for sums (0-element)	new object
@1:C	identity for products (1-element)	new object
@2:C	init. for min (biggest value)	new object
@3:C	init. for max (smallest value)	new object

The duplication operator is never called explicitly but its definition is required, for instance, by certain types of assignment.

A constructor may also be named `"@&I"` if it takes a single parameter of a basic type: the compiler will use such a constructor to perform implicit conversions in subroutine parameters.

The definition of @0 for a given type *C* implies the aggregate operator `SUM` if *@+(C,C):C* is defined for this type and the aggregate `OR` if *@o(C,C):C* is defined. Similarly, with the 1-element @1 the Mosel compiler can generate the aggregate operator `PROD` if *@*(C,C):C* is defined and the aggregate `AND` if *@a(C,C):C* is defined.

The definition of @2 for a given type *C* implies the aggregate operator `MIN` if the type has property `XPRM_DTYP_ORD`. Similarly, with @3 the Mosel compiler can generate the aggregate operator `MAX` if the type is ordered.

1.4.4.2 Arithmetic operators

Operator	Operation	Return value	Remarks
@+(A,B):C	addition	A + B $\rightarrow$ C	commutative
@S(A,B):C	addition	B + A $\rightarrow$ C	
@-(A,B):C	subtraction	A - B $\rightarrow$ C	implied by @+(A,B):C with @-(B):B
@-(A):C	negation	- A $\rightarrow$ C	
@*(A,B):C	multiplication	A * B $\rightarrow$ C	commutative
@/(A,B):C	division	A / B $\rightarrow$ C	
@d(A,B):C	integer division	A div B $\rightarrow$ C	
@m(A,B):C	modulo operation	A mod B $\rightarrow$ C	
@^(A,B):C	exponential operation	$A^B \rightarrow C$	

If @S and @0 are defined for a given type, the aggregate operator `SUM` can be generated by the Mosel compiler (if @S is not defined, @+ will be used instead, the difference is the order of operands). The same generation occurs for the aggregate operator `PROD` if @1 and @* are defined. The `SUM` operator can also be generated when the addition returns a different type. In this case, if type1+type1->type2, operator @0 for type2 is required as well as the operation type1+type2->type2 (commutativity does not apply for this construct).

Where operations are marked 'commutative', Mosel deduces the result for (B,A) if the operation is defined for (A,B), assuming that A and B are of different types.

A and *B* (if of an external type) must be deleted by the operator.

1.4.4.3 Logical operators

Usually, if a type is defined for these operators, it is not defined for the arithmetic operators.

Operator	Operation	Return value
@a(A,B):C	logical 'and'	A and B → C
@o(A,B):C	logical 'or'	A or B → C
@n(A):C	logical negation	not A → C

If @a and @1 are defined for a given type, the aggregate operator `AND` can be generated by the Mosel compiler. The same generation occurs for the aggregate operator `OR` if @0 and @o are defined.

A and *B* (if of an external type) must be deleted by the operator.

1.4.4.4 Comparators

Operator	Operation	Return value	Remarks
@<(A,B):C	strictly less	A<B → C	implied by @n(C):C with @g(A,B):C
@>(A,B):C	strictly greater	A>B → C	implied by @n(C):C with @l(A,B):C
@l(A,B):C	less or equal	A ⩽ B → C	implied by @n(C):C with @>(A,B):C
@g(A,B):C	greater or equal	A ⩾ B → C	implied by @n(C):C with @<(A,B):C
@=(A,B):C	equality	A=B → C	implied by @n(C):C with @#(A,B):C
@#(A,B):C	difference	A ≠ B → C	implied by @n(C):C with @=(A,B):C

A, *B* and *C* may be of any type. If *C* is of an external type, the operator is therefore a constructor. If a comparator is not defined but the indicated complementary and the negation exist, Mosel constructs the missing operator. The compiler will also generate the operators returning a boolean result if both *A* and *B* are of the same type and the corresponding `compare` function is fully implemented (cf Section 1.2.3).

1.4.4.5 "is_" operators

Operator	Operation	Return value
@e(B):C	SOS type 1	B is_sos1 → C
@t(B):C	SOS type 2	B is_sos2 → C
@f(A):C	free	A is_free → C
@c(A):C	continuous	A is_continuous → C
@i(A):C	integer	A is_integer → C
@b(A):C	binary	A is_binary → C
@p(A,B):C	partial integer	A is_partint B → C
@s(A,B):C	semi continuous	A is_semcont B → C
@r(A,B):C	semi continuous integer	A is_semint B → C

A, *B* and *C* may be of any type. If *C* is of an external type, the operator is therefore a constructor. *A* is always a reference to an existing object and *B* can be any expression.

1.4.4.6 Assignment

Operator	Operation	Return value	Remarks
@:(C,A)	direct assignment	C:=A	
@M(C,A)	subtractive assignment	C-=A	implied by @:(C,A) with @-(C,A):C
@P(C,A)	additive assignment	C+=A	implied by @:(C,A) with @+(C,A):C

A must be deleted by the operator. The compiler will use the copy function of the type (see Section 1.2.3) to implement a direct or additive assignment (between 2 objects of the same type) if the corresponding operators are not defined or if *C* requires to be duplicated.

1.4.4.7 "As statement" operator

Operator	Operation
@_(A)	expression A is accepted as statement

A must be deleted by the operator.

This operator is used when an expression can be used in place of a statement (normally, an expression must be either assigned to an identifier or employed as parameter for a routine). Mosel implements this operator on linear expressions. For instance, the expression '`x <= 10`' can be assigned to an identifier of type `linctr` or used as is in place of a statement.

1.5 Defining services

Services are declared in the table of services (see Section 1.2.4). Each entry of the table is the pair (*service code, service implementation*) (a pointer). This section describes the available codes together with the functionality the user has to provide in order to implement the corresponding service.

1.5.1 Service "Reset"

```
XPRM_SRV_RESET: void *reset(XPRMcontext ctx, void *libctx, int version)
```

During the execution of a model, a module may have to store some data that is specific to this particular session. This information is called its *context of execution*. Each function of the module is always called with 2 contexts: the first one is the Mosel execution context and the second one the module context. The service `XPRM_SRV_RESET` is used to handle this module context: when the execution of the model starts, this function is called with a Mosel context and the constant `NULL` as its parameters. At this call, the `reset` function must return a pointer that will be used as the module context for the following calls to functions of the module.

When the model is reset (either before a new execution or before deleting it from memory) the `reset` function is called again but with the pointer it returned after its first execution. This time, the `reset` function has to release the resources used by the module context.

The last parameter sent to this function is the version number required by the running model. This information may be useful if the module can emulate different versions: from this function it may build an appropriate context depending on the version requested.

1.5.2 Service "Module priority"

```
XPRM_SRV_PRIORITY: int priority
```

This service may be used to define a priority value for the module. Modules with lower priority values are initialised before those with an higher priority. The resetting (*i.e.* call to the `onexit` function and second call to the `reset` function) is performed in the reverse order of the initialisation. The default priority value is `0`, specifying a new value requires the use of the `XPRM_MKPRIORITY` macro (*e.g.* `XPRM_MKPRIORITY(100)`).

1.5.3 Service "Unload"

```
XPRM_SRV_UNLOAD: void unload(void)
```

Mosel may decide to unload a module when it is not required any more. In some cases, a module has to release some resources it has allocated during its initialization (for instance, it uses a licensed software and has to release a license or some global data). For this purpose, this function is called just before Mosel unloads the module.

1.5.4 Service "Check Version"

```
XPRM_SRV_CHKVER: int chkvers(int requested_version)
```

The convention for module version numbers is to use a code version with 3 numbers (*major, minor, release*). When loading a module at runtime, Mosel checks if the obtained module is compatible with the one requested by the model (at compile time, the version numbers of all used modules are stored in the BIM file). A module version A can be used in place of module version B if *major(A) = major(B)* and *minor(A)* $\geqslant$ *minor(B)*. With the "check version" service a module can change this default behavior. This may be useful if, for instance, a module can emulate the behaviour of an older version. The parameter `requested_version` is the version number expected by the model to be able to run. If this function returns 0, the module is accepted; otherwise it is rejected due to the incompatibility in version numbers. As an alternative to this function the service `XPRM_SRV_COMPAT` can be defined (see 1.5.23).

1.5.5 Service "Find Parameter"

```
XPRM_SRV_PARAM: int findparm(const char *pname, int *type, int why,
                             XPRMcontext ctx, void *libctx)
```

This function is used to translate a parameter name into an internal code. This code is then used at run time by the Mosel Virtual Machine for calling the special routines `getparam` and `setparam`. If the function returns a value smaller than 0, the parameter named `pname` is not supported by the module; otherwise, the value returned is the code expected by `getparam` and `setparam`. Moreover, the function `findparm` has to set the paramater type to the type of this parameter. The parameter type is bit encoded using the type constants (`XPRM_TYP_INT`, `XPRM_TYP_REAL`, `XPRM_TYP_STRING`, `XPRM_TYP_BOOL`) plus the access rights (`XPRM_CPAR_READ`, `XPRM_CPAR_WRITE`): a parameter tagged `XPRM_CPAR_READ` can be accessed with the `getparam` function and a parameter tagged `XPRM_CPAR_WRITE` can be set using `setparam`.
The two last arguments passed to this function are usual execution contexts while the third one indicates how the function is used: argument `why` is `XPRM_FNDP_MCREAD` (0) when the compiler looks for a parameter to be read (with `getparam`); it is `XPRM_FNDP_MCWRITE` (1) when the compiler requires a parameter for writing (call to `setparam`). In both cases the module context `libctx` is `NULL`. When the execution of the model is starting and module parameters are set via the model parameter string this function is called with a value of `XPRM_FNDP_RTWRITE` (2) for `why` (before a call to `setparam`). During execution when another module requests the value of a parameter using `getdsoparam` this function is called with a value of `XPRM_FNDP_NIREAD` (3) for `why`. Finally, `why` takes the value `XPRM_FNDP_RTREAD` (4) when the function is used after execution by the library function `XPRMgetdsoparam`.
This service must be defined if the module provides control parameters.

1.5.6 Service "List of Parameters"

```
XPRM_SRV_PARLST: void *nextpar(void *ref, const char **name, const char **desc,
                               int *type)
```

This service is used to display the list of parameters provided by the module (for instance by using the command `examine` of the command line interpreter). It is called repeatedly until it returns `NULL`. The

first argument is a pointer in the internal table of parameters (handled by the module). When this pointer is NULL, the function has to return the first parameter. The information about the parameter is its name, a textual description desc of its meaning (may be an empty string) and its type (using the same encoding as for the service XPRM_SRV_PARAM). The function must return a pointer to the next entry in the list of parameters that can be used as input for the next call to this function. When the information about the last parameter of the module has been returned, the return value must be NULL.

1.5.7 Service "Inter-Module Communication Interface"

```
XPRM_SRV_IMCI: void *imci
```

This service may be used to implement communication between modules at the C language level. The value of this service (a pointer) is returned by the function getdsoctx. Typically, this pointer references a table of functions which are entry points to the module.

1.5.8 Service "Module Dependency List"

```
XPRM_SRV_DEPLST: const char *deplst[]
```

This service defines a table of modules that are automatically loaded when compiling models using this module. Note that this service is used only at compilation time and the corresponding modules will not be loaded during execution if they are not effectively required by the model. Every entry of this table is the name of a module as it is used with the uses directive. The last entry of the list must be NULL.

1.5.9 Service "IO Driver List"

```
XPRM_SRV_IODRVS: XPRMiodrvtab iodrvs[]
```

This service defines the table of IO drivers implemented by this module. Every entry of this table is a pair (*driver name, table of IO functions*). The driver name corresponds to the identifier to be used in file names and the table of functions lists operations supported by the driver (Section 1.6). The last entry of the list must be the pair (NULL, NULL).

1.5.10 Service "On Exit"

```
XPRM_SRV_ONEXIT: void onexit(XPRMcontext ctx, void *libctx, int status)
```

This service may be useful when some clean up operations have to be performed after the model has been run. The *onexit* function is called just before the end of the execution of the model if the service *reset* is implemented and has succeeded (*i.e.* the module context libctx is not NULL). The status provided here is the execution status of the model: it indicates why processing is terminating.

1.5.11 Service "Check Restrictions"

```
XPRM_SRV_CHKRES: int chkres(int restr)
```

This service must be defined for the module to be loaded when Mosel is running in restricted mode (refer to the Mosel Reference Manual for further details). The *chkres* function is called just after the module has been initialised such that it can check whether it supports the active restrictions: a return value of 0 indicates that the module will observe the stated restrictions; any other value will cause the module to be unloaded.

The restrictions are passed to this routine through the restr parameter as a bit encoded integer. The following restrictions may be set:

XPRM_RESTR_NOWRITE Disable write access

`XPRM_RESTR_NOREAD` Disable read access

`XPRM_RESTR_NOEXEC` Disable routines allowing to execute external commands

`XPRM_RESTR_WDONLY` Restrict disk access to current working directory

`XPRM_RESTR_NOTMP` Disable temporary directory

`XPRM_RESTR_NODB` Disable database access

The Mosel file management routines (like `fopen`) already enforce read/write restrictions for the files they handle. It is the responsiblity of the module to implement the restrictions when it uses any other file management routines. For instance, accessing directly a file using a system routine should only be done after having checked the validity of the file name with `pathcheck`; starting a separate process to execute an external command should be disabled if the restriction `NOEXEC` is active etc.

1.5.12 Service "Update Version"

```
XPRM_SRV_UPDVERS: void updvers(int event, int what, int *version)
```

This service is used by the Mosel compiler only: the function is called whenever a feature of the module (routine,type or parameter) is used by the model being compiled. The first argument indicates why the function is called and how to interpret the second argument (`what`). Possible values are:

`XPRM_UPDV_INIT` module loaded for compilation of a model (`what=0`) or a package (`what=1`)

`XPRM_UPDV_FUNC` function code `what` required

`XPRM_UPDV_TYPE` module type code `what` required

`XPRM_UPDV_GPAR` control parameter code `what` required for `getparam`

`XPRM_UPDV_SPAR` control parameter code `what` required for `setparam`

`XPRM_UPDV_ENDP` parser has finished analysing the file. Argument `what` is 0 if this operation succeeded

The last argument is a pointer to the current version number for this module that will be stored in the BIM file after compilation: the function may update this version number depending on the requirements of the model.

When the function is called for the first time (`XPRM_UPDV_INIT`) the version number may be 0: in this case, it must be changed to the minimum version supported by the module (that must be accepted by the service `CHKVER`). When the function is called for the last time (`XPRM_UPDV_ENDP`), changing the version number will make the parser fail.

1.5.13 Service "Implied Dependency List"

```
XPRM_SRV_IMPLST: const char *implst[]
```

This service defines a table of modules that imply the use of this module. At compile time, the module is added to the dependency list (see Section 1.5.8) of each of the listed modules. Every entry of this table is the name of a module as it is used with the `uses` directive. The last entry of the list must be `NULL`.

1.5.14 Service "Annotations"

```
XPRM_SRV_ANNOT: const char *anns[]
```

This service defines a table of global annotations. These annotations are added to any model using the module. Note that when compiling a package only `"mc."` annotations are processed. In the table an annotation is represented by 2 entries: the annotation name followed by its value (for instance `{"mc.def","myann global",NULL}`). The last entry of the list must be `NULL`.

1.5.15 Service "DSO stream"

```
XPRM_SRV_DSOSTRE: void* fct_open(XPRMmodel model, int mode, char *opts, char *params,
                          int (*fct_data)(void *fctx, char *buf, int len),
                          int (*fct_close)(void *fctx),
                          char *errmsg, int errmsglen)
```

This service defines a function to open a stream to the module from a remote instance. The function is called when a remote instance opens the file `"mcmd:dsostream dsoname parameters"` where `dsoname` is the name of the target module. The stream to create is characterised by a model reference (that may be `NULL`), the open mode, the `mcmd` opening options and the remaining parameters of the open string. If the operation is successful the function must return a non-null context pointer and update the parameter `fct_data`: for an output stream it will be called whenever data is available for reading (any return value smaller than `len` is interpreted as an error); for an input stream it will be used when the remote instance is expecting data: in this case up to `len` bytes have to be copied into `buf` and the return value is the amount of data sent. A return value of 0 indicates an end of stream and a negative value an error. Optionally `fct_close` may also be set: it will be used when the stream is closed to release its resources. In case of error the function must return NULL and copy an error message into the provided buffer.

1.5.16 Service "Required types"

```
XPRM_SRV_REQTYPS: const char *reqtyp[]
```

This service defines a table of native types that are required by a module and that must be available in order to be able to execute models using this module. Every entry of this table is the name of a type (*e.g.* `"text"`) optionally prefixed by the name of the module defining it (*e.g.* `"mmsystem.text"`). The last entry of the list must be `NULL`. Note that the modules defining the required types must be first listed in the dependency list (see Section 1.5.8).

1.5.17 Service "Provider"

```
XPRM_SRV_PROVIDER: const char *prov
```

This service defines the identity of the provider for this module as returned by the function `getdsoprop` for the property `XPRM_PROP_SYSCOM`.

1.5.18 Service "Namespace groups"

```
XPRM_SRV_NSGRP: const char **nsgrps
```

This service may be used to define namespace groups from a module similar to those created via the `nsgroup` contruct in the language. The data of this service is an array of constant strings terminated by a `NULL` reference. Each record consists in two strings: the first one is the name of the namespace and the second defines the list of packages belonging to the group (package names separated by comas without space). If this list is empty the namespace can be used by any package.

1.5.19 Service "Memory use"

```
XPRM_SRV_MEMUSE: size_t memuse(XPRMcontext ctx, void *libctx,
                          void *ref, int code)
```

This routine is used by Mosel to report memory usage of the entire module or of an instance of one of its type. When it is called with `NULL` for `ref` and `0` for `code` the function is expected to return the total amount of memory the module has allocated so far for the running model. When `ref` is not `NULL` it is a reference to an entity of type `code` (internal code as specified in the table of types, see Section 1.2.3) and the function has to return the amount of memory used by this entity. If the information is not available `-1` must be returned.

1.5.20 Service "Static module"

```
XPRM_SRV_STATIC: XPRM_STATIC_INST|XPRM_STATIC_PROC
```

If this service is defined and its associated value is the constant `XPRM_STATIC_INST` the module is handled as a static module: it will not be unloaded (even if it is no longer required) until the Mosel library itself is released. If the value is `XPRM_STATIC_PROC` the module will never be unloaded (although the service *unload* will be called when the Mosel library is released).

1.5.21 Service "Get array indices"

```
XPRM_SRV_ARRIND: int getarrind(XPRMcontext ctx, void *libctx,
                           void *ndx, int code,
                           XPRMarray arr, int *indices, int op)
```

This routine is used when Mosel needs to access an array dereferenced via an *array indexer*: the function is expected to return in `indices` the tuple of indices corresponding to the data stored in `ndx` for the array `arr`. The parameter `ndx` is a reference to an entity of type `code` (internal code as specified in the table of types, see Section 1.2.3). The last parameter, `op`, identifies the kind of access to be performed: `XPRM_OPNDX_EXISTS` (call to `exists`), `XPRM_OPNDX_DEL` (call to `delcell`), `XPRM_OPNDX_GET` (get the value of the cell) or `XPRM_OPNDX_SET` (set the value of the cell). The returned information must be suitable for using a function like `getarrval` for instance. The function must return `0` on success, a positive value in case of an out of range and a negative value to notify an error.

1.5.22 Service "Deprecation List"

```
XPRM_SRV_DEPREC: unsigned int deprec[]
```

This service consists in a list of pairs of unsigned integers sorted in ascending order on the first component of each pair. The last entry of the list must be `0,0`. The first component of each pair is interpreted as a subroutine code (see Section 1.2.2) and the second as a version number encoded using `XPRM_VERSION`: this is the version number from which the corresponding routine is deprecated. The compiler will use this information for generating deprecation warning messages.

1.5.23 Service "Minimum compatible version"

```
XPRM_SRV_COMPAT: int versnum
```

This service may be used to define the smallest version compatible with the current module. This can be used as a replacement for the service `CHKVER`: a version number is considered compatible if it is between the value of this service and the current version of the module. The `XPRM_MKCOMPAT` macro can be used to initialise this service (*e.g.* `XPRM_MKCOMPAT(1,0,0)`).

1.6 Defining IO drivers

Mosel accesses files through IO drivers: a driver provides a set of functions implementing basic IO operations (open, close, read a block, write a block). As a consequence, any kind of data source may be exposed as a file (or *stream*) in the Mosel environment as long as it can be accessed through the basic

operations listed above.
IO drivers are declared using the XPRM_SRV_IODRVS service which points to a table of drivers. Each driver is identified by its name and a table of functions. This table of functions is of the form (*operation code*, *function*) which associates to each code a function performing the operation (except for operation XPRM_IOCTRL_INFO which is only a text string describing the driver). The last record of this table must be the pair (0, NULL).

Example:

```
static XPRMiofcttab mydrv_fcts[]=
        {
         {XPRM_IOCTRL_OPEN, mydrv_open},
         {XPRM_IOCTRL_CLOSE, mydrv_close},
         {XPRM_IOCTRL_READ, mydrv_read},
         {XPRM_IOCTRL_WRITE, mydrv_write},
         {XPRM_IOCTRL_INFO, "mydrv description"},
         {0,NULL}
        };

static XPRMiodrvtab iodrivers[]=
        {
         {"mydrv", mydrv_fcts},
         {NULL, NULL}
        };

static XPRMdsoserv tabserv[]=
        {
         {XPRM_SRV_IODRVS, iodrivers}
        };
```

In addition to basic operations, a driver may provide implementations for the initializations block, file removal and file renaming. The “initializations from/to” routines get direct access to Mosel internal data and can therefore work at the binary level. The other file operations may be required for resource management.

All functions performing IO operations are sent a Mosel execution context. However, this context may be NULL if the stream is used outside of the execution of a model (for instance for compiling a model source). It is not necessary to provide an implementation for all of the possible operations. However, the *open* function and one of the data transfer routines are mandatory (*read* or *write* or *initfrom* or *initto*).

1.6.1 Operation “Open Stream”

```
XPRM_IOCTRL_OPEN: void *open(XPRMcontext ctx, int *mode, const char *fname,
                             unsigned int *enc, int *bufsize)
```

This function is called to open a stream associated to the given file name (the provided file name does not include driver name). Parameter mode is a pointer to the open mode. This value is bit encoded and the following bits may be set:

XPRM_F_BINARY	open in binary mode (default is text mode)
XPRM_F_WRITE	open for writing (default is for reading)
XPRM_F_APPEND	when open for writing, do not reset file but append data to the end of the original file
XPRM_F_ERROR	when open for writing, stream will be used as an error stream.
XPRM_F_LINBUF	when open for writing, stream is line buffered: buffer is flushed after each line (by default streams are flushed when full or when an explicit flush is executed)
XPRM_F_INIT	stream open for an ‘initialisations’ block

XPRM_F_SILENT	stream open in silent mode (error messages are not displayed)
XPRM_F_DELCLOSE	delete file after the stream has been closed

These modes may be combined (for instance XPRM_F_WRITE | XPRM_F_LINBUF) so testing the mode should be done using masks. The open function may alter the mode by changing some bits (LINBUF, INIT and SILENT; others are ignored) and by using bits 16 to 31 that are not used by Mosel although saved with the stream's mode (which can be retrieved later using function fgetinfo and is provided to the close function). Note the particular meaning of INIT: if the function is called with this bit set, the stream will be used for an initialisations block. Resetting this bit indicates that the driver provides an handler for this operation and expects that Mosel will use this handler. Otherwise, even if the driver publishes the operation, the default procedures are used.
Mosel saves the current input and output streams after execution of the open function if new files have been opened using fopen. These current streams are restored when calling operations "close", "read", "skip", "write", "init from" and "init to". This is useful when the stream implements a filter (*i.e.* it preprocesses some data actually stored in another file).

The parameter enc is the encoding used when the file is open in text mode (*i.e.* flag XPRM_F_BINARY is not set). If no encoding has been explicitly stated in the file name (using the "enc:" prefix) the value of this variable is XPRM_FE_ENCDEF (that is equivalent to UTF-8). Otherwise this value consists in 2 16-bits words: the first part is the encoding ID as used by the XPRNLS library (please refer to the XPRNLS Reference Manual for further explanation), it can be extracted using the mask XPRM_FE_MSK_ENC. The second part of the value is used to record the encoding options (XPRM_FE_STRICT, XPRM_FE_UNIX, XPRM_FE_DOS, XPRM_FE_BOM, XPRM_FE_NOBOM). The function may modify the encoding or disable it entirely by switching to binary mode.

The parameter bufsize is the size (in kilobytes) of the internal buffer allocated for the stream. A driver can change this setting that must remain between 2 (the default value) and 64.

The return value of this function will be used as input for other IO functions. A return value of NULL indicates a failure: in this case, the function setioerrmsg may be used to set a descriptive text for the error that is displayed after the function returns (if the stream is not open in silent mode).

1.6.2 Operation "Close Stream"

```
XPRM_IOCTRL_CLOSE: int close(XPRMcontext ctx, void *stream, int mode)
```

This optional function is called to close a stream previously open using the corresponding open function (after it has been flushed if it is an output stream). The second parameter is the pointer returned by a successful execution of the "open" function and the third parameter is the current mode of the stream. If an error has been detected on this stream, the bit XPRM_F_IOERR is set.

This function must return 0 in case of success, all other values are interpreted as error codes. In the later case, the function setioerrmsg may be used to set a descriptive text for the error that is displayed after the function returns (if the stream is not open in silent mode).

1.6.3 Operation "Read Block"

```
XPRM_IOCTRL_READ: long read(XPRMcontext ctx, void *stream, void *buffer,
                            unsigned long size)
```

This function is used to load the stream buffer open for reading. Parameters buffer and size define location and size of the buffer associated to the stream. This function returns the number of bytes actually copied into the buffer. A value of 0 characterises an end of file and a negative value is interpreted as an error. In the later case, the function setioerrmsg may be used to set a descriptive text for the error that is displayed after the function returns (if the stream is not open in silent mode).

1.6.4 Operation "Skip Block"

```
XPRM_IOCTRL_SKIP: int skip(XPRMcontext ctx, void *stream, int size)
```

This optional function is used to skip a block of data from an input stream. If the operation is successful a positive value has to be returned; the special value `-2` indicates that the corresponding stream does not support the operation. In this case the system will read (and discard) the number of bytes to be skipped. A negative value is interpreted as an error and the function `setioerrmsg` may be used to set a descriptive text for the error that is displayed after the function returns (if the stream is not open in silent mode).

1.6.5 Operation "Write Block"

```
XPRM_IOCTRL_WRITE: long write(XPRMcontext ctx, void *stream, void *buffer,
                              unsigned long size)
```

This function is used to flush the stream buffer open for writing. Parameters `buffer` and `size` define location and size of the buffer associated to the stream. This function must return a positive value if successful - any negative value or zero is interpreted as an error status. In the later case, the function `setioerrmsg` may be used to set a descriptive text for the error that is displayed after the function returns (if the stream is not open in silent mode).

1.6.6 Operation "Initializations From"

```
XPRM_IOCTRL_IFROM: int initfrom(XPRMcontext ctx, void *stream, int nbrec,
                                const char **labels, int *types,
                                XPRMalltypes **adrs, int *nbread)
```

This function implements an `initializations from` block. For this function to be called, the *open* function must reset the `INIT` bit of the open mode. The information provided describes the block to be processed: number of records `nbrec`, for each record *i* its label `labels`*[i]*, its type `types`*[i]* and its address `adrs`*[i]* (direct address for Mosel objects and indirect address for objects of external types). If a label is associated to a tuple of arrays, the corresponding address is a list of arrays.

If auto finalization is in use (this is indicated by the control parameter `"autofinal"`, see `getparam`), sets must be finalized using `fnlset` just after they have been initialized. Also, in order to handle the case of index sets of implicit dynamic arrays, it is necessary to call `beginarrinit` before the initialization of each array and `endarrinit` after the process has completed.

During its execution the function should set to `NULL` each entry *i* of the `adrs` array for which reading has been completed and store in `nbread`*[i]* the number of successfully input lines (*i.e.* when reading a tuple of arrays, 1 line means 1 value for each array). The return value of *initfrom* is interpreted as the number of records that have not been successfully read in (*i.e.* 0 for success). In case of an IO error the function should return -1.

1.6.7 Operation "Initializations To"

```
XPRM_IOCTRL_ITO: int initto(XPRMcontext ctx, void *stream, int nbrec,
                            const char **labels, int *types, XPRMalltypes **adrs)
```

This function implements an `initializations to` block. For this function to be called, the *open* function must reset the `INIT` bit of the open mode. The information provided describes the block to be processed: number of records `nbrec`, for each record *i* its label `labels`*[i]*, its type `types`*[i]* and its address `adrs`*[i]* (direct address for Mosel objects and indirect address for objects of external types). If a label is associated to a tuple of arrays, the corresponding address is a list of arrays.

During its execution the function should set to `NULL` each entry *i* of the `adrs` array for which saving has been completed. The return value of *initto* is interpreted as the number of records that have not been

successfully saved (*i.e.* 0 for success). In case of an IO error the function should return -1.

1.6.8 Operation "Remove File"

```
XPRM_IOCTRL_RM: int remove(XPRMcontext ctx, const char *todel)
```

This function is called to *remove* (or delete) a file. Parameter `todel` is the name of the file to be deleted (IO driver name is not included). Mosel does not check whether the file is currently open: depending on the driver this may make the operation impossible and should be checked by the function if necessary. The convention for return values is as follows: 0 indicates a success, 1 if the file cannot be accessed and 4 if the operation is not possible (*e.g.* file open or protected).

1.6.9 Operation "Move File"

```
XPRM_IOCTRL_MV: int move(XPRMcontext ctx, const char *src, const char *dst)
```

This function is called to *move* (or rename) a file. Parameters `src` and `dst` are names of the source and destination files (IO driver name is not included). Mosel does not check whether the files are currently open: depending on the driver this may make the operation impossible and should be checked by the function if necessary. The convention for return values is as follows: 0 indicates a success, 1 if the source file cannot be accessed, 2 if destination file cannot be open for writing, 3 if the copy failed and 4 if the source cannot be removed after copy. The special value -1 can be used to tell Mosel to perform the operation by deleting the file after having made a copy of it.

1.6.10 Operation "File Size"

```
XPRM_IOCTRL_SIZE: size_t fsize(XPRMcontext ctx, const char *file)
```

This function is called to retrieve the size of a file. Parameter `file` is the name of the file to be considered (IO driver name is not included). Mosel does not check whether the file is currently open: depending on the driver this may make the operation impossible and should be checked by the function if necessary. On error `(size_t)-1` must be returned.

1.7 Static modules

Modules are usually implemented as dynamic libraries: modules are represented as files that Mosel loads when required. It is also possible to embed a module into a program using the Mosel Libraries. In this case, the module must be *registered* in Mosel before it is used by any model. This registration is performed by a call to the function `XPRMregstatdso` which receives as parameters the name of the module (this is the name one uses to request the module in a `uses` directive in the model) and the reference to the initialization function of this module. The registration function initializes immediately the module by calling its initialization function and records the module as a static module (it cannot be unloaded). Note that the type of an initialization function of a static module is `int` instead of `DSO_INIT`.

Example:

```
#include "xprm_mc.h"
#include "xprm_ni.h"
int mymodule_init(XPRMnifct nifct, int *interver, int *libver, XPRMdsointer **interf);
...
int main()
{
 XPRMinit();
 XPRMregstatdso("mymodule", mymodule_init);
 /* Now the module "mymodule" is available */
 XPRMcompmod(NULL, "test.mos", NULL, NULL);
 ...
}
```

```
/* Functions of the module must be included in the program */
...
```

CHAPTER 2

Functions of the Native Interface

During its initialization, the module receives a pointer of type `XPRMnifct`. This entity is the address of the table of functions of the Native Interface. Through this table the module can call Mosel functions in order to get information on the objects currently handled by the running model but also change values of these objects as well as call functions and procedures of the model.

Example:

```
static XPRMnifct mm;
...
mm->dispmsg(ctx,"error message\n")   /* Display an error message */
...
```

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

2.1 List access

addellist

Purpose

Add an element to a list.

Synopsis

```
int addellist(XPRMcontext ctx, XPRMlist list, int type, XPRMalltypes *elt);
```

Arguments

`ctx`	Mosel's execution context
`list`	Reference to a list
`type`	Type of the element to add
`elt`	The element to add

Return value

0 if successful, a positive value otherwise.

Further information

This function appends a new element to the end of a list: it becomes the last element of the list. The function fails if the list is not dynamic or the element is not of a compatible type.

Related topics

`insellist`, `regstring`.

insellist

Purpose

Insert an element into a list.

Synopsis

```
int insellist(XPRMcontext ctx, XPRMlist list, int type, XPRMalltypes *elt);
```

Arguments

`ctx`	Mosel's execution context
`list`	Reference to a list
`type`	Type of the element to add
`elt`	The element to add

Return value

0 if successful, a positive value otherwise.

Further information

This function inserts a new element at the beginning of a list: it becomes the first element of the list. The function fails if the list is not dynamic or the element is not of a compatible type.

Related topics

`addellist`, `regstring`.

getlistsize

Purpose

Get the size of a list.

Synopsis

```
unsigned int getlistsize(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

`getlisttype`.

getlisttype

Purpose

Get the type of a list.

Synopsis

```
int getlisttype(XPRMlist list);
```

Argument

`list` Reference to a list

Return value

List type.

Further information

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

`getlistsize`,`getnextlistelt`.

getnextlistelt

Purpose

Get the next element of a list.

Synopsis

```
void *getnextlistelt(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 `getnextlistelt`.

Further information

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

Related topics

`getlisttype`,`getprevlistelt`.

getprevlistelt

Purpose

Get the previous element of a list.

Synopsis

```
void *getprevlistelt(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 `getnextlistelt`.

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 `getlisttype` if all elements have the same type.

Related topics

`getlisttype`,`getnextlistelt`.

resetlist

Purpose

Remove all elements of a list.

Synopsis

```
int resetlist(XPRMcontext ctx, XPRMlist list);
```

Arguments

`ctx` Mosel's execution context

`list` Reference to a dynamic list

Return value

0 if successful, a positive value otherwise.

Further information

The function resets a list by removing all elements it contains. It is not possible to reset a constant or finalised list.

2.2 Set access

addelset

Purpose

Add an element to set.

Synopsis

```
int addelset(XPRMcontext ctx, XPRMset set, XPRMalltypes *elt, int *ndx);
```

Arguments

`ctx`	Mosel's execution context
`set`	Reference to a set
`elt`	The element to add
`ndx`	Returned index of the element added

Return value

0 if successful, a positive value otherwise.

Further information

This function adds a new element to a set. The function fails if the set is not dynamic and the element is not already contained in the set. On success the index of the element is returned in `ndx`. Note that when applied to a range set, the index value is the inserted value itself.

Related topics

`getelsetval`, `getelsetndx`, `regstring`.

fnlset

Purpose

Finalize a set.

Synopsis

```
void fnlset(XPRMcontext ctx, XPRMset set);
```

Arguments

`ctx` Mosel's execution context

`set` Reference to a set

Further information

This function as the same effect as the `finalise` procedure of the Mosel language.

resetset

Purpose

Remove all elements of a set.

Synopsis

```
int resetset(XPRMcontext ctx, XPRMset set);
```

Arguments

`ctx` Mosel's execution context

`set` Reference to a dynamic set

Return value

0 if successful, a positive value otherwise.

Further information

The function resets a set by removing all elements it contains. It is not possible to reset a constant or finalised set.

getelsetndx

Purpose

Get the index of a set element.

Synopsis

```
int getelsetndx(XPRMcontext ctx, XPRMset set, XPRMalltypes *elt);
```

Arguments

`ctx` Mosel's execution context

`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. If applied to a range set, the returned value is always `elt->integer`.

Related topics

`getfirstsetndx`, `getlastsetndx`, `getelsetval`.

getelsetval

Purpose

Get the value of an element of a set.

Synopsis

```
XPRMalltypes *getelsetval(XPRMcontext ctx, XPRMset set, int ind,
      XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`set`	Reference to a set
`ind`	Index number
`value`	Pointer to an area where the result is returned

Return value

The fourth argument or `NULL`.

Further information

1. 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`.
2. When getting element values to enumerate the content of a dynamic array with several dimensions, the use of `mapset`/`unmapset` may increase significantly the efficiency of `getelsetval`.

Related topics

`mapset`, `getelsetndx`.

mapset

Purpose

Modify the structure of a set for fast element retrieval.

Synopsis

```
void mapset(XPRMcontext ctx, XPRMset set);
```

Arguments

`ctx`	Mosel's execution context
`set`	Reference to a set

Further information

1. This function modifies the internal representation of a set in order to improve the efficiency of function `getelsetval`. After this function has been called, the set must not be modified until `unmapset` is used.
2. This function is effective only on dynamic general sets, it is however safe to use it on other type of sets (range and/or constant sets).
3. If the function is used several times on a given set, `unmapset` has to be called the same number of times to restore the set in its initial state.
4. If several sets are *mapped*, it is preferable to call `unmapset` in reverse order to minimise memory fragmentation (*i.e.* `mapset(ctx,s1); mapset(ctx,s2)` ... `unmapset(ctx,s2); unmapset(ctx,s1)`).

Related topics

`unmapset`, `getelsetval`.

unmapset

Purpose

Restore a mapped set to its initial state.

Synopsis

```
void unmapset(XPRMcontext ctx, XPRMset set);
```

Arguments

`ctx`	Mosel's execution context
`set`	Reference to a set

Further information

1. This function performs the inverse operation of `mapset` that has to be done before the set can be modified again.
2. This function is effective only on sets for which `mapset` succeeded: applying it to all other types of sets is a no operation.

Related topics

`mapset`, `getelsetval`.

getfirstsetndx

Purpose

Get the index of the first element in a given set.

Synopsis

```
int getfirstsetndx(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Index of the first element in the set.

Further information

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

Related topics

`getsetsize`, `getlastsetndx`.

getlastsetndx

Purpose

Get the index of the last element in a set.

Synopsis

```
int getlastsetndx(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Index of the last element in the set.

Further information

1. 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.
2. It is recommended to test whether the set is not empty (using function `getsetsize`) before calling this function.

Related topics

`getfirstsetndx`, `getsetsize`.

getsetsize

Purpose

Get the size of a set.

Synopsis

```
unsigned int getsetsize(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

`getsettype`.

getsettype

Purpose

Get the type of a set.

Synopsis

```
int getsettype(XPRMset set);
```

Argument

`set` Reference to a set

Return value

Bit encoded set type.

Further information

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

Related topics

`getsetsize`.

isinset

Purpose

Check if an element is contained in a set.

Synopsis

```
int isinset(XPRMcontext ctx, XPRMset set, XPRMalltypes *elt);
```

Arguments

`ctx` Mosel's execution context

`set` Reference to a set

`elt` Reference to the element

Return value

1 if the element is contained in the set, 0 otherwise.

Further information

This function checks whether an element is contained in a set.

2.3 Array access

`beginarrinit`	Begin an array initialization.	p. 65
`chkarrind`	Check whether an index tuple of an array is valid.	p. 49
`clsarrsrtndx`	Release the sorted index table of a hashmap array.	p. 50
`cmpindices`	Compare two index tuples.	p. 51
`delarrcell`	Delete an array entry.	p. 52
`endarrinit`	Terminate an array initialization.	p. 66
`existsarrentry`	Check whether a given entry in a sparse array has been created.	p. 53
`getarrdim`	Get the number of dimensions of an array.	p. 56
`getarrsets`	Get the index sets of an array.	p. 57
`getarrsize`	Get the size of an array.	p. 58
`getarrtype`	Get the type of an array.	p. 59
`getarrval`	Get the value of an array entry.	p. 60
`getfirstarrentry`	Get the list of indices of the first entry of an array.	p. 54
`getfirstarrtruentry`	Get the list of indices of the first true entry of an array.	p. 55
`getlastarrentry`	Get the list of indices of the last entry of an array.	p. 61
`getnextarrentry`	Get the list of indices of the next entry of an array.	p. 62
`getnextarrtruentry`	Get the list of indices of the next true entry of an array.	p. 63
`setarrval`	Set the value of an array entry.	p. 64

chkarrind

Purpose

Check whether an index tuple of an array is valid.

Synopsis

```
int chkarrind(XPRMarray array, const 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

`cmpindices`, `existsarrentry`.

clsarrsrtndx

Purpose

Release the sorted index table of a hashmap array.

Synopsis

```
void clsarrsrtndx(XPRMcontext ctx, XPRMarray array);
```

Arguments

`ctx` Mosel's execution context

`array` Reference to a hashmap array

Return value

1 if the index table was removed, 0 if no operation was performed and -1 if the array is not a hashmap array.

Further information

To enumerate cells of a hashmap array a sorted table of the indices of the array is created. This function can be used to release this table. Note that this table is automatically deleted when the structure of the array changes (*i.e.* a cell is added or removed).

cmpindices

Purpose

Compare two index tuples.

Synopsis

```
int cmpindices(int nbdim, const int ind1[], const 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

`chkarrind`.

delarrcell

Purpose

Delete an array entry.

Synopsis

```
int delarrcell(XPRMcontext ctx, XPRMarray array, const int indices[]);
```

Arguments

`ctx`	Mosel's execution context
`array`	Reference to an array
`indices`	`n`-tuple of indices where `n` is the number of dimensions of the array `array` (may be `NULL`)

Return value

0 if executed successfully, a positive value otherwise (the array is locked).

Further information

This function deletes the specified cell of an array: on a dynamic array the cell is effectively deleted, on a dense array the cell is reset (the cell remains unchanged if the type does not support this operation). When used with a `NULL` pointer as the set of indices the function deletes the entire array. In the case of a dense array the data will be reallocated after the first access to the array.

existsarrentry

Purpose

Check whether a given entry in a sparse array has been created.

Synopsis

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

Arguments

`array` Reference to an array

`indices` `n`-tuple of indices where `n` is the number of dimensions of the array `array`

Return value

`1` if the entry exists, `0` otherwise.

Further information

If an array is sparse its elements are not created at its declaration. This function indicates if a given element has been created.

Related topics

`chkarrind`.

getfirstarrentry

Purpose

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

Synopsis

```
int getfirstarrentry(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

`getfirstarrtruentry`, `getlastarrentry`, `getnextarrentry`.

getfirstarrtruentry

Purpose

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

Synopsis

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

Arguments

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

Return value

0 if executed succesfully, a positive value otherwise.

Further information

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

Related topics

`getfirstarrentry`, `getlastarrentry`, `getnextarrtruentry`.

getarrdim

Purpose

Get the number of dimensions of an array.

Synopsis

```
int getarrdim(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

`getarrsets`, `getarrsize`, `getarrtype`.

getarrsets

Purpose

Get the index sets of an array.

Synopsis

```
void getarrsets(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

`getarrdim`, `getarrsize`, `getarrtype`.

getarrsize

Purpose

Get the size of an array.

Synopsis

```
unsigned int getarrsize(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

`getarrdim`, `getarrsets`, `getarrtype`.

getarrtype

Purpose

Get the type of an array.

Synopsis

```
int getarrtype(XPRMarray array);
```

Argument

`array` Reference to an array

Return value

Type of the array.

Further information

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

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

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

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

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

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

Related topics

`getarrdim`, `getarrsets`, `getarrsize`.

getarrval

Purpose

Get the value of an array entry.

Synopsis

```
int getarrval(XPRMarray array, const 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

`setarrval`

getlastarrentry

Purpose

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

Synopsis

```
int getlastarrentry(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

`getfirstarrentry`, `getfirstarrtruentry`.

getnextarrentry

Purpose

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

Synopsis

```
int getnextarrentry(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

`getfirstarrentry`, `getfirstarrtruentry`, `getnextarrtruentry`.

getnextarrtruentry

Purpose

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

Synopsis

```
int getnextarrtruentry(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 `getnextarrentry`. With a dynamic array, this function returns the index tuple of the next true entry.

Related topics

`getfirstarrentry`, `getfirstarrtruentry`, `getnextarrentry`.

setarrval

Purpose

Set the value of an array entry.

Synopsis

```
int setarrval(XPRMcontext ctx, XPRMarray array, const int indices[],
      XPRMalltypes *value)
int setarrvalint(XPRMcontext ctx, XPRMarray array, const int indices[],
      int valint)
int setarrvalreal(XPRMcontext ctx, XPRMarray array, const int indices[],
      double valreal)
int setarrvalstr(XPRMcontext ctx, XPRMarray array, const int indices[],
      const char *valstr)
int setarrvalbool(XPRMcontext ctx, XPRMarray array, const int indices[],
      int valint)
```

Arguments

`ctx`	Mosel's execution context
`array`	Reference to an array
`indices`	`n`-tuple of indices where `n` is the number of dimensions of the array `array`
`value`	Reference to a value
`valint`	An integer value
`valreal`	A real value
`valstr`	A text string

Return value

0 if executed succesfully, a positive value otherwise.

Further information

1. These functions set the value of an array entry that corresponds to a given tuple of indices for the given array.
2. The first form of this function selects the right value type (integer, real, string or boolean) based on the type of the array. If applied to an array of any other type, the reference to the object is returned in the `value` parameter and the function returns 2.
3. When `setarrvalstr` is applied to an array of a type published by a module, it uses `dsotypfromstr` for performing the assignment.

Related topics

`getarrval`, `regstring`.

beginarrinit

Purpose

Begin an array initialization.

Synopsis

```
int beginarrinit(XPRMcontext ctx,XPRMarray array);
```

Arguments

`ctx` Mosel's execution context

`array` Reference to an array

Return value

0 if executed succesfully, a positive value otherwise.

Further information

1. This function should be called just before initializing an array (typically from within an *initializations from* routine): this allows to handle transparently the finalization of index sets when automatic finalization is in use and to transform implicit dynamic arrays into static arrays when possible.

2. After this routine has been applied to an array, the use of `getarrval` on this array is disabled until `endarrinit` is called.

3. The initialization procedure must be concluded by a call to `endarrinit` in order restore the array data structure.

Related topics

`endarrinit`.

endarrinit

Purpose

Terminate an array initialization.

Synopsis

```
int endarrinit(XPRMcontext ctx,XPRMarray array, int status);
```

Arguments

`ctx`	Mosel's execution context
`array`	Reference to an array
`status`	Current status

Return value

If `status` is non-zero then this value is returned, otherwise 0 if executed successfully or a positive value on failure.

Further information

This function must be called after the initialization of an array initiated with a call to `beginarrinit`.

Related topics

`beginarrinit`.

2.4 Module types access

`cmpval`	Compare two objects of the same external type.	p. 68
`copyval`	Perform an assignment between two objects of the same external type.	p. 69
`dsotypfromstr`	Initialise an object of a module or union type using a string.	p. 71
`dsotyptostr`	Get a string representation from a module or union type reference.	p. 70
`findattrdesc`	Find an attribute descriptor from its name.	p. 72
`getarrindices`	Get the tuple of indices corresponding to an array indexer.	p. 73
`getattr`	Get an attribute of an entity.	p. 74

cmpval

Purpose

Compare two objects of the same external type.

Synopsis

```
int cmpval(XPRMcontext ctx,int type, void *left,void *right);
```

Arguments

`ctx`	Mosel's execution context
`type`	Code of the external type and operation (*i.e.* `XPRM_COMPARE_*`)
`left`	Left operand of the comparison
`right`	Right operand of the comparison

Return value

Result of the operation or `XPRM_COMPARE_ERROR` in case of error.

Further information

This function calls directly the *compare* routine of the module (see Section 1.2.3). It is therefore recommended to check whether the type supports this functionality and the requested operation before using this function (see `gettypeprop`).

Related topics

`gettypeprop`.

copyval

Purpose

Perform an assignment between two objects of the same external type.

Synopsis

```
int copyval(XPRMcontext ctx,int type, void *dst,void *src);
```

Arguments

`ctx`	Mosel's execution context
`type`	Code of the external type and operation (*i.e.* `XPRM_CPY_*`)
`dst`	Entity to be assigned (must not be `NULL`)
`src`	Source entity

Return value

0 if successful, 1 otherwise.

Further information

1. This function calls directly the *copy* routine of the module (see Section 1.2.3). It is therefore recommended to check whether the type supports this functionality and the requested operation before using this function (see `gettypeprop`).

2. This routine can also be used with structured user defined types (like sets, arrays or records).

3. Instead of a code of an external type the argument `type` may be `XPRM_TYP_CTL`, `XPRM_STR_SET`, `XPRM_STR_LST` or `XPRM_STR_ARR` to perform copies between linear constraints, sets, lists or arrays. Source and destination entities must have the same type (the function does not check compatibility of its arguments).

Related topics

`gettypeprop`.

dsotyptostr

Purpose

Get a string representation from a module or union type reference.

Synopsis

```
int dsotyptostr(XPRMcontext ctx,int type, void *value, char *str,
      int size);
```

Arguments

`ctx`	Mosel's execution context
`type`	Code of the external type
`value`	Entity to convert
`str`	Destination buffer
`size`	Size of `str`

Return value

Size of the generated string (excluding the terminating null byte) , -1 in case of error or -2 if the feature is not supported by the type.

Further information

1. This function calls directly the *tostring* routine of the module (see Section 1.2.3). It is therefore recommended to check whether the type supports this functionality before using this function (see `gettypeprop`).
2. The returned length might be larger than `size-1`. In this case this return value is the minimum buffer size (not including the terminating null byte) required to generate the text representation and the destination string `str` is not populated.
3. The flag `XPRM_TFSTR_BIN` might be added to the type code in order to retrieve a binary representation of the object.
4. This function may also be applied to a union type. In this case the current value stored in the union is converted to its textual representation. The symbol `"?"` will be returned if the union is not defined, and `-2` will be returned if the value is not a scalar or if the corresponding type does not support conversion to string.

Related topics

`gettypeprop`, `dsotypfromstr`.

dsotypfromstr

Purpose

Initialise an object of a module or union type using a string.

Synopsis

```
int dsotypfromstr(XPRMcontext ctx,int type, void *ref,const char *str,const
      char **end);
```

Arguments

`ctx`	Mosel's execution context
`type`	Code of the external type
`ref`	Entity to initialise (must not be `NULL`)
`str`	Initialisation string
`end`	Reference to return a pointer after the last character used (can be `NULL`)

Return value

0 if successful, 1 in case of error or 2 if the feature is not supported by the type.

Further information

1. This function calls directly the *fromstring* routine of the module (see Section 1.2.3). It is therefore recommended to check whether the type supports this functionality before using this function (see `gettypeprop`).
2. The flag `XPRM_TFSTR_BIN` might be added to the type code in order to process a binary representation of the object. In this mode the `end` pointer must reference the first byte after the data buffer (such that `*end-str` is the size of the buffer).
3. This function may also be used to initialize a union type. In this case the conversion is performed as if the union was of its first compatible type (the value of the union is switched to this type if necessary). The operation will fail if this type cannot be initialized from a string.

Related topics

`gettypeprop`, `dsotyptostr`.

findattrdesc

Purpose

Find an attribute descriptor from its name.

Synopsis

```
XPRMattrdesc findattrdesc(XPRMcontext ctx, int type, const char **name, int
      *atype);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type number for the attribute
`name`	Name of the attribute

Return value

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

Further information

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

Related topics

`getattr`.

getarrindices

Purpose

Get the tuple of indices corresponding to an array indexer.

Synopsis

```
int getarrindices(XPRMcontext ctx,int type, void *ndx,XPRMarray arr, int
      *indices, int op);
```

Arguments

`ctx`	Mosel's execution context
`type`	Code of the indexer type
`ndx`	Reference to the indexer
`arr`	Array to be accessed
`indices`	Integer buffer to store the indices
`op`	Operation that will be performed on the array (*e.g.* `XPRM_OPNDX_GET`)

Return value

`0` if successful, a positive value in case of an out of range and a negative value for an error.

Further information

This function calls directly the *Get array indices* service of the module (see Section 1.5.21). It is therefore recommended to check whether the type is actually an array indexer before using this function (see `gettypeprop`).

Related topics

`gettypeprop`.

getattr

Purpose

Get an attribute of an entity.

Synopsis

```
int getattr(XPRMcontext ctx, XPRMattrdesc attrdesc, void *ref, XPRMalltypes
      *value);
```

Arguments

`ctx`	Mosel's execution context
`attrdesc`	An attribute descriptor
`ref`	An entity of the type associated to the attribute
`value`	Pointer to an area where the value of the attribute is returned

Return value

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

Further information

This function makes it possible to retrieve an attribute of an entity using an attribute descriptor as returned by `findattrdesc`.

Related topics

`findattrdesc`.

2.5 Record access

`getfieldval`	Get the value of a field of a record.	p. 77
`getnextfield`	Get the next field of a record type.	p. 76
`setfieldval`	Set the value of a field of a record.	p. 78

getnextfield

Purpose

Get the next field of a record type.

Synopsis

```
void *getnextfield(XPRMcontext ctx, void *ref, int typcode, const char
      **name, int *type, int *number);
```

Arguments

`ctx`	Mosel's execution context
`ref`	Reference pointer or `NULL`
`typcode`	Type code of the record
`name`	Field name
`type`	Field type code
`number`	Field number (in the record)

Return value

Reference pointer for the next call to `getnextfield`.

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 `getfieldval` (`setfieldval`) to retrieve (set) the value of the corresponding field in an object of this record type.

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

Related topics

`getfieldval`, `setfieldval`.

getfieldval

Purpose

Get the value of a field of a record.

Synopsis

```
void getfieldval(XPRMcontext ctx, int typcode, void *ref, int number,
      XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`ref`	Reference to the record
`typcode`	Type code of the record
`number`	Field number (in the record)
`value`	Pointer to an area where the field value is returned

Further information

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

Related topics

`getnextfield`, `setfieldval`.

setfieldval

Purpose

Set the value of a field of a record.

Synopsis

```
int setfieldval(XPRMcontext ctx, int typcode, void *ref, int number,
      XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`ref`	Reference to the record
`typcode`	Type code of the record
`number`	Field number (in the record)
`value`	Pointer to an area where the field value is stored

Return value

0 if executed succesfully, a positive value otherwise (wrong type).

Further information

This function sets the value of a field of a record for types integer, real, string and boolean. The field number must be obtained from the function `getnextfield`. Its value is valid as long as the model is loaded in memory.

Related topics

`getnextfield`, `setfieldval`, `copyval`.

2.6 Union access

getifunvalue

Purpose

Get the content of a union and perform type expansion as necessary.

Synopsis

```
int getifunvalue(XPRMcontext ctx, int type, XPRMalltypes *value, int
      expand);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type of the referenced value
`value`	Pointer to an area where the value is stored (to be updated)
`expand`	If non-zero user types will be expanded

Return value

An encoded type, `0` if the reference is an undefined union or `-1` in case of error (typically a `NULL` reference).

Further information

1. If the referenced value is a union the function `getunvalue` is called and the value is updated with the content of this union, the returned type corresponds to this new value.
2. If the option `expand` is stated the type will be expanded using `gettypeprop`: for instance if the encoded type refers to a set of integers the function will return `XPRM_STR_SET|XPRM_TYP_INT` instead of the code for this type (the expansion is executed only on user types defining arrays, sets or lists).
3. This function will return its `type` argument and keep `value` unchanged if it cannot perform any operation.

getnextuncomptype

Purpose

Get the next compatible type of a union type.

Synopsis

```
void *getnextuncomptype(XPRMcontext ctx, void *ref, int typcode, int
      *type);
```

Arguments

`ctx`	Mosel's execution context
`ref`	Reference pointer or `NULL`
`typcode`	Type code of the union
`type`	Returned type ID

Return value

Reference pointer for the next call to `getnextuncomptype`.

Further information

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

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

Related topics

`getuntype`, `getuntypeid`, `isuncompat`.

getuntype

Purpose

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

Synopsis

```
int getuntype(XPRMunion un);
```

Argument

`un` Reference to a union entity

Return value

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

Further information

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

Related topics

`getunvalue`, `getuntypeid`.

getuntypeid

Purpose

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

Synopsis

```
int getuntypeid(XPRMunion un);
```

Argument

`un` Reference to a union entity

Return value

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

Further information

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

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

Related topics

`getunvalue`, `getuntype`, `getuntypeself`.

getuntypeself

Purpose

Get the type ID of a union.

Synopsis

```
int getuntypeself(XPRMunion un);
```

Argument

`un` Reference to a union entity

Return value

A type ID of the union.

Further information

This function retrieves the type ID of a union from its reference.

Related topics

`getuntypeid`, `getuntype`.

getunvalue

Purpose

Get the value stored in a union.

Synopsis

```
int getunvalue(XPRMcontext ctx, XPRMunion un, XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`un`	Reference to a union entity
`value`	Pointer to an area where the value is returned

Return value

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

Further information

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

Related topics

`getifunvalue`, `getuntypeid`, `getuntype`, `setunvalue`.

isuncompat

Purpose

Test whether a type ID is compatible with a union type.

Synopsis

```
int isuncompat(XPRMcontext ctx, int untype, int typeid);
```

Arguments

`ctx`	Mosel's execution context
`untype`	ID of a union type
`typeid`	ID of the type to test

Return value

`1` if the provided type is compatible with the union, `0` otherwise.

Related topics

`getnextuncomptype`, `setunvalue`.

resetunion

Purpose

Reset a union entity.

Synopsis

```
int resetunion(XPRMcontext ctx, XPRMunion un);
```

Arguments

`ctx`	Mosel's execution context
`un`	Reference to a union entity

Return value

`0` if the operation succeeded or `1` in case of error.

Further information

This routine restores a union to its initial state: the stored value is released and the entity is marked as *undefined*. The operation will fail if it is applied to a constant reference.

Related topics

`setunvalue`.

setunvalue

Purpose

Set the value of a union.

Synopsis

```
int setunvalue(XPRMcontext ctx, XPRMunion un, int type, XPRMalltypes
      *value);
```

Arguments

`ctx`	Mosel's execution context
`un`	Reference to a union entity
`type`	ID of the type of the value to assign
`value`	Pointer to an area where the value is stored

Return value

`0` if successful, any other value indicates an error.

Further information

1. The value associated with the union is replaced by a copy of the provided value that must have a type compatible with the union (otherwise the operation fails). If the current value of the union is of the same type as the provided value then a direct assignment between the entity stored in the union and the new value is performed. Otherwise, the current value of the union is released, a new instance of the required type is created before performing a direct assignment. Note that although no implicit type conversion is performed, assigning an integer to a union compatible only with reals will succeed (*i.e.* the integer value will be automatically converted to real).

2. If the provided value is also a union the assignment operation is executed on the value stored in this union, if its type is not compatible with the destination of the assignment the operation will fail. If this value is an undefined union then the destination entity will also become undefined (*i.e.* the current value of the union is released and the entity returns to its initial state).

3. If the given type is 0 and the value is a `NULL` pointer or a reference to `NULL` this routine will have the same effect as calling `resetunion`.

Related topics

`unionwrap`, `getunvalue`.

unionwrap

Purpose

Set the value of a union as a reference to an entity.

Synopsis

```
XPRMunion unionwrap(XPRMcontext ctx, XPRMunion un, int type, XPRMalltypes
      *value);
```

Arguments

`ctx`	Mosel's execution context
`un`	Reference to a union entity
`type`	ID of the type of the value to wrap
`value`	Pointer to an area where the value is stored

Return value

Parameter `un` or `NULL` in case of error.

Further information

1. This routine has a similar effect as `setunvalue` except that when the provided value is not a constant of a basic type, it stores in the union a reference to this value instead of duplicating it.
2. A union can store a reference to a basic type (*e.g.* `XPRM_STR_REF | XPRM_TYP_INT`) but since basic types have no reference counting such a reference must be valid for the duration of the execution of the model.
3. If the provided value is also a union the wrapping operation is executed on the value stored in this union, the operation will fail if its type is not compatible with the destinaton union. The operation will also fail if this value is an undefined union.

Related topics

`setunvalue`, `getunvalue`.

2.7 Problem and solution access

exportprob

Purpose

Export the active problem to a file.

Synopsis

```
int exportprob(XPRMcontext ctx, const char *options, const char *fname,
      XPRMlinctr obj);
```

Arguments

`ctx`	Mosel's execution context
`options`	format of the output. Possible value are: `""` LP output format, minimization (default) `"m"` MPS output format `"p"` Maximization (default is minimization) `"s"` Use scrambled names `"x"` Hexadecimal numbers for MPS
`fname`	File name, may be `NULL`
`obj`	Objective to use for optimization, may be `NULL`

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

getact

Purpose

Get the activity value of a linear constraint.

Synopsis

```
double getact(XPRMcontext ctx, XPRMlinctr ctr);
```

Arguments

`ctx` Mosel's execution context

`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

`getcsol`, `getslack`.

getcsol

Purpose

Get the solution value of a linear constraint.

Synopsis

```
double getcsol(XPRMcontext ctx, XPRMlinctr ctr);
```

Arguments

`ctx` Mosel's execution context

`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

`getdual`, `getslack`.

getctrnextterm

Purpose

Enumerate the list of terms contained in a linear constraint.

Synopsis

```
void *getctrnextterm(XPRMcontext ctx, XPRMlinctr ctr, void *prev,
       XPRMmpvar *var, double *coeff);
```

Arguments

`ctx` Mosel's execution context

`ctr` Reference to a linear constraint

`prev` Last value returned by this function. Should be `NULL` for the first call

`var` Pointer to return the decision variable reference for the current term

`coeff` Pointer to return the coefficient of the current term

Return value

The value to be used as `prev` for the next call or `NULL` when all terms have been returned.

Example

The following function displays the terms of a linear constraint.

```
void displinctr(XPRMcontext ctx, XPRMlinctr ctr)
{
 void *prev;
 XPRMmpvar v;
 double coeff;

 prev=mm->getctrnextterm(ctx, ctr, NULL, &v, &coeff);
 mm->printf(ctx, "%g ",coeff);
 while(prev!=NULL) {
  prev=mm->getctrnextterm(ctx, ctr, prev, &v, &coeff);
  mm->printf(ctx, "%+g %p ", coeff, v);
 }
 mm->printf(ctx, "\n");
}
```

Further information

1. This function can be called repeatedly to enumerate all terms of a linear constraint. For the first call, the parameter `prev` must be `NULL` and the function returns the constant term of the linear constraint (the value returned for `var` is then `NULL` and `coeff` contains the constant term). For the following calls, the value of `prev` must be the last value returned by the function. The enumeration is completed when the function returns `NULL`.
2. With a range constraint it is possible to retrieve the lower bound of the range using the special value `-1` for the `prev` parameter. In this case the function returns always `NULL`.

getctrnum

Purpose

Get the row number of a linear constraint.

Synopsis

```
int getctrnum(XPRMcontext ctx, XPRMlinctr ctr);
```

Arguments

`ctx`	Mosel's execution context
`ctr`	Reference to a linear constraint

Return value

`>= 0`	Row number of the linear constraint
`-1`	Row number not available
`< -1`	SOS number (SOS are numbered -2,-3,...)

Further information

This function returns the row number of a linear constraint. A value of -1 is returned if no problem is available or if the constraint does not belong to the current problem. A negative value < -1 refers to a SOS (SOS numbers are -2,-3,... in this context).

Related topics

`getvarnum`.

getctrtyp

Purpose

Get the type of a linear constraint

Synopsis

```
int getctrtyp(XPRMlinctr ctr);
```

Argument

`ctr` Reference to a linear constraint

Return value

Bit encoded constraint type.

Further information

This function returns the type and status of the given constraint. The type may be extracted using the macro `XPRM_GETCTYPE(c)`. The possible types are:

`XPRM_CTYPE_UNCONS` unbounded constraint (a linear expression)

`XPRM_CTYPE_GEQ` for $\geqslant$ constraint

`XPRM_CTYPE_LEQ` for $\leqslant$ constraint

`XPRM_CTYPE_EQ` for $=$ constraint

`XPRM_CTYPE_SOS1` for `is_sos1`

`XPRM_CTYPE_SOS2` for `is_sos2`

`XPRM_CTYPE_CONT` for `is_continuous`

`XPRM_CTYPE_INT` for `is_integer`

`XPRM_CTYPE_BIN` for `is_binary`

`XPRM_CTYPE_PINT` for `is_partint`

`XPRM_CTYPE_SEC` for `is_semcont`

`XPRM_CTYPE_SINT` for `is_semint`

`XPRM_CTYPE_FREE` for `is_free`

The status of the constraint is checked using the macro `XPRM_CHKCSTAT(c,s)` where `s` is one of:

`XPRM_CSTAT_EMPTY` the constraint is empty (it may contain a constant term)

`XPRM_CSTAT_HIDN` the constraint is hidden

`XPRM_CSTAT_TEMP` the constraint is temporary (it will be dropped after the termination of the calling function)

getdual

Purpose

Get the dual value of a linear constraint.

Synopsis

```
double getdual(XPRMcontext ctx, XPRMlinctr ctr);
```

Arguments

`ctx` Mosel's execution context

`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

`getact`, `getcsol`, `getslack`.

getobjval

Purpose

Get the objective function value.

Synopsis

```
double getobjval(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

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

`getprobstat`.

getprobstat

Purpose

Get the problem status of a model.

Synopsis

```
int getprobstat(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

Bit encoded problem status or 0.

Further information

This function returns the status of the current 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

`getobjval`.

getrcost

Purpose

Get the reduced cost value of a variable.

Synopsis

```
double getrcost(XPRMcontext ctx, XPRMmpvar var);
```

Arguments

`ctx` Mosel's execution context

`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

`getvsol`.

getslack

Purpose

Get the slack value of a linear constraint.

Synopsis

```
double getslack(XPRMcontext ctx, XPRMlinctr ctr);
```

Arguments

`ctx` Mosel's execution context

`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

`getcsol`, `getslack`.

getvarnum

Purpose

Get the column number of a decision variable.

Synopsis

```
int getvarnum(XPRMcontext ctx, XPRMmpvar var);
```

Arguments

`ctx` Mosel's execution context

`var` Reference to a variable

Return value

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

Further information

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

Related topics

`getctrnum`.

getvsol

Purpose

Get the solution value of a variable.

Synopsis

```
double getvsol(XPRMcontext ctx, XPRMmpvar var);
```

Arguments

`ctx` Mosel's execution context

`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, MIP: last integer solution or 0).

Related topics

`getrcost`.

2.8 Matrix related functions

The functions presented in this section are used to obtain the matrix representation of the current problem (LP/MIP) from Mosel and to inform Mosel about the result of a solution process that has been used on the provided matrix. The data structure used to represent a matrix (`XPRMmatrix`) is not described in this document, the reader is referred to the header file `mosel_sl.h` for further details. The structure `XPRMmipsolver` is documented with the function `loadmat`.

chgmatsolv

Purpose

Change the MIP solver interface currently in use.

Synopsis

```
int chgmatsolv(XPRMcontext ctx, XPRMmipsolver *mipslv, void *mipctx);
```

Arguments

`ctx`	Mosel's execution context
`mipslv`	A MIP solver interface (may be `NULL` to preserve the current value)
`mipctx`	The associated context (may be `NULL` to preserve the current value)

Return value

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

Further information

This function replaces the currently used MIP solver interface or its context (this interface is provided to Mosel by the function `loadmat`). No operation is performed with the solver to be replaced (in particular, the matrix is not unloaded).

Related topics

`loadmat`, `getmatsolv`.

genmpnames

Purpose

Generate row, SOS and column names.

Synopsis

```
void genmpnames(XPRMcontext ctx, XPRMlinctr obj, XPRMmpvar extra[], int
      strip);
```

Arguments

`ctx` Mosel's execution context

`obj` Objective. Possible values are

`NULL`	No objective defined
`XPRM_KEEPOBJ`	Last objective used
`linctr`	Reference to a linear constraint to be used as the objective

`extra` Array of decision variables to be added to the matrix even if they appear in no constraint. The last reference of the array must be `NULL`. This parameter may be `NULL`

`strip` Use scrambled names if not 0

Example

see `getnextcol` and `getmpname`

Further information

This function generates the row, SOS and column names that can be accessed with function `getmpname`. Note that these names can be generated only if a matrix exists (this function may therefore generate a matrix if none is available).

Related topics

`getnextcol`, `getmpname`.

getinfcause

Purpose

Get the reference of the variable causing the failure in 'loadmat'.

Synopsis

```
XPRMmpvar getinfcause(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

Reference to a decision variable or `NULL` (if no information is available).

Further information

This function may be called after the `loadmat` function if this one returns a positive value (the matrix could not be loaded because of inconsistent bounds). The reference returned is the variable for which bounds are inconsistent.

Related topics

`loadmat`, `getvcinfcause`

getmatsize

Purpose

Get matrix size information.

Synopsis

```
void getmatsize(XPRMcontext ctx, int *ncol, int *nrow, int *nsos, int
      *ngents);
```

Arguments

`ctx`	Mosel's execution context
`ncol`	A pointer to return the number of columns (may be `NULL`)
`nrow`	A pointer to return the number of rows (may be `NULL`)
`nsos`	A pointer to return the number of SOS (may be `NULL`)
`ngents`	A pointer to return the number of non-continuous variables (may be `NULL`)

Example

see `getmpname`

Further information

This function returns the number of columns, rows, sos and global entities defined by the current problem. This information is available only once a matrix has been generated.

Related topics

`loadmat`

getmatsolv

Purpose

Get the MIP solver interface currently in use.

Synopsis

```
int getmatsolv(XPRMcontext ctx, XPRMmipsolver **mipslv, void **mipctx);
```

Arguments

`ctx`	Mosel's execution context
`mipslv`	A pointer to return the MIP solver interface (may be `NULL`)
`mipctx`	A pointer to return the context associated (may be `NULL`)

Return value

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

Further information

This function returns the MIP solver interface currently in use (this interface is provided to Mosel by the function `loadmat`).

Related topics

`loadmat`, `getmatsolv`.

getmpname

Purpose

Get a row, SOS or column name.

Synopsis

```
const char *getmpname(XPRMcontext ctx, int type, int ind);
```

Arguments

`ctx`	Mosel's execution context	
`type`	`XPRM_MPNAM_COL`	Get the name of a column,
	`XPRM_MPNAM_ROW`	Get the name of a row,
	`XPRM_MPNAM_SOS`	Get the name of a SOS
`ind`	Index of the row/SOS/column (value returned by `getvarnum` and `getctrnum`)	

Return value

The entity name or `NULL` if names are not available or the entity does not belong to the matrix.

Example

```
/* Display all names */
void disprsnames(XPRMcontext ctx)
{
 int ndx,ncol,nsos,nrow;

 if(mmx->getmatsolv(ctx, NULL, NULL)<3)
 {                                    /* only if a problem exists */
  mmx->genmpnames(ctx, XPRM_KEEPOBJ, NULL, 0);
  mmx->getmatsize(ctx, NULL, &nrow, &nsos, NULL);
  for(ndx=0;ndx<ncol;ndx++)
   mm->printf(ctx, "%s\n", mmx->getmpname(ctx, XPRM_MPNAM_COL, ndx));
  for(ndx=0;ndx<nrow;ndx++)
   mm->printf(ctx, "%s\n", mmx->getmpname(ctx, XPRM_MPNAM_ROW, ndx));
  for(ndx=0;ndx<nsos;ndx++)
   mm->printf(ctx, "%s\n", mmx->getmpname(ctx, XPRM_MPNAM_SOS, ndx));
 }
}
```

Further information

This function is used to access the row, column and SOS names generated by the function `genmpnames`. Index numbers start with 0 but the numbering returned by `getctrnum` for SOS (-2, -3...) is also accepted.

Related topics

`genmpnames`, `getctrnum`, `getvarnum`.

getnextcol

Purpose

Get the next column in the matrix.

Synopsis

```
void *getnextcol(XPRMcontext ctx, void *adr, XPRMmpvar *var, int *col);
```

Arguments

`ctx`	Mosel's execution context
`adr`	Value returned by the previous call to this function or `NULL` for the first call
`var`	A pointer to return the variable reference
`col`	A pointer to return the column number

Return value

The value to be used as the `adr` parameter for a subsequent call or `NULL` when all columns have been enumerated.

Example

```
/* Display column names */
void dispcolnames(XPRMcontext ctx)
{
 void *ptr;
 XPRMmpvar v;
 int ndx;

 if(mmx->getmatsolv(ctx, NULL, NULL)<3)
 {                                      /* only if a problem exists */
  mmx->genmpnames(ctx, XPRM_KEEPOBJ, NULL, 0);
  ptr=NULL;
  while((ptr=mmx->getnextcol(ctx, ptr, &v, &ndx))!=NULL)
  mm->printf(ctx, "%p=%s\n", v,
             mmx->getmpname(ctx, XPRM_MPNAM_COL, ndx));
 }
}
```

Further information

This function returns the next column information (decision variable reference and column number) in the matrix. The second parameter is used to store the current location in the table: the value returned by the function has to be used as input for the subsequent call until the function returns `NULL`.

Related topics

`getnextrow`, `getnextsos`.

getnextrow

Purpose

Get the next row in the matrix.

Synopsis

```
void *getnextrow(XPRMcontext ctx, void *adr, XPRMlinctr *ctr, int *row);
```

Arguments

`ctx` Mosel's execution context

`adr` Value returned by the previous call to this function or `NULL` for the first call

`ctr` A pointer to return the constraint reference

`row` A pointer to return the row number

Return value

The value to be used as the `adr` parameter for a subsequent call or `NULL` when all rows have been enumerated.

Example

see `getnextcol`

Further information

This function returns the next row information (constraint reference and row number) in the matrix. The second parameter is used to store the current location in the table: the value returned by the function has to be used as input for the subsequent call until the function returns `NULL`.

Related topics

`getnextcol`, `getnextsos`.

getnextsos

Purpose

Get the next SOS in the matrix.

Synopsis

```
void *getnextsos(XPRMcontext ctx, void *adr, XPRMlinctr *ctr, int *sos);
```

Arguments

`ctx` Mosel's execution context

`adr` Value returned by the previous call to this function or `NULL` for the first call

`ctr` A pointer to return the constraint reference

`sos` A pointer to return the SOS number

Return value

The value to be used as the `adr` parameter for a subsequent call or `NULL` when all SOS have been enumerated.

Further information

This function returns the next SOS information (constraint reference and SOS number) in the matrix. The second parameter is used to store the current location in the table: the value returned by the function has to be used as input for the subsequent call until the function returns `NULL`.

Related topics

`getnextcol`, `getnextrow`.

getprobnextctr

Purpose

Get the next constraint in the problem.

Synopsis

```
XPRMlinctr getprobnextctr(XPRMcontext ctx, void **prev);
```

Arguments

`ctx` Mosel's execution context

`prev` Pointer to a pointer used by the function to store its current status. This variable has to be initialised with `NULL` for the first call

Return value

The next constraint in the problem or `NULL` when the last constraint has been returned.

Further information

This function returns the next constraint in the problem. The second parameter is used to store the current location in the table: it must be a pointer to a pointer. For the first call, its value has to be initialised with `NULL` and the function returns the first constraint.

Related topics

`getprobobj`

getprobobj

Purpose

Get the current objective function.

Synopsis

```
XPRMlinctr getprobobj(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

The linear constraint used as objective function or `NULL` if the objective function is not available.

Further information

This function returns the current objective function. Note that the objective function is not available if it was defined as a local linear constraint.

Related topics

`getprobnextctr`

getvarorder

Purpose

Get the internal order number of a decision variable.

Synopsis

```
int getvarorder(XPRMcontext ctx,XPRMmpvar var);
```

Arguments

`ctx` Mosel's execution context

`var` A reference to a decision variable or `NULL`

Return value

Internal order number or the maximum number of variables if `var` is `NULL`.

Further information

1. At its creation a variable is assigned an order number. This order number never changes and it is used as the sorting criterion when generating the matrix.

2. If the argument `var` is `NULL`, the returned value corresponds to the number of variables created so far.

Related topics

`reordercols`, `isvarbefore`

getvcinfcause

Purpose

Get the reference of the variable or constraint causing the failure in 'loadmat'.

Synopsis

```
void *getvcinfcause(XPRMcontext ctx, int *what);
```

Arguments

`ctx` Mosel's execution context

`what` A reference to report the type of the returned pointer (`1` for a decision variable and `2` for a constraint

Return value

Reference to a decision variable or constraint or `NULL`.

Further information

This function may be called after the `loadmat` function if this one returns a positive value (the matrix could not be loaded because of inconsistent bounds). The reference returned is the variable (type `XPRMmpvar`) or constraint (type `XPRMlinctr`) for which bounds are inconsistent.

Related topics

`loadmat`

loadmat

Purpose

Produce a matrix representation of the current problem.

Synopsis

```
int loadmat(XPRMcontext ctx, XPRMlinctr obj, XPRMmpvar extra[], int frel,
      XPRMmipsolver *mipslv, void *mipctx);
```

Arguments

`ctx` Mosel's execution context

`obj` Objective. Possible values are

`NULL`	No objective defined
`XPRM_KEEPOBJ`	Keep last objective that was used
`linctr`	Reference to a linear constraint to be used as the objective

`extra` Array of decision variables to be added to the matrix even if they appear in no constraint. The last reference of the array must be `NULL`. This parameter may be `NULL`.

`frel` Bit encoded options:

`MM_MAT_FORCE`	load the matrix in all cases, otherwise do it only if something has changed since the last call
`MM_MAT_HUGE`	use 64 bit indexing for matrix and SOS coefficients

`mipslv` MIP solver interface

`mipctx` A pointer used as a context when calling functions of the MIP solver interface

Return value

0 if successful, `XPRM_PBINF` if the problem was found infeasible (due to inconsistent bounds) and -1 if an error has occurred.

Example

```
int my_loadmat(XPRMcontext ctx, void *mipctx, XPRMmatrix *m);
void my_delmat(XPRMcontext ctx, void *mipctx);
double my_getsol_c(XPRMcontext ctx, void *mipctx, int what, int col);
double my_getsol_r(XPRMcontext ctx, void *mipctx, int what, int row);

XPRMmipsolver my_MIPi= {           /* define the MIP solver interface: */
 {'N','G','L','E','R','1','2'},    /* constraint types */
 {' ','I','B','P','S','R'},        /* variable types */
 my_loadmat,                       /* to load the matrix */
 my_delmat,                        /* to delete a previously loaded matrix */
 my_getsol_c,                      /* to get solution for columns */
 my_getsol_r};                     /* to get solution for rows */
...
/* Mosel Procedure for loading the problem: */
/* loadprob(obj) */
int c_loadprob(XPRMcontext ctx, void *libctx)
{
 XPRMlinctr obj;

 obj=XPRM_POP_REF(ctx);
 if(mmx->loadmat(ctx, obj, NULL, 0, &my_MIPi, libctx)<0)
  return XPRM_RT_ERROR;
 else
  return XPRM_RT_OK;
}
...
/* C function for loading the matrix into 'lpsolver' */
int my_loadmat(XPRMcontext ctx, void *mipctx, XPRMmatrix *m)
```

```
{
 XPRMlinctr obj;

 if(m->mstart!=NULL)+                /* full (re)load */
 lpsolver_loadmat(m->ncol, m->nrow, m->qrtype, m->rhs, m->range, m->obj,
                  m->mstart, m->mrwind, m->dmatval, m->dlb, m->dub);
 else                                /* only the objective has to be updated */
  lpsolver_updateobj(m->obj)
 return 0;
}
```

Further information

This function can be used when a module requires a matrix representation of the current problem. The communication between Mosel and the module is defined by the *MIP solver interface* structure `XPRMmipsolver`:

`char ctypecode[7]` characters to use to specify constraint types (*i.e.* 'l' for less or equal)

`char vtypecode[6]` characters to use to specify variable types (*i.e.* 'b' for binary)

`int (*loadmatrix)(XPRMcontext,void *,XPRMmatrix*)` send the matrix to the module

`void (*delmatrix)(XPRMcontext,void *)` delete a matrix previously loaded with `loadmat`

`double (*getsol_var)(XPRMcontext,void *,int,int)` get solution values for columns

`double (*getsol_ctr)(XPRMcontext,void *,int,int)` get solution values for rows (this entry may be `NULL`)

Each time Mosel calls one of these *communication functions*, it uses as the second parameter, the value of `mipctx` provided when `loadmat` is called.

When this function is called, if this is required (problem modified since last call) or forced (because of `frel`), the matrix is generated using the information provided by `ctypecode` and `vtypecode`. `vtypecode[0]` is not used except if it is `'+'`: in this case the default lower bound for variables is 0 instead of -infinity. The matrix is then passed to the module by using the function `loadmatrix` (the return value of which must be 0 if successful). Note that if only the objective has been modified since the last call, the entry `mstart` of the structure `XPRMmatrix` is `NULL` (only the objective is provided). After this call, the data structure `XPRMmatrix` is freed (so, if the module needs it for later access, it has to make a copy of it).

The function `delmatrix` is used when a matrix previously loaded via `loadmatrix` becomes useless (for instance before loading an updated version of the matrix).

If the module makes available solution values (which is done by calling the function `setprobstat`), the function `getsol_var` is called whenever Mosel requires the solution value (with second parameter 0) or the reduced cost (with second parameter 1) of the given column (third parameter: the first column has number 0).

Similarly, `getsol_ctr` is used to get the slack value (second parameter is 0) or the dual value (second parameter is 1) of the given row (third parameter: the first row has number 0).

If `MM_MAT_HUGE` is requested, fields `nzr` and `nzrsos` are set to `-1` (if non null), fields `nzr_l` and `nzrsos_l` must be used instead. Similarly, `mstart_l` and `msstart_l` are populated in place of `mstart` and `msstart`.

Related topics

`setprobstat`, `reordercols`, `getvcinfcause`.

isvarbefore

Purpose

Compare order number of two decision variables.

Synopsis

```
int isvarbefore(XPRMcontext ctx,XPRMmpvar var1,XPRMmpvar var2);
```

Arguments

`ctx`	Mosel's execution context
`var1`	A reference to a decision variable
`var2`	A reference to a decision variable

Return value

Result of comparison: `getvarorder(var1)<getvarorder(var2)`.

Related topics

`getvarorder`

reordercols

Purpose

Change the order of the columns in a matrix.

Synopsis

```
int reordercols(XPRMcontext ctx, XPRMmatrix *mat, int order[]);
```

Arguments

`ctx` Mosel's execution context

`mat` Matrix generated by Mosel

`order` An array of integers of size `mat->ncol`

Return value

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

Further information

This function may be called from the `loadmatrix` function (see `loadmat`) to reassign the column numbers to the variables. The array `order` contains a permutation of the column numbers (0,..., `mat->ncol`-1) that defines the new order of the matrix columns.

Related topics

`loadmat`, `getvarorder`.

resetsolv

Purpose

Release the memory used for solving a problem.

Synopsis

```
void resetsolv(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Further information

This function releases all resources allocated for solving the problem (*e.g.* column/row mapping to Mosel variables/constraints; names, solution values...). If a MIP solver is defined, the 'delmatrix' function it defines is called.

Related topics

`loadmat`

setprobstat

Purpose

Set the problem status.

Synopsis

```
void setprobstat(XPRMcontext ctx, int status, double objval);
```

Arguments

`ctx` Mosel's execution context

`status` Bit encoded problem status:

`XPRM_PBSOL`	A solution is available,
`XPRM_PBOPT`	Optimal solution found,
`XPRM_PBUNF`	Optimization interrupted,
`XPRM_PBINF`	Problem is infeasible,
`XPRM_PBUNB`	Problem is unbounded,
`XPRM_PBOTH`	Optimization failed

`objval`

Further information

This function sets the problem status in Mosel. It is usually invoked after an optimization procedure in order to tell Mosel the result of the operation and whether a solution is available. Note that the status `XPRM_PBSOL` can be combined with the other status values (*e.g.* `XPRM_PBOPT | XPRM_PBSOL`).

Related topics

`getprobstat`

2.9 Dictionary access

buildnames

Purpose

Generate entities names for a given set of types.

Synopsis

```
int buildnames(XPRMcontext ctx, int nbt, int *types,
      int (**flts)(XPRMcontext ctx, void *ctf, int type, void *ref),
      int (**snms)(XPRMcontext ctx, void *ctf, int type, void *ref, const
      char *name), void **ctfs);
```

Arguments

`ctx`	Mosel's execution context
`nbt`	Number of types to look for
`types`	Types to consider (must be referenced types like `XPRM_TYP_LINCTR` or any module type)
`flts`	Filter functions to select for which entities a name has to be generated (one for each type)
`snms`	Functions to record the generated names (one for each type)
`ctfs`	Function contexts (one for each type)

Return value

`0` if successful, `1` in case of error

Further information

1. This routine enumerates the symbols of the model dictionary to locate the entities of the selected types (both direct references and arrays). For each of the found entities, it generates a name based on the dictionary identifier that it passes back to the caller through one of the `snms` functions. This name might come from a temporary buffer and the function must save a copy of it if it needs to be accessed after the callback.

2. Symbols are processed in two steps: the names provided by the use of `setname` are first enumerated and then the symbols from the model dictionary. As a consequence a given entity might be reported two times if it has been assigned a name via `setname` and it is also a public symbol.

3. The parameters `types`, `flts`, `snms` and `ctfs` are arrays of size `nbt`: they characterise what types to look for and how to process them. The function `flts(t)` is called each time an entity of type `types(t)` has been found. It is provided with some specific context, `ctfs(t)`, the type number of the entity, `types(t)`, and its reference. `buildnames` will generate a name for this entity only if this function returns a non-zero value. This identifier will then be passed to `snms(t)`. The process is aborted if this function returns a non-zero value.

Related topics

`setentname`.

findident

Purpose

Find an identifier in the dictionary.

Synopsis

```
int findident(XPRMcontext ctx, const char *name, XPRMalltypes *value,int
      flags);
```

Arguments

`ctx`	Mosel's execution context
`name`	Identifier
`value`	Pointer to an area where the dictionary entry is returned
`flags`	Bit encoded options. Possible values:
	`XPRM_FID_NOLOC` Look only for global symbols (local symbols, if any, are ignored)
	`XPRM_FID_BTREF` For a reference to a basic type entity return the pointer instead of the value

Return value

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

Example

See `getnextproc`

Further information

1. This function returns the dictionary entry of a given identifier for a given model together with its type. By default the dictionary contains only global public symbols, private symbols are included when the model is compiled with option `"-g"`. The routine will also return local symbols (according to the current context) if the model was compiled with option `"-G"` and the option flag does not include `XPRM_FID_NOLOC`.

2. The returned type information is bit encoded and associates a *type code* and a *structure* that can be extracted using the macros `XPRM_TYP(t)` and `XPRM_STR(t)`.
   The possible structures are:
   `XPRM_STR_CONST` the object is a constant scalar
   `XPRM_STR_REF` the object is a reference
   `XPRM_STR_LIST` the object is a list
   `XPRM_STR_SET` the object is a set
   `XPRM_STR_ARR` the object is an array
   `XPRM_STR_PROC` the object is a procedure or function
   `XPRM_STR_MEM` the object is a memory block
   `XPRM_STR_UTYP` the object is a user defined type

   Depending on the structure, the possible type codes are:
   `XPRM_TYP_NOT` no type (procedure)
   `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, list, set, array, function)
   `XPRM_TYP_BOOL` Boolean (constant, reference, list, set, array, function)
   `XPRM_TYP_MPVAR` decision variable (reference, list, set, array)
   `XPRM_TYP_LINCTR` linear constraint (reference, list, set, array)

   Any other value designates an external type (type provided by a module or defined in the model). Moreover, if the structure is `XPRM_STR_UTYP`, the identifier is the name of a user type and the value (an integer) corresponds to the expanded form of this type (see `gettypeprop`). Otherwise, the function `gettypeprop` 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 `findident` is decoded as follows depending on the structure:
   `value.integer` for constant, reference (without option `XPRM_FID_BTREF`) or user type
   `value.real` for constant or reference (without option `XPRM_FID_BTREF`)
   `value.string` for constant or reference (without option `XPRM_FID_BTREF`)
   `value.boolean` for constant or reference (without option `XPRM_FID_BTREF`)
   `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) or a reference to a basic type (if option `XPRM_FID_BTREF` is used)
   `value.proc` for procedure and function
   `value.memblk` for memory block

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

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

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

3. When the model is compiled with debug information (flags `"-g"` or `"-G"`) both public and private

Related topics

`getnextident`, `findtypecode`, `gettypeprop`.

findtypecode

Purpose

Find the code associated to a type.

Synopsis

```
int findtypecode(XPRMcontext ctx, const char *name);
```

Arguments

`ctx` Mosel's execution context

`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

`gettypeprop`.

getannotations

Purpose

Retrieve annotations of a model.

Synopsis

```
int getannotations(XPRMcontext ctx, const char *ident, const char *prefix,
      const char **ann, int maxann);
```

Arguments

`ctx`	Mosel's execution context
`ident`	Symbol to be considered or `NULL` for global declarations
`prefix`	Filtering prefix
`ann`	Array of size `maxann` where to store the annotations (can be `NULL`)
`maxann`	Size of `ann` (to get up to `maxann/2` annotations)

Return value

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

Further information

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

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

Related topics

`getnextanident`.

getentname

Purpose

Retrieve the name associated to an entity.

Synopsis

```
const char *getentname(XPRMcontext ctx, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`ref`	Entity reference

Return value

Name associated to the entity or `NULL`.

Related topics

`setentname`.

getnextanident

Purpose

Get the next annotated identifier in the dictionary.

Synopsis

```
const char *getnextanident(XPRMcontext ctx, void **ref);
```

Arguments

`ctx` Mosel's execution context

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

Return value

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

Further information

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

Related topics

`getnextident`, `getannotations`.

getnextident

Purpose

Get the next identifier in the dictionary.

Synopsis

```
const char *getnextident(XPRMcontext ctx, void **ref);
```

Arguments

`ctx` Mosel's execution context

`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 `findident`.

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

`findident`.

getnextparam

Purpose

Get the next parameter of the model.

Synopsis

```
const char *getnextparam(XPRMcontext ctx, void **ref);
```

Arguments

`ctx` Mosel's execution context

`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 running 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.

getnextproc

Purpose

Get the next overloaded version of a procedure or function.

Synopsis

```
XPRMproc getnextproc(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.

Example

The following code extract shows how to find the function `mosfct(i:integer,r:real):boolean`.

```
int find_mosfct(XPRMcontext ctx)
{
 XPRMalltypes fct;
 const char *partyp;
 int nbpar,type;

 if(XPRM_STR(mm->findident(ctx, "mosfct", &fct))==XPRM_STR_PROC)
  do {
   mm->getprocinfo(fct.proc, &partyp, &nbpar, &type);
   if((XPRM_TYP(type)==XPRM_TYP_BOOL) && (nbpar==2) && !strcmp(partyp,"ir"))
    return 1;
   fct.proc=mm->getnextproc(fct.proc);
  } while(fct.proc!=NULL);
 return 0;
}
```

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. Note that this function does not give access to any subroutines provided by modules.

Related topics

`getprocinfo`.

getprocinfo

Purpose

Get the procedure/function information.

Synopsis

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

Arguments

`proc`	Reference to a procedure or function
`partyp`	Returned string of parameter types
`nbpar`	Returned number of parameters
`type`	Bit encoded returned type of the subroutine

Return value

0 if successful, 1 otherwise.

Example

see `getnextproc`

Further information

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

`i`	an integer
`r`	a real
`s`	a text string
`b`	a Boolean
`v`	a decision variable (type `mpvar`)
`c`	a linear constraint (type `linctr`)
`I`	a range set
`a`	an array (of any kind)
`e`	a set (of any type)
`l`	a list (of any type)
`|xxx|`	external type named '*xxx*'. A type code may also be given as '*%???*' where '*???*' (3 hexadecimal digits) is the code number
`!xxx!`	the set named '*xxx*'
`Andx.t`	an array indexed by '*ndx*' of the type '*t*'. '*ndx*' is a string describing the type of each indexing set.
`Et`	a set of type '*t*'
`Lt`	a list of type '*t*'
`?`	any type
`*`	function with variable number of parameters (this character is the last one of the string)

For instance, the procedure:

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

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

Related topics

`getnextproc`.

gettypeprop

Purpose

Get a property of a type.

Synopsis

```
int gettypeprop(XPRMcontext ctx, int type,
      int prop, XPRMalltypes *value);
```

Arguments

`ctx` Mosel's execution context

`type` Code of a type

`prop` Property to retrieve. Possible values:

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

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

Return value

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

Further information

1. This function returns a property of an external type (types provided by modules or user defined). For the properties `XPRM_TPROP_NAME` and `XPRM_TPROP_SIGN`, the information is returned in `value->string`, for `XPRM_TPROP_ITIPS` the information is recorded in `value->ref` and for the 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
   `XPRM_MTP_CMP` Test of equality is supported by this type
   `XPRM_MTP_SHARE` An entity of this type can be declared as shared
   `XPRM_MTP_TFBIN` Type supports export/import in binary format
   `XPRM_MTP_ORD` Type supports comparison
   `XPRM_MTP_CONST` Type supports constant definition
   `XPRM_MTP_ANDX` Type is an array indexer
   `XPRM_MTP_NAMED` Type supports name association

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

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

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

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

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

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

9. The property `XPRM_TPROP_ITYPS` is an array of integers describing the types of the fields of a constant record (it is `NULL` if the referenced type is not a record). The first cell of this array is the number of fields and each following entry is the type of the corresponding field (this information is required when implementing an IO driver supporting initialisations blocks).

10. The information returned for the property `XPRM_TPROP_NBELT` depends on the kind of type considered: for an array this is the number of dimensions (`getarrdim`), for a subroutine this corresponds to the number of parameters it requires (`getprocinfo`), for a union this is the number of compatible types of this union (`getnextuncomptype`), for a record this is the number of fields it contains (`getnextfield`) and for a problem this gives the number of components (`getnextpbcomp`).

Related topics

`getnextfield`, `getnextuncomptype`, `getnextpbcomp`, `findtypecode`.

csrtoref

Purpose

Get the constant object associated to a CSREF structure.

Synopsis

```
void *csrtoref(XPRMcontext ctx, void *csr);
```

Arguments

`ctx` Mosel's execution context

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

Return value

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

Further information

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

Related topics

`gettypeprop`, `newcsr`.

getnextpbcomp

Purpose

Get the next component of a problem type.

Synopsis

```
void *getnextpbcomp(XPRMcontext ctx, void *ref, int typcode,
       int *type);
```

Arguments

`ctx`	Mosel's execution context
`ref`	Reference pointer or `NULL`
`typcode`	Type code
`type`	Returned type of the component

Return value

Reference pointer for the next call to `getnextpbcomp`.

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 `findident`. 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`).

setentname

Purpose

Associate a name with an entity.

Synopsis

```
int setentname(XPRMcontext ctx, int type, void *ref, const char *name);
```

Arguments

`ctx`	Mosel's execution context
`type`	Entity type or `XPRM_STR_ARR` for an array
`ref`	Entity reference
`name`	New name for the entity or `NULL` (it must be a registered string, see `regstring`)

Return value

`0` on success, `-1` if the pointer is `NULL` and `-2` if the type does not support this functionality.

Further information

1. This function makes it possible to associate a name with an entity, this identifier is used by `buildnames` to generate its output (*e.g.* matrix names are produced this way). Only decision variables (`XPRM_TYP_MPVAR`), linear constraints (`XPRM_TYP_LINCTR`) and native types with the property `XPRM_DTYP_NAMED` (like `nlctr`) can be used with this function.

2. Both scalars and arrays can be named. In the case of an array the name generation is performed by `buildnames` such that array cells of a dynamic array created after this call will also be named appropriately.

3. The name association will be removed if the provided name is `NULL` (no operation is performed if this association does not exist).

Related topics

`getentname`, `buildnames`.

2.10 Model execution and handling of modules

callproc

Purpose

Call a procedure/function of the running model.

Synopsis

```
int callproc(XPRMcontext ctx, XPRMproc proc, XPRMalltypes *parst);
```

Arguments

`ctx` Mosel's execution context

`proc` Reference to a routine (as returned by `findident` or `getnextproc`)

`parst` An array containing the parameters for the routine

Return value

`XPRM_RT_OK` if execution succeeded, an error code otherwise (as returned by `XPRMrunmod`).

Example

The following code extract shows how to call the function `mosfct(i:integer,r:real):boolean`.

```
void callfct(XPRMcontext ctx)
{
 XPRMalltypes fct,parst[2];

 mm->findident(ctx, "mosfct", &fct);
 parst[1].integer=10;
 parst[0].real=5.5;
 mm->printf(ctx, "mosfct(10,5.5)=");
 mm->callproc(ctx, fct.proc, parst);
 mm->printf(ctx, "%d", parst[0].boolean);
}
```

Further information

1. Execute the procedure or function `proc` that is defined in a currently running model. The routine reference is obtained from `findident` or `getnextproc`. In the array `parst`, the parameters for the function must be stored in reverse order. If the routine is a function, the return value is stored in the first cell of `parst`.
2. If the procedure terminates by calling `exit(code)`, the return value is `XPRM_RT_EXIT` and the exit code is stored in the first cell of `parst`.

Related topics

`findident`, `getnextproc`, `getprocinfo`.

finddso

Purpose

Find a DSO descriptor from a module name.

Synopsis

```
XPRMdsolib finddso(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

`opendso`.

opendso

Purpose

Open a module and retrieve its IMCI interface.

Synopsis

```
XPRMdsolib opendso(XPRMcontext ctx, const char *name, void **imci);
```

Arguments

`ctx`	Mosel's execution context
`name`	Name of the module to open
`imci`	Reference to a pointer where to store the IMCI entry of the module (may be `NULL`)

Return value

A reference to a DSO descriptor or `NULL` if the module cannot be loaded.

Further information

1. This function *opens* a module by loading it (if it is not already in core memory) and incrementing its reference count. The module is not reset if the running model is not using it.

2. Each module opened with this function must be closed using `closedso`.

Related topics

`closedso`, `finddso`.

closedso

Purpose

Close a module previously opened with `opendso`.

Synopsis

```
void closedso(XPRMcontext ctx, const char *name);
```

Arguments

`ctx` Mosel's execution context

`name` Name of the module to close

Further information

This function *closes* a module by updating its reference count.

Related topics

`opendso`.

getdsoctx

Purpose

Get the running context and IMCI interface of a module.

Synopsis

```
void **getdsoctx(XPRMcontext ctx, XPRMdsolib dso, void **imci);
```

Arguments

`ctx`	Mosel's execution context
`dso`	Reference to a dynamic shared object loaded by Mosel
`imci`	Reference to a pointer where to store the IMCI entry of the module (may be `NULL`)

Return value

The reference to the location where the context of the module is (will be) stored or `NULL` if the running model does not use this module.

Further information

1. This function may be used for inter-module communication. It returns the location where the module context is stored. The context may not be available when this function is called if the module has not yet been initialized, this is likely to happen when `getdsoctx` is called from within a reset function and no priority has been defined for the module (see Section 1.5.2). In this case the return value is a valid pointer to `NULL` that will be updated after the requested module is initialized.

2. If the last parameter is not `NULL`, it is used to return the value of the service `XPRM_SRV_IMCI` of the given module. Note that the function will populate this parameter even if it returns `NULL`.

getdsoprop

Purpose

Get a property of a dynamic shared object.

Synopsis

```
int getdsoprop(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
`XPRM_PROP_NBREF`	Number of loaded models that use the module
`XPRM_PROP_PATH`	Path to the actual module file
`XPRM_PROP_COMPAT`	Smallest compatible version (0 if not available)

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

Further information

This function returns information about a given module. The type of the property (specified via the `prop` argument) decides how the argument `value` is interpreted: 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`.

getdsoparam

Purpose

Get the current value of a control parameter.

Synopsis

```
int getdsoparam(XPRMcontext ctx, XPRMdsolib dso, const char *name,
      int *type, XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`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. This function requires that the model uses the requested module.
2. If the argument `dso` is `NULL`, the function looks for the value of Mosel parameter (like `"realfmt"`).
3. The type can be decoded using the macro `XPRM_TYP`. Moreover, the bits `XPRM_CPAR_READ` and `XPRM_CPAR_WRITE` are set to indicate if the parameter can be read or written respectively (using `getparam` and `setparam`).

Related topics

`getparam`, `setdsoparam`.

getmodprop

Purpose

Get a property of running model.

Synopsis

```
int getmodprop(XPRMcontext ctx, int prop, XPRMalltypes *value);
```

Arguments

`ctx` Mosel's execution context

`prop` Property to retrieve. Possible values:

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

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

Return value

0 if successful, 1 otherwise.

Further information

1. This function returns information about the running 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`, `VERSION`, `SECSTAT`, `NBBIM` and `NBTYPES`; `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`.

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

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

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

getnextmoddso

Purpose

Get the next module loaded for the model.

Synopsis

```
void *getnextmoddso(XPRMcontext ctx, void *ref, XPRMdsolib *dso);
```

Arguments

`ctx` Mosel's execution context

`ref` Reference pointer or `NULL`

`dso` Returned reference to a dynamic shared object

Return value

Reference pointer for the next call to `XPRMgetnextmoddso`.

Further information

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

getparam

Purpose

Get a Mosel control parameter value.

Synopsis

```
int getparam(XPRMcontext ctx, int parnum, XPRMalltypes *value);
```

Arguments

`ctx` Mosel's execution context

`parnum`

0	For `"zerotol"` (real),
1	For `"realfmt"` (string),
2	For `"ioctrl"` (Boolean),
3	For `"iostatus"` (integer),
4	For `"nbread"` (integer),
5	For `"readcnt"` (Boolean)
6	For `"UTC"` (Boolean)
7	For `"autofinal"` (Boolean)
8	For `"tmpdir"` (string)
9	For `"workdir"` (string)
10	For `"restrict"` (integer)
11	For `"modelname"` (string)
12	For `"modelnumber"` (integer)

`value` Parameter value, the type depends on the parameter.

Return value

0 if successful, 1 otherwise.

Further information

Get the value of a control parameter of Mosel. To get the value of a module control parameter, the function `getdsoparam` must be used. The reader is referred to the Mosel Reference Manual for a description of the Mosel control parameters.

Related topics

`getdsoparam`.

setdsoparam

Purpose

Set the value of a control parameter.

Synopsis

```
int setdsoparam(XPRMcontext ctx, XPRMdsolib dso, const char *name,
      int type, XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`dso`	Reference to a dynamic shared object loaded by Mosel or `NULL`
`name`	Name of the control parameter (lower case only)
`type`	Type of the new value
`value`	New value for the control parameter

Return value

0 if successful, 1 otherwise.

Further information

1. This function sets the value of a control parameter of the given module. This function requires that the model uses the requested module.
2. An automatic conversion is performed if the provided value is not of the expected type (the function will fail if this conversion is not possible).
3. If the argument `dso` is `NULL`, the function looks for the value of Mosel parameter (like `"realfmt"`).

Related topics

`getparam`, `getdsoparam`.

stoprun

Purpose

Stop the current execution.

Synopsis

```
void stoprun(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Further information

When this function has been called, the current execution stops as soon as possible (*i.e.* after the termination of the current function). Note that the execution is also aborted if a native function returns `XPRM_RT_STOP`.

chkinterrupt

Purpose

Check whether the model is signaled for termination.

Synopsis

```
int chkinterrupt(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

0 if execution can continue, `XPRM_RT_STOP` if model is expected to terminate.

Further information

A native function which execution time is longer than 1 or 2 seconds should call this function regularly and abort its processing as soon as possible when the return value is not 0.

2.11 Input and output

dispmsg

Purpose

Display an error message.

Synopsis

```
void dispmsg(XPRMcontext ctx, const char *format, ...);
```

Arguments

`ctx`	Mosel's execution context or `NULL`
`format`	*printf* style format

Further information

1. This function sends an error message to the error output stream of the model corresponding to the given context. If the context provided is `NULL`, the messages goes to the global error output stream of Mosel. The parameters expected by this fonction correspond to those of the standard C function `printf`. The special format `%r` can also be used to display real values: it implies the use of the format specified by control parameter `realfmt` (see `getparam`).

2. An *error stream* is always available even before and after execution of the model.

Related topics

`printf`.

getstreambuf

Purpose

Get the data currently held in an IO buffer.

Synopsis

```
int getstreambuf(XPRMcontext ctx, int mode,int *size,char **buffer);
```

Arguments

`ctx`	Mosel's execution context	
`mode`	`XPRM_F_READ`	Buffer of the input stream
	`XPRM_F_WRITE`	Buffer of the output stream
	`XPRM_F_ERROR`	Buffer of the error stream
`size`	Pointer to an area where the buffer size is stored	
`buffer`	Pointer to an area where the buffer reference is stored (may be `NULL`)	

Return value

0 if successful.

Further information

For all I/O operations Mosel uses buffers of 2048 bytes. This function makes it possible to access the data currently held in the buffers associated to the current streams: for an input stream this corresponds to the data that has already been loaded from the driver but not yet consumed by the model and for an output stream this is the data that has been output but not yet sent to the driver.

fclose

Purpose

Close the file corresponding to the current input, output or error stream.

Synopsis

```
int fclose(XPRMcontext ctx, int flag);
```

Arguments

`ctx`	Mosel's execution context	
`flag`	`XPRM_F_READ`	Close input stream
	`XPRM_F_WRITE`	Close output stream
	`XPRM_F_ERROR`	Close error stream

Return value

0 if successful.

Further information

This function closes the file associated to the current input, output or error stream. Output streams are automatically flushed before being closed. After the stream has been released, the stream that was active before is selected. Closing the default input, output or error stream (*i.e.* the streams available at start up) has no effect.

Related topics

`fopen`.

fcopy

Purpose

Copy a file.

Synopsis

```
int fcopy(XPRMcontext ctx, const char *src, const char *dst);
```

Arguments

`ctx`	Mosel's execution context
`src`	Source file name
`dst`	Destination file name

Return value

0 if successful, 1 if `src` cannot be open, 2 if `dst` cannot be open and 3 if an error occured during the copy.

Further information

This function makes a copy of file `src`. Source and destination file names do not have to use the same IO driver. The function will perform no operation and return `0` if `src` and `dst` are identical text strings.

feof

Purpose

Check if the current input stream is at the end of the file.

Synopsis

```
int feof(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

-1 in case of error, 1 if the end of file has been reached, 0 otherwise.

Further information

This function returns the end-of-file status of the current input stream.

fflush

Purpose

Flush the current output stream.

Synopsis

```
int fflush(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

0 if successful, 1 otherwise.

Further information

This function flushes the current output stream: all pending messages (still stored in buffers) are processed.

fgetid

Purpose

Get the stream number of the current input, output or stream.

Synopsis

```
int fgetid(XPRMcontext ctx, int flag);
```

Arguments

`ctx`	Mosel's execution context	
`flag`	`XPRM_F_READ`	Get input stream number
	`XPRM_F_WRITE`	Get output stream number
	`XPRM_F_ERROR`	Get error stream number

Return value

Stream number or `-2` if the corresponding stream is not available.

Further information

This function returns the stream number of the current input, output or error stream.

Related topics

`fselect`.

fgets, fgetsl

Purpose

Read a text string from the current input stream.

Synopsis

```
char *fgets(XPRMcontext ctx, char *s, int size);
int fgetsl(XPRMcontext ctx, char *s, int size);
```

Arguments

`ctx`	Mosel's execution context
`s`	Buffer where to store the string
`size`	Size of the buffer (number of bytes)

Return value

`s` if successful or `NULL` if the end of file has been reached. `fgetsl` returns the number of bytes read.

Further information

These functions read a text string from the default input stream.

fmove

Purpose

Move (rename) a file.

Synopsis

```
int fmove(XPRMcontext ctx, const char *src, const char *dst);
```

Arguments

`ctx`	Mosel's execution context
`src`	Source file name
`dst`	Destination file name

Return value

0 if successful, 1 if `src` cannot be open, 2 if `dst` cannot be open, 3 if an error occured during the copy and 4 if `src` cannot be removed after copy.

Further information

This function renames file `src` to file `dst` . Source and destination file names do not need to use the same IO driver. If both are using the same IO driver and this driver supports the function, the operation is performed by the driver otherwise the original file is first duplicated then deleted. Deletion can be performed only if the driver of the source file supports file removal.

fopen

Purpose

Open a file and select it as the current input/output/error stream.

Synopsis

```
int fopen(XPRMcontext ctx, int flag, const char *fname);
```

Arguments

`ctx` Mosel's execution context

`flag` Open mode (may be combined). Possible values are

`XPRM_F_READ`	Open for reading
`XPRM_F_WRITE`	Open for writing (reset the file)
`XPRM_F_ERROR`	Open for writing an error stream (reset the file)
`XPRM_F_APPEND`	Open for writing (append)
`XPRM_F_BINARY`	Open in binary mode (default is text mode)
`XPRM_F_LINBUF`	If open for writing, flushes buffer after end of each line (default when writing to a console or for an error stream)
`XPRM_F_SILENT`	Do not display IO error messages
`XPRM_F_DELCLOSE`	Delete the file after the stream has been closed

`fname` Name of the file to open

Return value

The stream number or a negative value in case of error.

Further information

1. This function opens a file and assigns the resulting stream to the current input stream (file open for reading), to the current output stream (file open for writing) or error stream (flag `XPRM_F_ERROR` used). The value returned can be used as input for function `fselect`.

2. The *binary mode* is only effective under the Windows operating system. Its main effect is to disable the conversion of the `\n` character into the sequence `\r\n` when outputting text strings.

Related topics

`fclose`, `normfname`.

fread

Purpose

Read a block of data from the current input stream.

Synopsis

```
long fread(XPRMcontext ctx, void *buf, long size);
```

Arguments

`ctx`	Mosel's execution context
`buf`	Buffer where to store read data
`size`	Size of the buffer (number of bytes)

Return value

The number of bytes actually read, 0 when end of stream has been reached and -1 in case of error.

Further information

This function reads a block of data from the default input stream. A returned size smaller than the maximum authorised does not necessarily implies an end of stream.

fremove

Purpose

Remove (delete) a file.

Synopsis

```
int fremove(XPRMcontext ctx, const char *todel);
```

Arguments

`ctx` Mosel's execution context

`todel` File to be removed

Return value

0 if successful, 1 if `todel` cannot be open and 4 if the operation is not possible.

Further information

This function deletes file `todel`. A return value of 4 may indicate that the driver does not support file removal.

fselect

Purpose

Select a stream to be the current input, output or error stream.

Synopsis

```
int fselect(XPRMcontext ctx, int stream);
```

Arguments

`ctx`	Mosel's execution context
`stream`	Stream number

Return value

The stream number or -1 in case of error.

Further information

This function selects a stream as the current input, output or error stream depending on the status of the stream (*i.e.* a stream open for reading is assigned to the current input stream).

Related topics

`fgetid`.

fsize

Purpose

Get the size of a file.

Synopsis

```
size_t fsize(XPRMcontext ctx, const char *f);
```

Arguments

`ctx`	Mosel's execution context
`f`	File to be considered

Return value

Size of the file in bytes or `-1` in case of error.

Further information

The error value `-1` will be returned either if the file cannot be found or if the corresponding IO driver does not support the functionality.

fskip

Purpose

Skip a block of data from the current input stream.

Synopsis

```
int fskip(XPRMcontext ctx, int size);
```

Arguments

`ctx` Mosel's execution context

`size` Number of bytes to skip

Return value

A negative value indicates an error.

Further information

If the IO driver handling the input stream does not support this operation, the specified amount of data will be read and discarded.

fwrite

Purpose

Write a block of data to the current output stream.

Synopsis

```
long fwrite(XPRMcontext ctx, void *buf, long size);
```

Arguments

`ctx`	Mosel's execution context
`buf`	Buffer where data to be written is stored
`size`	Size of the buffer (number of bytes)

Return value

0 if successful, -1 in case of error.

Further information

This function writes a block of data to the default output stream. The successful completion of this function does not guarantee that all data is actually saved since IO operations are buffered. A call to `fflush` can be used to make sure all data is transmitted.

Related topics

`fflush`.

fgetinfo

Purpose

Retrieve information about current input, output or error stream.

Synopsis

```
int fgetinfo(XPRMcontext ctx, int *mode, int *line, int *col,
      const char **iodrv, const char **filename);
```

Arguments

`ctx`	Mosel's execution context
`mode`	Pointer to store mode
`line`	Pointer to store current line
`col`	Pointer to store current col
`iodrv`	Pointer to store IO driver name for this stream
`filename`	Pointer to store file name for this stream

Return value

Stream number or `-2` if the corresponding stream is not available.

Further information

This function returns information of a stream through its parameters. Parameters may be `NULL`when the corresponding information is not required. The second parameter is used both for input and output: if it is `NULL` or points to a 0 value, the function returns information for the input stream. If its value is `XPRM_F_WRITE`, the function returns information for output stream and if its value is `XPRM_F_ERROR` the information is related to the error stream. This parameter is updated by the function to reflect the actual value of mode for the corresponding stream (see `fopen`). Note that bit `XPRM_F_IOERR` is set when an error has been encountered during an IO operation on the corresponding stream. Line and column information are valid only when the stream is read using `fgets`.

Related topics

`fgets`, `fopen`.

printf

Purpose

Send a message to the current output stream.

Synopsis

```
int printf(XPRMcontext ctx, const char *format, ...);
```

Arguments

`ctx`	Mosel's execution context
`format`	*printf* style format

Return value

The number of characters printed or -1 in the case of an error.

Further information

This function sends a message to the current output stream of the running model. The parameters expected by this function correspond to those of the standard C function `printf`. The special format `%r` can also be used to display real values: it implies the use of the format specified by control parameter `realfmt` (see `getparam`).

Related topics

`dispmsg`.

setioerrmsg

Purpose

Set an error message for an IO driver operation.

Synopsis

```
void setioerrmsg(XPRMcontext ctx, const char *msg, int ecode);
```

Arguments

`ctx`	Mosel's execution context
`msg`	Error message
`ecode`	Error code

Further information

This function may be used from functions implementing an IO driver (namely operations `open`, `close`, `read`, `write` and `skip`) in order to set a descriptive message before returning an error status (see Section 1.6).

2.12 Miscellaneous

newmuid

Purpose

Generate a unique identifier.

Synopsis

```
const char *newmuid(XPRMcontext ctx)
```

Argument

`ctx` Mosel's execution context

Return value

An identifier string.

Further information

1. This function returns a string of the form `muid#_xxx` where `#` is an execution number in hexadecimal (specific to this model execution) and `xxx` a random hexadecimal number. It is guaranteed that each generated value will never be returned again.

2. The returned value is a registered string (see `regstring`).

newcsr

Purpose

Create a constant reference.

Synopsis

```
void *newcsr(XPRMcontext ctx, int type, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type code of the constant reference
`ref`	A reference of an entity corresponding to the specified type

Return value

A reference to a new CSREF entity.

Further information

Mosel saves references to constant objects of native or record types in entities of structure `XPRM_STR_CSREF`. This function creates a new object of this type from an exisiting object. For instance if the expanded code of the provided type `type` is `(XPRM_STR_CSREF|tt)` then creating an instance of type `type`, that corresponds to a constant of type `tt`, will be achieved by calling this function with a reference to an object of type `tt` (*i.e.* the value of the constant to create).

Related topics

`newref`, `csrtoref`.

newref

Purpose

Increase the reference count of a referenced object.

Synopsis

```
void *newref(XPRMcontext ctx,int type, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type or structure of the object
`ref`	Reference of the object or `NULL`

Return value

The reference to an object (*i.e.* the third argument or a new instance) or `NULL` in case of error.

Further information

1. If a native function needs to use the reference to an object after its termination, it must tell the system that the corresponding object must be preserved even if it is no longer used in the model: this is the role of this function. When the reference becomes useless for the native code, it must be released by a call to `delref`. The parameter `type` can be `XPRM_TYP_MPVAR`, `XPRM_TYP_LINCTR`, `XPRM_STR_ARR`, `XPRM_STR_LIST`, `XPRM_STR_SET`, `XPRM_STR_PROC` or the type code of an external type.

2. If the third argument is `NULL`, this function returns the reference to a newly created object. However this function cannot be used to create a constant reference, use `newcsr` for this purpose.

Related topics

`delref`.

delref

Purpose

Decrease the reference count of a referenced object.

Synopsis

```
void delref(XPRMcontext ctx,int type, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type or structure of the object
`ref`	Reference of the object

Further information

This function must be used after a reference saved with `newref` becomes useless for the native code in order to let the system release the object in case it is no longer referenced. The parameter `type` can be `XPRM_TYP_MPVAR`, `XPRM_TYP_LINCTR`, `XPRM_STR_ARR`, `XPRM_STR_LIST`, `XPRM_STR_SET`, `XPRM_STR_PROC` or the type code of an external type.

Related topics

`newref`.

dbggetlocation

Purpose

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

Synopsis

```
int dbggetlocation(XPRMcontext ctx, int lndx, int *line,
       const char **fname);
```

Arguments

`ctx`	Mosel's execution context
`lndx`	Line index, -1 for current location or -2 for the last location of the model
`line`	Pointer to an area where the line number is returned
`fname`	Pointer to an area where the file name is returned

Return value

0 if successful, 1 otherwise (invalid parameters)

Further information

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

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

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

getrand

Purpose

Generate a random number.

Synopsis

```
double getrand(XPRMcontext ctx);
```

Argument

`ctx` Mosel's execution context

Return value

A randomly generated number between 0 and 1.

Further information

This function returns a random number in the range `[0,1)` using the generator associated to the running model.

gettypeid

Purpose

Get the type ID of a set, list, array or union.

Synopsis

```
int gettypeid(XPRMcontext ctx, int st, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`st`	Structure of the reference (*e.g.* `XPRM_STRUCT_ARRAY`)
`ref`	A reference to the entity

Return value

A type ID, `0` if the entity is not a reference to a user type and `-1` in case of error (NULL reference of invalid structure).

Example

Considering the following definitions and assuming that `ref1` refers to `S1` and `ref2` refers to `S2`, the call `gettypeid(ctx, XPRM_STR_SET, ref1)` will report a positive value (the type ID of `myset`) while `gettypeid(ctx, XPRM_STR_SET, ref2)` will return `0` (because this set is not an instance of any user type).

```
declarations
   myset=set of integer
   S1:myset
   S2:set of integer
end-declarations
```

Further information

This function can be applied to arrays (`XPRM_STRUCT_ARRAY`), sets (`XPRM_STRUCT_SET`), lists (`XPRM_STRUCT_LIST`) and unions (`XPRM_STRUCT_UNION`). It will return a non-zero value only if the corresponding reference is an instance of a user-defined type.

getversions

Purpose

Get version numbers.

Synopsis

```
int getversions(int whichone);
```

Argument

`whichone` Which version number to return:

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

Return value

The version number requested or 0 in case of error.

Further information

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

hashmix

Purpose

Compute a hash value of a data buffer.

Synopsis

```
unsigned int hashmix(XPRMcontext ctx,unsigned int hv,const void *buf,size_t
      len);
```

Arguments

`ctx`	Mosel's execution context
`hv`	An initial hash value
`buf`	Data buffer to process
`len`	Size of the data buffer

Return value

Hash value for the given data buffer.

Further information

This function computes a hash value for a given data buffer, it can be used for implementing the `XPRM_CPY_HASH` option of the module type function *copy* (see Section 1.2.3). The provided initial value `hv` can be used to combine a list of buffers by using the returned value of a previous call as input for the following.

hmdel

Purpose

Release a hashmap.

Synopsis

```
int hmdel(XPRMcontext ctx,XPRMhashmap hm);
```

Arguments

`ctx`	Mosel's execution context
`hm`	The hashmap to release

Return value

`0` if sucessful or `-1` in case of error.

Further information

This function releases the resources used by a *hashmap* previously created using `hmnew`. Hashmaps that are not explicitly freed using this routine are automatically released when the model is reset (after its execution).

Related topics

`hmset`, `hmget`, `hmnew`, `hmdump`.

hmdump

Purpose

Retrieve the content of a hashmap.

Synopsis

```
unsigned int hmdump(XPRMcontext ctx,XPRMhashmap hm, size_t *keys, size_t
      *vals, unsigned int nbmax);
```

Arguments

`ctx`	Mosel's execution context
`hm`	A hashmap
`keys`	Array to retrieve the keys (may be `NULL`)
`vals`	Array to retrieve the values (may be `NULL`)
`nbmax`	Size of the arrays `keys` and `vals`

Return value

The number of entries recorded in the map (may be greater than `nbmax`).

Related topics

`hmset`, `hmget`, `hmdel`, `hmnew`, `hmenum`.

hmenum

Purpose

Iterate over all elements of a hashmap.

Synopsis

```
int hmenum(XPRMhashmap hm, void *ctf, int (*getpair)(void *ctf, unsigned
      int nb, size_t key, size_t val));
```

Arguments

`hm`	A hashmap
`ctf`	Function context for `getpair`
`getpair`	Function to call for each pair <key,value>(may be `NULL`)

Return value

The number of elements in the hashmap or `-1` if the enumeration has been interrupted.

Further information

1. This function enumerates all elements of a hashmap: each pair <key,value>is passed back to the caller through the `getpair` function that receives its context (`ctf`) and the number of elements that have been processed so far in addition to the key and its associated value.

2. The value returned by the `getpair` function decides how to proceed: if it returns `0` the enumeration continues; with the value `-1` the current element will be removed from the hashmap before continuing; any other value will cause the enumeration to terminate (and `hmenum` will return `-1`).

Related topics

`hmdump`.

hmfind

Purpose

Get a reference to the value associated with a key in a hashmap.

Synopsis

```
size_t *hmfind(XPRMcontext ctx,XPRMhashmap hm, size_t key, int how);
```

Arguments

`ctx`	Mosel's execution context
`hm`	A hashmap
`key`	Key to be found
`how`	Handling method:
	`XPRM_HM_SET` Create the entry if it does not exist yet
	`XPRM_HM_UPD` Return `NULL` if the key could not be found

Return value

A reference to the value associated with the key or `NULL` if the key could not be found.

Related topics

`hmget`, `hmset`, `hmnew`, `hmdel`, `hmdump`.

hmget

Purpose

Get the value associated with a key in a hashmap.

Synopsis

```
size_t hmget(XPRMcontext ctx,XPRMhashmap hm, size_t key);
```

Arguments

`ctx`	Mosel's execution context
`hm`	A hashmap
`key`	Key to be found

Return value

The value associated with the key or `0` if the key could not be found.

Related topics

`hmfind`, `hmset`, `hmnew`, `hmdel`, `hmdump`.

hmnew

Purpose

Create a new hashmap.

Synopsis

```
XPRMhashmap hmnew(XPRMcontext ctx,unsigned int minsize,int flags);
```

Arguments

`ctx`	Mosel's execution context
`minsize`	Initial size of the hash table (`0` for default size)
`flags`	Reserved for future use (should be `0`)

Return value

A reference to the newly created hashmap or `NULL` in case of error.

Further information

1. This function creates a *hashmap* from the current context of execution. This datastructure makes it possible to record <key,value>pairs and retrieve efficiently the value associated to a given key.

2. Hashmaps and their associated datastructures are allocated using the memory management routines of the running model, as a consequence, hashmaps are automatically released when the model is reset. During the execution of the model it is however possible to release a map using function `hmdel`

Related topics

`hmset`, `hmget`, `hmdel`, `hmdump`.

hmset

Purpose

Set the value associated with a key in a hashmap.

Synopsis

```
size_t hmset(XPRMcontext ctx,XPRMhashmap hm, size_t key, size_t val, int
      how);
```

Arguments

`ctx` Mosel's execution context

`hm` A hashmap

`key` Key to be found

`val` Value for the key

`how` Handling method:

`XPRM_HM_SET`	Create a new entry or update the value
`XPRM_HM_CLS`	Do not set the value: drop the entry if it already exists
`XPRM_HM_ONCE`	Do not modify an existing entry
`XPRM_HM_UPD`	Only update an existing entry

Return value

The value associated with the key.

Related topics

`hmget`, `hmnew`, `hmdel`, `hmdump`.

isdefined

Purpose

Check whether the provided entity is defined.

Synopsis

```
int isdefined(XPRMcontext ctx, int type, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type or structure of the provided reference
`ref`	A reference to an entity of the specified type

Return value

`-1` if the type code is not valid, `1` if the entity is *defined*, or `0` otherwise.

Further information

1. The parameter `type` can be either a type number or a structure code (*e.g.* `XPRM_STR_UNION`).
2. In the case of a reference to a subroutine or a union this function will succeed if the reference is not `NULL` and the container has received a value. For other types it will return `0` if `ref` is `NULL` and `1` otherwise.

memalloc

Purpose

Allocate memory from the Mosel memory manager.

Synopsis

```
void *memalloc(XPRMcontext ctx, size_t size, int flags);
```

Arguments

`ctx` Mosel's execution context

`size` Amount of memory to allocate in bytes

`flags` Possible values (they can be combined):

- `XPRM_MEM_ZERO` The allocated memory is initialised with zeros
- `XPRM_MEM_NULL` Instead of exiting the process, the value `NULL` is returned if memory cannot be allocated

Return value

A pointer to the allocated memory or `NULL` in case of error

Further information

This function allocates a block of memory from the Mosel memory manager. This memory block might be released using `memfree` but all memory allocated by this function is automatically returned to the system when the model is reset.

Related topics

`memfree`, `stackalloc`.

memfree

Purpose

Free a block of memory of the Mosel memory manager.

Synopsis

```
void memfree(XPRMcontext ctx, void *p, size_t size);
```

Arguments

`ctx`	Mosel's execution context
`p`	Address of the memory block to release
`size`	Size of the memory block to release

Further information

1. This function releases a block of memory previously allocated by a call to `memalloc`. The parameter `size` must have the same value as the one used to perform the allocation (behaviour is undefined and may lead to memory corruption if the values differ).

2. The memory block released by this function is not necessarily freed: it might be recycled for being reused by the library but all memory allocated by `memalloc` is automatically returned to the system when the model is reset.

Related topics

`memalloc`.

normfname

Purpose

Normalize a file name.

Synopsis

```
char *normfname(char *fname, const char *ext, int force);
```

Arguments

`fname`	Path and file name to normalize
`ext`	Extension to append to the file name
`force`	If 0, the extension is appended to the file name only if it has no extension; otherwise, the provided extension replaces the existing file name extension

Return value

The parameter `fname`.

Further information

This function prepares a file name (including its path) for being used with operating system functions by setting the correct path separator (*e.g.* replace '/' by '\' under Windows) and appends a given file extension to it. The array `fname` must be large enough to receive the provided extension `ext`.

Related topics

`fopen`.

pathcheck

Purpose

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

Synopsis

```
int pathcheck(XPRMcontext ctx,const char *path,char *fullpath,
      int maxlen,int acc);
```

Arguments

`ctx`	Mosel's execution context
`path`	Path (or file name) to be processed
`fullpath`	Buffer to return the expanded path
`maxlen`	Size of `fullpath`
`acc`	Operation to perform. Possible values: `XPRM_RCHK_READ` Check whether path or file can be read `XPRM_RCHK_WRITE` Check whether path or file can be written `XPRM_RCHK_NOCHK` Only expand the path without testing access `XPRM_RCHK_IODRV` Check whether the path includes an IO driver and expand `"tmp:"`

Return value

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

Further information

1. This routine returns an absolute path to the file name it gets as input and optionally checks whether access is allowed according to the current restrictions.

2. The function will return an error code if option `XPRM_RCHK_IODRV` is used and the provided path includes an IO driver different from `":"` (default driver) and `"tmp:"` (temporary directory). With this option, the prefix `"tmp:"` is replaced by the absolute path to the temporary directory and `":"` is removed before path expansion. Note that in this mode the leading blanks of the input string are trimmed before any processing.

realtostr

Purpose

Generate the textual representation of a real value.

Synopsis

```
int realtostr(XPRMcontext ctx, char *buf, int bufsize, const char *realfmt,
      double v);
```

Arguments

`ctx`	Mosel's execution context
`buf`	Text buffer to store the resulting string
`bufsize`	Size of `buf`
`realfmt`	C format for the conversion (may be `NULL`)
`v`	Value to convert

Return value

Length of the generated string.

Further information

This function performs the same conversion as `printf`, in particular it handles special values *nan* and *inf* and reports `0` if the value to convert is smaller than the current zero tolerance (Mosel parameters `zerotol` and `txtztol`). If the provided `realfmt` is `NULL`, the format defined in the current context is used (Mosel parameter `realfmt`).

Related topics

`getparam`.

regstring, regstringl

Purpose

Register a text string.

Synopsis

```
const char *regstring(XPRMcontext ctx, const char *string);
const char *regstringl(XPRMcontext ctx, const char *string, size_t len);
```

Arguments

`ctx`	Mosel's execution context
`string`	Text string to register. It must be NULL terminated unless its length is provided via the `len` parameter.
`len`	Length of the string in bytes (not including the terminating null character)

Return value

Registered text string.

Further information

Mosel requires that each text string is registered. If a native function returns a newly generated string or uses a new string in a Mosel data structure (like a set), this string must be registered with this function before it is used. The returned value is a copy of the string argument (unless the provided parameter was already a registered string). Registered strings remain valid until the end of execution of the model (*i.e.* they are never released or moved).

Related topics

`setglobal`, `addelset`, `addellist`, `insellist`, `setarrval`, `setfieldval`.

setglobal

Purpose

Set the value of a global identifier.

Synopsis

```
int setglobal(XPRMcontext ctx, const char *text, XPRMalltypes *value);
```

Arguments

`ctx`	Mosel's execution context
`text`	Name of the object to be assigned a value
`value`	Value to be assigned

Return value

`0`	Assignment succeeded
`1`	Name not found
`2`	Operation not permitted

Further information

This function sets the value of a global object (model identifier declared in a `declarations` block outside of any procedure or function).

Related topics

`regstring`.

date2jdn

Purpose

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

Synopsis

```
int date2jdn(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

`jdn2date`,`time`.

jdn2date

Purpose

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

Synopsis

```
void jdn2date(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 `date2jdn` or `time`.

Related topics

`date2jdn`,`time`.

time

Purpose

Get the current date and time.

Synopsis

```
void time(XPRMcontext ctx,int *jdn, int *t,int *tz);
```

Arguments

`ctx`	Mosel's execution context
`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)*864000000000i64+(__int64)t*10000i64)`.

Related topics

`jdn2date`,`date2jdn`.

memoryuse

Purpose

Get an estimate of the memory usage of an entity, a module or the entire model.

Synopsis

```
size_t memoryuse(XPRMcontext ctx,int type, void *ref);
```

Arguments

`ctx`	Mosel's execution context
`type`	Type or structure of the object or `0`
`ref`	Reference of the object or `NULL`

Return value

An estimate of the memory usage in bytes or `-1` if the evaluation cannot be performed.

Further information

1. If the `type` argument is 0 the reference `ref` is interpreted as a constant string. If this string is `NULL` or empty the returned value corresponds to the total amount of memory used by the running model including the loaded modules (if they implement the service `XPRM_SRV_MEMUSE`). If the string is not empty it corresponds to the name of a module: the returned value is the memory consumed by this module that must be currently used by the model.

2. The argument `type` can be either a type code (`XPRM_TYP_*` constants or an extended type code) or a structure code (`XPRM_STR_SET`, `XPRM_STR_LIST` or `XPRM_STR_ARR`). For the types `integer`, `real`, `boolean` and `mpvar` the value returned is the constant amount of memory required by a variable of the corresponding type. For a reference to a `string` or `linctr` the effective memory used by the internal datastructure is returned. In the case of a set or a list of a referenced type only the memory used to represent the collection is accounted, not its content. However the value reported for an array or record includes the memory used by the content of the structure except for strings.

3. For a module type it is required for the module to implement the service `XPRM_SRV_MEMUSE` for the result to be meaningful.

stackalloc

Purpose

Allocate memory on the Mosel execution stack.

Synopsis

```
void *stackalloc(XPRMcontext ctx, size_t size, int flags);
```

Arguments

`ctx` Mosel's execution context

`size` Amount of memory to allocate in bytes

`flags` Possible values (they can be combined):

`XPRM_MEM_ZERO`	The allocated memory is initialised with zeros
`XPRM_MEM_NULL`	Instead of exiting the process, the value `NULL` is returned if memory cannot be allocated

Return value

A pointer to the allocated memory or `NULL` in case of error

Further information

This function allocates a block of memory on the Mosel execution stack. This memory block must be released before the termination of the calling subroutine by a call to `stackfree` otherwise there is no guarantee on when the deallocation will occur (the stack is anyway released at program termination).

Related topics

`stackfree`, `memalloc`.

stackfree

Purpose

Release memory from the Mosel execution stack.

Synopsis

```
void stackfree(XPRMcontext ctx, void *top);
```

Arguments

`ctx` Mosel's execution context

`top` New stack level

Further information

This function restores the level of the Mosel execution stack to the specified address. This reference address is a pointer returned by a preceding call to `stackalloc` and all the memory allocated by any following invocation of the memory allocator are also released (*i.e.* if several memory blocks have been allocated on the stack only the first one must be released with this function independently of the relative order of the corresponding addresses).

Related topics

`stackalloc`.

Appendix

Appendix A

Compiling and storing modules

Unless they are static, modules have to be compiled as dynamic libraries for the host operating system. The following table recalls the minimum set of options to use with the default C compilers for the supported operating systems. We assume here that the file `mymodule.c` contains the source of the module "mymodule" and that the environment variable `MOSEL` points to the installation directory of Mosel.

Linux:

```
gcc -D_REENTRANT -shared -I${MOSEL}/include -o mymodule.dso mymodule.c
```

Solaris:

```
cc -D_REENTRANT -G -I${MOSEL}/include -o mymodule.dso mymodule.c
```

Windows:

```
cl /MD /LD /I%MOSEL%\include /Femymodule.dso mymodule.c
```

The resulting DSO file has to be stored either in the `dso` directory of the Mosel installation or in a location that the environment variable `MOSEL_DSO` points to (this variable is defined in a similar way as the `PATH` environment variable, *i.e.* it is a list of directories).

Note that modules may also be written in C++. In this case, the initialization function has to be declared as a standard C function in order to be located by Mosel (this is not required for static modules).

Example:

```
extern "C" {
 DSO_INIT mymodule_init(XPRMnifct nifct, int *interver, int *libver,
                        XPRMdsointer **interf)
 {
 ...
 }
};
```

Appendix B

Debugging modules

Since modules are loaded dynamically by Mosel, it may be difficult to use a debugger for analysing the behaviour of the program. A work around consists in compiling the module as part of a simple program (static module) that initialises Mosel then executes a model (using the embeded module to debug). A debugger can be used on this program to trace the operation of the module which is not loaded dynamically any more.

Example: the following program can be used to debug the module 'mymodule'.

```
#include <stdlib.h>
#include "xprm_mc.h"

#include "mymodule.c"              /* Include the source of the module */

int main()
{
 int rts;

 XPRMinit();                       /* Declare the module as static */
 XPRMregstatdso("mymodule",mymodule_init);
                                   /* Execute a test model */
 XPRMexecmod("g","mymodule_test",NULL,&rts,NULL);
 return rts;
}
```

The program source `dsodbg.c` provided with the NI examples can be used as a shell for debugging modules.

APPENDIX C

Contacting FICO

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

FICO Customer Support

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

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

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

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

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

Documentation

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

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

FICO Learning

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

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

Sales and maintenance

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

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

About FICO

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

Index