dash optimization

Xpress-Mosel Native Interface
Reference manual

Release 2.0

Last update December 2006

Published by Dash Optimization Ltd

How to Contact Dash

USA, Canada and all Americas

Dash Optimization Inc

Information and Sales: info@dashoptimization.com
Licensing: license-usa@dashoptimization.com
Product Support: support-usa@dashoptimization.com

Tel: +1 (201) 567 9445
Fax: +1 (201) 567 9443

Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Japan

Dash Optimization Japan

Information and Sales: info@jp.dashoptimization.com
Licensing: license@jp.dashoptimization.com
Product Support: support@jp.dashoptimization.com

Tel: +81 43 297 8836
Fax: +81 43 297 8827

WBG Marive-East 21F FASuC B2124
2-6 Nakase Mihama-ku
261-7121 Chiba
Japan

Worldwide

Dash Optimization Ltd

Information and Sales: info@dashoptimization.com
Licensing: license@dashoptimization.com
Product Support: support@dashoptimization.com

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

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

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

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 (*mmetc*, *mmsystem*, *mmodbc*, *mmquad*, and *mmxprs*) 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 1.5.4) 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 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`
- `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*). 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.
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 $\geq$ 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. If the subroutine is a procedure the type must be XPRM_TYP_NOT.

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 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*'
!*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*'
*	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).

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

```
int functionname(XPRMctx *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 1.4.3.

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.

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.

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)(XPRMctx *ctx, void *libctx, void *ref,int typnum)
```

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_PNCTX` 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.

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

```
void (*fdelete)(XPRMctx *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. 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)(XPRMctx *ctx, void *libctx, void *obj, char *dest,
                int maxsize,int typnum)
```

The textual representation of `obj` has to be copied into `dest` the maximum length of which is `maxsize`. The function must return the length of the generated string 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.
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)(XPRMctx *ctx, void *libctx, void *obj,
                  const char *src,int typnum)
```

The object `obj` is initialized with the content of the string `src`. If successful, the function must return 0; any other value is interpreted as a failure.
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 used by Mosel for assignments not explicitly stated (*e.g.* in array initialization or when assigning records).

```
void (*copy)(XPRMctx *ctx, void *libctx, void *dest,void *src,int typnum)
```

The object `dest` receives the content (or becomes a copy) of `src`. 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.

Example:

```
static XPRMdsotyp tabtyp[]=
{
 {"firsttype", 1, 0, createfirst, del1, tostr1, fromstr1,copy1},
 {"secondtype", 2, 0, create2, delete2, NULL, NULL, 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_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 1.4.3) have the same prototype

```
int functionname(XPRMctx *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 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. The following macros are provided for accessing the Mosel stack.

Macros for taking objects from the stack:

```
XPRM_POP_INT(XPRMcontext ctx)
XPRM_POP_REAL(XPRMcontext ctx)
XPRM_POP_STRING(XPRMcontext ctx)
XPRM_POP_REF(XPRMcontext ctx)
```

Macros for putting objects onto the stack:

```
XPRM_PUSH_INT(XPRMcontext ctx, i)
XPRM_PUSH_REAL(XPRMcontext ctx, r)
XPRM_PUSH_STRING(XPRMcontext ctx, s)
XPRM_PUSH_REF(XPRMcontext ctx, r)
```

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(XPRMctx *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 1.5.4) 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 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 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`).

1.4.3.1 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

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

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.

1.4.3.2 Arithmetic operators

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

If @+ and @0 are defined for a given type, the aggregate operator `SUM` can be generated by the Mosel compiler. 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.

1.4.3.3 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 $\rightarrow$ C
@o(A,B):C	logical 'or'	A or B $\rightarrow$ C
@n(A):C	logical negation	not A $\rightarrow$ 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.

1.4.3.4 Comparators

Operator	Operation	Return value	Remarks
@<(A,B):C	strictly less	A<B $\rightarrow$ C	implied by @n(C):C with @g(A,B):C
@>(A,B):C	strictly greater	A>B $\rightarrow$ C	implied by @n(C):C with @l(A,B):C
@l(A,B):C	less or equal	A $\leq$ B $\rightarrow$ C	implied by @n(C):C with @>(A,B):C
@g(A,B):C	greater or equal	A $\geq$ B $\rightarrow$ C	implied by @n(C):C with @<(A,B):C
@=(A,B):C	equality	A=B $\rightarrow$ C	implied by @n(C):C with @#(A,B):C
@#(A,B):C	difference	A $\neq$ B $\rightarrow$ 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.

1.4.3.5 "is_" operators

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

1.4.3.6 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.

1.4.3.7 "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 "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.

1.5.3 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) = minor(B)* and *release(A)* $\geq$ *release(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.

1.5.4 Service "Find Parameter"

```
XPRM_SRV_PARAM: int findparm(const char *pname, int *type)
```

This function is used at compilation time to translate a parameter name into an internal code. At run time, the Mosel Virtual Machine uses this internal code for calling the special routines `getparam` and `setparam`. If the function returns a value smaller or equal to 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`.
This service must be defined if the module provides control parameters.

1.5.5 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 is being returned, the return value must be `NULL`.

1.5.6 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.

1.5.7 Service "Module Dependency List"

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

This service defines a table of modules that are required by a module and that must be loaded in order to be able to execute models using this module. 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`.

1.5.8 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`).

1.5.9 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.

1.6 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_ERROR, mydrv_error},
         {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)
```

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. This flag is reset after the stream is open and indicates a state of error everywhere else.
`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)

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", "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 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, if the stream is not open in silent mode, the error function is called to obtain a descriptive text for the error.

Note that this function is called from a critical section: only one file is open, closed, removed or renamed at a time even if several models are being executed concurrently.

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_ERROR is set.

This function must return 0 in case of success, all other values are interpreted as error codes. In the later case, and if the stream is not open in silent mode, the error function is called to obtain a descriptive text for the error.

Note that this function is called from a critical section: only one file is open, closed, removed or renamed at a time even if several model are being executed concurrently.

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, and if the stream is not open in silent mode, the error function is called to obtain a descriptive text for the error.

1.6.4 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, and if the stream is not open in silent mode, the error function is called to obtain a descriptive text for the error.

1.6.5 Operation "Error Message"

```
XPRM_IOCTRL_ERROR: int error(XPRMcontext ctx, void *stream, char *msg,
                             unsigned long len)
```

This function is called the first time one of the basic IO operations reports an error in order to get a descriptive text about the error. If a message is available, the function must return a non-zero value after having copied the text in the msg buffer (the message must be NUL terminated and not exceed len-1 characters). A return value of 0 indicates that no message is available.

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.

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 successfuly input line (*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).

Note that this function is called from a critical section: only one file is open, closed, removed or renamed at a time even if several models are being executed concurrently.

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.

Note that this function is called from a critical section: only one file is open, closed, removed or renamed at a time even if several models are being executed concurrently.

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 */
...
```

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

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

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

`getfirstsetnxd`, `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

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

Related topics

`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

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`.

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`.

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

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

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

Related topics

`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

These functions set the value of an array entry that corresponds to a given tuple of indices for the given array. The first form of this function selects the right value type based on the type of the array.

Related topics

`getarrval`, `regstring`.

2.4 Module types access

dsotyptostr

Purpose

Get a string representation from a module 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 string
`size`	Maximum length of the string

Return value

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

Further information

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

Related topics

`gettypeprop`, `dsotypfromstr`.

dsotypfromstr

Purpose

Initialise an object of a module type using a string.

Synopsis

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

Arguments

`ctx`	Mosel's execution context
`type`	Code of the external type
`ref`	Entity to initialise (must not be `NULL`)
`str`	Initialisation string

Return value

0 if successful, 1 otherwise.

Further information

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

Related topics

`gettypeprop`, `dsotyptostr`.

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
`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. It is therefore recommended to check whether the type supports this functionality before using this function (see `gettypeprop`).
2. This routine can also be used with structured user defined types (like sets, arrays or records).

Related topics

`gettypeprop`.

2.5 Record access

getnextfield

Purpose

Get the next field of a record type.

Synopsis

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

Arguments

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

Return value

Reference pointer for the next call to `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.

Related topics

`getfieldval`, `setfieldval`.

getfieldval

Purpose

Get the value of a field of a record.

Synopsis

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

Arguments

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

Further information

The field number must be obtained from the function `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 code, void *ref, int number,
        XPRMalltypes *value);
```

Arguments

`ctx` Mosel's execution context

`ref` Reference to the record

`code` Type code of the record

`number` Field number (in the record)

`value` Pointer to an area where the field value is 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 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 (only relevant for LP format — default is minimization)
`"s"`	Use scrambled names

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

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

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`.

getctrnum

Purpose

Get the row number of a linear constraint.

Synopsis

```
int getctrnum(XPRMlinctr ctr);
```

Argument

`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 main 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 $\geq$ constraint

`XPRM_CTYPE_LEQ` for $\leq$ 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 main 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(XPRMmpvar var);
```

Argument

`var` Reference to a variable

Return value

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

Further information

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

2.7 Dictionary access

findident

Purpose

Find an identifier in the dictionary.

Synopsis

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

Arguments

`ctx` Mosel's execution context

`text` Identifier

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

Return value

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

Example

See getnextproc

Further information

This function returns the dictionary entry of a given identifier for a given model, together with the type and structure of the entry. Type and structure are bit encoded and can be extracted using the macros `XPRM_TYP(t)` and `XPRM_STR(t)`.
The possible structures are:

`XPRM_STR_CONST` the object is a constant

`XPRM_STR_REF` the object is a reference to a scalar

`XPRM_STR_LIST` the object is a list

`XPRM_STR_SET` the object is a set

`XPRM_STR_ARR` the object is an array

`XPRM_STR_PROC` the object is a procedure or function

`XPRM_STR_MEM` the object is a memory block

`XPRM_STR_UTYP` the object is a user defined type

Depending on the structure, the possible types are:

`XPRM_TYP_NOT` no type (procedure)

`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).

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 or reference

`value.real` for constant or reference

`value.string` for constant or reference

`value.boolean` for constant or reference

`value.mpvar` for reference

`value.linctr` for reference

`value.list` for list (to be used as input for list functions)

`value.set` for set (to be used as input for set functions)

`value.array` for array (to be used as input for array functions) `value.ref` for a reference to an external type (available operations depend on the actual type)

`value.proc` for procedure and function

`value.memblk` for memory block

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

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

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

Related topics

getnextident.

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

Related topics

`findident`.

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

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 (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*'

`*` 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 code

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

Return value

0 if successful, 1 otherwise.

Further information

1. This function returns a property of an external type (types provided by modules or user defined). For the property `XPRM_TPROP_NAME`, the type name is returned in `value->string`, for the 2 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
3. The expanded code is available for user defined types only: it corresponds to the actual type code associated to a user defined type. For instance, assuming the type `myset` is defined as a set of integer, getting the type expansion for the code associated to `myset` will give `XPRM_STR_SET|XPRM_TYP_INT` indicating that a reference to an entity of type `myset` has to be handled with functions for sets.
4. Trying to get the expanded code of a module type is an error: the function returns 1. This can be used to identify module types.
5. A user type which expanded code is `XPRM_STR_REC` is a record type. The public fields of a record type may be enumerated with `getnextfield`.

Related topics

`getnextfield`.

2.8 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 tø 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. Is 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`.

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.

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`.

Further information

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 fucntion is called if the module has not yet been initialized (which is likely to happen when `getdsoctx` is called from within a reset function). If the second parameter is not `NULL`, it is used to return the value of the service `XPRM_SERV_IMCI` of the given module.

getdsoprop

Purpose

Get a property of a dynamic shared object.

Synopsis

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

Arguments

`dso` Reference to a module loaded by Mosel

`prop` Property to retrieve. Possible values:

- `XPRM_PROP_NAME` Module name
- `XPRM_PROP_ID` Internal number of the module
- `XPRM_PROP_VERSION` Version number
- `XPRM_PROP_SYSCOM` Identity of the provider if the module is certified
- `XPRM_PROP_NBREF` Number of loaded models that use the module

`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` and `SYSCOM`; 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"`).

Related topics

`getparam`.

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)
`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`.

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.

2.9 Input and output

dispmsg

Purpose

Display an error message.

Synopsis

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

Arguments

`ctx` Mosel's execution context

`format` *printf* style format

Further information

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`).

Related topics

`printf`.

fclose

Purpose

Close the file corresponding to the current input or output 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

Return value

0 if successful.

Further information

This function closes the file associated to the current input or output 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 or output 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.

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 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 or output 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

Return value

Stream number.

Further information

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

Related topics

`fselect`.

fgets

Purpose

Read a text string from the current input stream.

Synopsis

```
char *fgets(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 characters)

Return value

`s` if successful or `NULL` if the end of file has been reached.

Further information

This function reads 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 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_APPEND` Open for writing (append)
- `XPRM_F_LINBUF` If open for writing, flushes buffer after end of each line (default when writing to a console)
- `XPRM_F_SILENT` Do not display IO error messages

`fname` Name of the file to open

Return value

The stream number or -1 in case of error.

Further information

This function open a file and assign the resulting stream to the current input stream (file open for reading) or to the current output straem (file open for writing). The value returned can used as input for function `fselect`.

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 or output stream.

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

Arguments
`ctx` Mosel's execution context

`stream` Stream number

Further information
This function selects a stream as the current input or output 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`.

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 or output 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.

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. 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_ERROR` 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`.

2.10 Miscellaneous

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/structure of the object
`ref`	Reference of the object

Return value

The reference to the object (*i.e.* the third argument).

Further information

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_STR`, `XPRM_TYP_VAR`, `XPRM_TYP_CTL`, `XPRM_STR_ARR`, `XPRM_STR_LIST`, `XPRM_STR_SET` or the type code of an external type.

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/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_STR`, `XPRM_TYP_VAR`, `XPRM_TYP_CTL`, `XPRM_STR_ARR`, `XPRM_STR_LIST`, `XPRM_STR_SET` or the type code of an external type.

Related topics

`newref`.

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.

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

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.

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`.

regstring

Purpose

Register a text string.

Synopsis

```
const char *regstring(XPRMcontext ctx, const char *string);
```

Arguments

`ctx`	Mosel's execution context
`string`	Text string to register

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.

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

Arguments

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

Further information

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

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.

AIX:

```
cc -q32 -brtl -bnolibpath -D_THREAD_SAFE -G -I${MOSEL}/include
   -o mymodule.dso mymodule.c
```

HP-UX:

```
cc -c +Z +DAportable -D_POSIX_C_SOURCE=199506L -I${MOSEL}/include mymodule.c
ld mymodule.o -b +s -o mymodule.dso
```

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.

Index