

Generalized file handling in Mosel

FICO™ Xpress Optimization Suite whitepaper

Last update 26 May, 2009

Generalized file handling in Mosel

Y. Colombani and S. Heipcke

Xpress Team, FICO, Leam House, Leamington Spa CV32 5YN, UK

<http://www.fico.com/xpress>

26 May, 2009

Abstract

This paper describes the I/O drivers of the Mosel distribution with examples of their use. With only minimal changes to his models the user may switch between data sources of different formats. The interaction and exchange of data between a model and the application executing it can be made more immediate and as a consequence, more efficient. It is also possible to avoid the creation of physical intermediate files by performing all operations in memory. The latter may be useful, for instance, in distributed applications.

With the help of the Mosel Native Interface (NI) the user may also implement his own drivers. We introduce some applications of this feature: a driver for working with compressed files and an example generating C code with a Mosel model.

Contents

1	Introduction	1
2	Example problem	2
3	Data source access from Mosel models	4
3.1	Simplified use of ODBC	4
3.1.1	Alternative implementation	5
3.2	The excel driver	6
3.3	diskdata data file format	7
4	Exchange of information with embedded models	8
4.1	mem: working in memory	8
4.2	raw: binary data format	9
4.3	sysfd: working with system file descriptors	11
4.4	cb: output to callback functions	12
5	User-defined drivers	12
5.1	Example: compression	13
5.1.1	Implementation	13
5.2	Example: code generation	15
5.2.1	Implementation	15
6	Summary	18

1 Introduction

Release 1.4 of Mosel introduces a generalization of the notion 'file'. So far, in statements of the Mosel language (such as initializations `from/to`, `fopen`, and `fclose`) and in the Mosel library functions (e.g. `XPRMcompmod`, `XPRMloadmod`, or `XPRMrunmod`) the term 'file' always implied 'physical file handled by the operating system'. From now on, a 'file' may be, for instance,

- a physical file (text or binary)
- a block of memory
- a file descriptor provided by the operating system
- a function (callback)
- a database

The type of the file is indicated by an extended file name: the actual file name is preceded by the name of the driver that is to be used to access it. We now write, for example, `mem:filename` to indicate that we work with a file held in memory, or `mmodbc.odbc:mydata.mdb` to access an MS Access database via the ODBC driver provided by the module `mmodbc`. The default driver (no driver prefix) is the standard Mosel file handling, meaning that no changes are required to any existing applications.

This paper describes the I/O drivers of the Mosel distribution with examples of their use. Some drivers provide interfaces to specific data sources (such as ODBC). Others serve to exchange information between the application running the Mosel libraries and a Mosel model in a very direct way by providing various possibilities of passing data back and forth in memory.

With the help of the Mosel Native Interface (NI) the user may also implement his own drivers. We introduce some applications of this feature: a driver for working with compressed files and an example generating C code with a Mosel model.

2 Example problem

The following small *knapsack problem* will be used as an example in the next sections.

A burglar sees 8 items, of different worths and weights. He wants to take the items of greatest total value whose total weight is not more than the maximum WTMAX he can carry.

We introduce binary variables $take_i$ for all i in the set of all items (*ITEMS*) to represent the decision whether item i is taken or not. $take_i$ has the value 1 if item i is taken and 0 otherwise. Furthermore, let $VALUE_i$ be the value of item i and $WEIGHT_i$ its weight. A mathematical formulation of the problem is then given by:

$$\begin{aligned} & \text{maximize } \sum_{i \in \text{ITEMS}} VALUE_i \cdot take_i \\ & \sum_{i \in \text{ITEMS}} WEIGHT_i \cdot take_i \leq WTMAX \quad (\text{weight restriction}) \\ & \forall i \in \text{ITEMS} : take_i \in \{0, 1\} \end{aligned}$$

The objective function is to maximize the total value, that is, the sum of the values of all items taken. The only constraint in this problem is the weight restriction.

The following Mosel model (`burglar2.mos`) implements this problem. In this implementation data are read from the text file `burglar.dat`. Note that the decision variables are declared after the initialization of the data, that is, at a point in the model where the contents of the set *ITEMS* is known. If an array is declared before its index sets are known it is created as a dynamic array. Dynamic arrays of basic types (such as `VALUE` and `WEIGHT`) grow by adding data entries, but for dynamic arrays of decision variables the entries need to be created explicitly once the index sets are known. By moving the declaration of the variables after the initialization of the data we can avoid this.

```

model Burglar2
uses "mmxprs"

declarations
  WTMAX = 102                ! Maximum weight allowed
  ITEMS: set of string      ! Index set for items
  VALUE: array(ITEMS) of real ! Value of items
  WEIGHT: array(ITEMS) of real ! Weight of items
end-declarations

initializations from "burglar.dat"
  [VALUE,WEIGHT] as "BurgData"
end-initializations

declarations
  take: array(ITEMS) of mpvar ! 1 if we take item i; 0 otherwise
end-declarations

! Objective: maximize total value
MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
forall(i in ITEMS) take(i) is_binary

maximize(MaxVal)                ! Solve the MIP-problem

! Solution output
forall(i in ITEMS) SOLTAKE(i):= getsol(take(i))

writeln("Solution:\n Objective: ", getobjval)
writeln(SOLTAKE)

initializations to "burglarout.txt"
  SOLTAKE
end-initializations

end-model

```

The data file `burglar.dat` read by this model has the following contents.

```

! Data file for 'burglar2.mos'
! Item      Value  Weight
BurgData: [(camera)  [ 15  2]
           (necklace) [100 20]
           (vase)     [ 90 20]
           (picture)  [ 60 30]
           (tv)       [ 40 40]
           (video)    [ 15 30]
           (chest)    [ 10 60]
           (brick)    [  1 10]]

```

The solution is output to the file `burglarout.txt` via `initializations to`. This prints the array `SOLTAKE` to a text file in `initializations` format:

```

'SOLTAKE': [('camera') 1 ('necklace') 1 ('vase') 1 ('picture') 1 ('tv') 0
           ('video') 1 ('chest') 0 ('brick') 0]

```

3 Data source access from Mosel models

Accessing data files with formats other than the `initializations` text file format has always been possible in Mosel with the help of modules that define additional functions specific to a format or data source type. It is now possible to use other formats in the `initializations` block, and hence to minimize the changes required in a model to switch between different data formats.

The drivers `odbc`, `excel` and `diskdata` described in this section are used in the `initializations` block of the Mosel language to provide access to new data file formats and data sources. Some of the drivers described in Section 4 (e.g. `raw` and `mem`) also provide such a functionality. However, they are only of use in models embedded into an application and run through the Mosel libraries, not in stand-alone Mosel models as is the case with the `odbc`, `excel` and `diskdata` drivers.

3.1 Simplified use of ODBC

The `odbc` I/O driver defined by the module `mmodbc` automatically generates the SQL queries that are required to read in data from an external data source (database or spreadsheet) or insert data into this data source. The following modifications need to be made to model `Burglar2` to switch to file access via ODBC:

```

model "Burglar2 (ODBC)"
  uses "mmodxprs"

  parameters
    CNCTIO = ''
  end-parameters
  ...
  initializations from CNCTIO
    [VALUE,WEIGHT] as "BurgData"
  end-initializations
  ...
  ! Insert solutions into database/spreadsheet: results from previous runs
  ! must be removed previously; otherwise the new results will either be
  ! appended to the existing ones or, if 'ITEM' has been defined as key field
  ! in a database, the insertion may fail.
  initializations to CNCTIO
    SOLTAKE as "SolTake"
  end-initializations
end-model

```

The *connection string* `CNCTIO` indicates the data source, for example:

- `CNCTIO = "mmodbc.odbc:burglar.xls"` for the MS Excel spreadsheet `burglar.xls`
- `CNCTIO = "mmodbc.odbc:burglar.mdb"` for the MS Access database `burglar.mdb`
- `CNCTIO = "mmodbc.odbc:DSN=mysql;DB=burglar"` for the mysql database `burglar`

To run this example, the ODBC driver for the corresponding data source must be present. The connection string may vary depending on the installation of the ODBC driver. For further detail on setting up an ODBC connection and working with ODBC in Mosel see the Xpress Whitepaper [‘Using ODBC and other database interfaces with Mosel’](#).

In the case of database connections, the data are read from a table called `BurgData` that contains (at least) a field for the indices (e.g., labeled `ITEM`), and the fields `VALUE` and `WEIGHT`. The results are written into the table `SolTake`, containing a field for the indices (e.g., `ITEM`) and

a field to receive the solution values (e.g., called `TAKE`). In a spreadsheet, `BurgData` and `SolTake` are the names of named ranges. The columns of the named ranges must bear headers (to be included in the selected range area), these headers are not used by the (generated) SQL commands and may therefore differ from the names used in the model. The range `BurgData` must have three columns, for instance bearing the headers `ITEM`, `VALUE`, and `WEIGHT`, and the range `SolTake` must have a column to receive the indices (e.g., labeled `ITEM`) and a second one to receive the solution values (e.g., `TAKE`).

The ODBC driver may take several options, see the section '`mmodbc`' in the '[Mosel Language Reference Manual](#)' for further detail.

3.1.1 Alternative implementation

The model `Burglar2` may be implemented as follows using standard `mmodbc` functionality.

```

model "Burglar2 (SQL)"
  uses "mmxprs", "mmodbc"

  parameters
    CNCT = ''
  end-parameters
  ...
  ! Reading data from file
  SQLconnect (CNCT)
  SQLexecute("select * from BurgData", [VALUE,WEIGHT])
  SQLdisconnect
  ...
  ! Solution output
  SQLconnect (CNCT)
  SQLexecute("delete from SolTake")      ! Cleaning up previous results: works
                                       ! only for databases, cannot be used
                                       ! with spreadsheets (instead, delete
                                       ! previous solutions directly in the
                                       ! spreadsheet file)
  SQLexecute("insert into SolTake values (?,?)", SOLTAKES)
  SQLdisconnect
end-model

```

As before, the connection string `CNCT` indicates the data source:

- `CNCT = "DSN=Excel Files;DBQ=burglar.xls"` for the MS Excel spreadsheet `burglar.xls`
- `CNCT = "DSN=MS Access Database;DBQ=burglar.mdb"` for the MS Access database `burglar.mdb`
- `CNCT = "DSN=mysql;DB=burglar"` for the mysql database `burglar`

All of these functions may be used in conjunction with the ODBC I/O driver. For instance, when working with a database it may be helpful to make the lines

```

SQLconnect (CNCT)
SQLexecute("delete from SolTake")

```

precede the insertion of the solution into the database to clean up previous results.

3.2 The excel driver

The (ab)use of the ODBC database access technology for communicating with spreadsheets sometimes causes unsatisfactory behavior of the spreadsheet software. In particular, Microsoft Excel spreadsheets are not updated correctly if they are open while ODBC tries to write to them and it is not possible to overwrite data output by previous model runs.

These drawbacks are remedied by the *excel* I/O driver, a software-specific I/O driver that accesses directly Excel spreadsheets without passing through ODBC. This driver supports all basic data access tasks (but none of the advanced SQL statements). If the spreadsheet file is kept opened while running the Mosel model the output is written to the spreadsheet without saving it. In addition, with this driver output always starts at the same place (that is, previous output gets overwritten).

```

model "Burglar2 (Excel)"
  uses "mmxprs"

  parameters
    CNCTIO = 'mmodbc.excel:burglar.xls'
  end-parameters
  ...

! Spreadsheet range includes header line -> option 'skip' to skip header
initializations from CNCTEXC
  [VALUE,WEIGHT] as "skip;BurgData"
end-initializations
...

! Insert solutions into spreadsheet: results from previous runs
! are overwritten by new output
! Only first line of output range is specified -> option 'grow'
initializations to CNCTEXC
  SOLTAKE as "skip;grow;SolTake"
end-initializations
end-model

```

The input and output ranges in the spreadsheet used with the *excel* driver contain just the data, no column headers. To work with the ODBC format of ranges the option `skip` needs to be used as shown in the example. Another option employed when outputting data is `grow`, this indicates that the output range is specified by a single row and may grow on demand.

Instead of naming the ranges in the spreadsheet it is equally possible to work directly with the cell references for the input and output ranges (including the worksheet name, which is 'burglar' in our case):

```

model "Burglar2 (Excel)"
  ...
  initializations from CNCTEXC
    [VALUE,WEIGHT] as "[burglar$B4:D11]"
  end-initializations
  ...

! Insert solutions into spreadsheet: results from previous runs
! are overwritten by new output
! Only first line of output range is specified -> option 'grow'
initializations to CNCTEXC
  SOLTAKE as "grow;[burglar$F4:G4]"
end-initializations
end-model

```

For further detail on this driver the reader is referred to the section '*mmodbc*' of the '*Mosel*'

Language Reference Manual.

3.3 `diskdata` data file format

With only minimal changes to the previous model we now switch to an entirely different data format, namely the `diskdata` text file format that is accessed via the `diskdata` I/O driver.

```
model Burglar2dd
...
initializations from "mmetc.diskdata:sparse;noq"
  [VALUE,WEIGHT] as "burglardd.dat"
end-initializations
...
initializations to "mmetc.diskdata:append,sparse"
  SOLTAKE as "burglarout.txt"
end-initializations
end-model
```

This will result in exactly the same behaviour as when using the procedure `diskdata` of the module `mmetc` instead of the `initializations` blocks.

```
model Burglar2ddb
  uses "mmetc", "mmxprs"
  ...
  ! Reading data from file
  diskdata(ETC_SPARSE+ETC_NOQ, "burglardd.dat", [VALUE,WEIGHT])
  ...
  ! Solution output
  diskdata(ETC_OUT+ETC_APPEND+ETC_SPARSE, "burglarout.txt", SOLTAKE)
end-model
```

The data file `burglardd.dat` used in both cases has the following contents.

```
! Data file for 'burglar2dd.mos'
camera, 15, 2
necklace, 100, 20
vase, 90, 20
picture, 60, 30
tv, 40, 40
video, 15, 30
chest, 10, 60
brick, 1, 10
```

The result file `burglarout.txt` is now formatted as shown here.

```
"camera",1
"necklace",1
"vase",1
"picture",1
"tv",0
"video",1
"chest",0
"brick",0
```

The `diskdata` I/O driver and the procedure `diskdata` are documented in the section '`mmetc`' in the ['Mosel Language Reference Manual'](#).

4 Exchange of information with embedded models

In this section we show several examples of the use of I/O drivers in the Mosel distribution. The full documentation of these drivers is contained in the '[Mosel Libraries Reference Manual](#)'.

4.1 mem: working in memory

The *mem* I/O driver makes it possible to use a block of memory as a data source. The following example shows how to use this driver to compile a Mosel model to memory (instead of creating a physical BIM file). This means that the resulting application does not generate any auxiliary file for handling the model. This feature may be of interest especially in distributed systems.

In the code printed below and in all subsequent examples the error handling is omitted for clarity's sake. The full examples are on the [Xpress website](#).

```
#include <stdio.h>
#include "xprm_mc.h"

int main()
{
    XPRMmodel mod;
    int result;
    char bimfile[2000];          /* Buffer to store BIM file */
    unsigned long bimfile_size; /* Buffer to store actual size of BIM file */
    char bimfile_name[64];      /* File name of BIM file */

    XPRMinit();                /* Initialize Mosel */

    /* Prepare file name for compilation using 'mem' driver: */
    /* "mem:base address/size[/actual size pointer]" */
    bimfile_size=0;
    sprintf(bimfile_name, "mem:%#lx/%u/%#lx",
            (unsigned long)bimfile, sizeof(bimfile), (unsigned long)&bimfile_size);

                                /* Compile model file to memory */
    XPRMcompmod(NULL, "burglar2.mos", bimfile_name, "Knapsack example");
    printf("BIM file uses %lu bytes of memory.\n", bimfile_size);

                                /* Load the BIM file from memory */
    mod=XPRMloadmod(bimfile_name, NULL);

    XPRMrunmod(mod, &result, NULL); /* Run the model */

    return 0;
}
```

The only change to standard use of the Mosel library functions is the replacement of the (physical) BIM file by an extended filename in function `XPRMcompmod`. The extended filename indicates the driver we want to use (`mem`), an address where to store the BIM file, and the space reserved for it. Notice that the size of the resulting BIM file is not known before compilation. In this example we reserve roughly twice the size of the model file for the BIM file. However, there is no guarantee that this is sufficient for other models. We therefore print out the size of the generated file as a control.

We may take working in memory one step further, by including the source of the Mosel model directly in the C source code. After compilation, we obtain a single all-in-one application file that does not require or generate any auxiliary file for handling the model.

The source of the model is included in the C source file `iodrvmem.c` as an array of characters:

```

const char source_of_model[]=
"model Burglar\n"
"uses 'mxxprs'\n"

"declarations\n"
" WTMAX = 102                ! Maximum weight allowed\n"
" ITEMS = {'camera', 'necklace', 'vase', 'picture', 'tv', 'video', \n"
"         'chest', 'brick'} ! Index set for items\n"
" VALUE: array(ITEMS) of real ! Value of items\n"
" WEIGHT: array(ITEMS) of real ! Weight of items\n"
" take: array(ITEMS) of mpvar ! 1 if we take item i; 0 otherwise\n"
"end-declarations\n"

"VALUE::(['camera', 'necklace', 'vase', 'picture', 'tv', 'video', \n"
"        'chest', 'brick']) [15,100,90,60,40,15,10,1]\n"
"WEIGHT::(['camera', 'necklace', 'vase', 'picture', 'tv', 'video', \n"
"         'chest', 'brick']) [2,20,20,30,40,30,60,10]\n"

"! Objective: maximize total value\n"
"MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)\n"

"! Weight restriction\n"
"sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX\n"
"! All variables are 0/1\n"
"forall(i in ITEMS) take(i) is_binary\n"

"maximize(MaxVal)                ! Solve the problem\n"

"! Print out the solution\n"
"writeln(\"Solution:\n Objective: \", getobjval)\n"
"forall(i in ITEMS) writeln(' take(', i, '): ', getsol(take(i)))\n"

"end-model";

```

In the compilation command we replace the filename `burglar2.mos` by an extended filename, consisting in the driver to be used (`mem`) and the address and size of the model source.

```

char mosfile_name[40];          /* File name of MOS file */

sprintf(mosfile_name, "mem:%#lx/%u",
        (unsigned long)source_of_model, sizeof(source_of_model));

/* Compile model from memory to memory */
XPRMcompmod(NULL, mosfile_name, bimfile_name, "Knapsack example");

```

The data in this Mosel model are hardcoded directly in the model. In the following section we show how to extend this example with data input from memory, using the `raw` driver.

4.2 `raw`: binary data format

The `raw I/O` driver provides an implementation of the `initializations` block in binary mode: instead of translating information from/to text format, data is kept in its raw representation. The following example (file `iodrvraw.c`) shows how to use this driver in combination with the `mem` driver to exchange arrays of data between a model and an application through memory. As shown in the previous section, we may work with a separate model file (this example) or alternatively include the model source directly in the model.

```

#include <stdio.h>
#include "xprm_mc.h"

const struct

```

```

{
    /* Initial values for array 'data': */
    const char *ind;          /* index name */
    double val,wght;         /* value and weight data entries */
} data[]={{"camera",15,2}, {"necklace",100,20}, {"vase",90,20},
          {"picture",60,30}, {"tv",40,40}, {"video",15,30},
          {"chest",10,60}, {"brick",1,10}};

double solution[8];        /* Array for solution values */

int main()
{
    XPRMmodel mod;
    int result;
    XPRMalltypes rvalue, itemname;
    XPRMset set;
    char bimfile[2000];     /* Buffer to store BIM file */
    unsigned long bimfile_size; /* Buffer to store actual size of BIM file */
    char bimfile_name[64]; /* File name of BIM file */
    char data_name[40];    /* File name of initial values for 'data' */
    char solution_name[40]; /* File name of solution values */
    char params[96];      /* Parameter string for model execution */

    XPRMinit();           /* Initialize Mosel */

    /* Prepare file name for compilation using 'mem' driver: */
    /* "mem:base address/size[/actual size pointer]" */
    bimfile_size=0;
    sprintf(bimfile_name, "mem:%#lx/%u/%#lx",
            (unsigned long)bimfile, sizeof(bimfile), (unsigned long)&bimfile_size);

    /* Compile model file to memory */
    XPRMcompmod(NULL, "burglar2r.mos", bimfile_name, "Knapsack example");
    printf("BIM file uses %lu bytes of memory.\n", bimfile_size);

    /* Load a BIM file from memory */
    mod=XPRMloadmod(bimfile_name, NULL);

    /* Prepare file names for 'initializations' using the 'raw' driver: */
    /* "rawoption[,...],filename" */
    /* (Here, 'filename' uses the 'mem' driver, data is stored in memory) */
    /* Options for 'raw': */
    /* 'slength=0': strings are represented by pointers to null terminated */
    /* arrays of characters (C-string) instead of fixed size arrays*/
    /* 'noindex': only array values are expected - no indices requested */

    sprintf(data_name, "slength=0,mem:%#lx/%u", (unsigned long)data,
            sizeof(data));
    sprintf(solution_name, "noindex,mem:%#lx/%u", (unsigned long)solution,
            sizeof(solution));

    /* Pass file names as execution param.s */
    sprintf(params, "DATA='%s',SOL='%s'", data_name, solution_name);

    XPRMrunmod(mod, &result, params); /* Run the model */

    /* Display solutions values obtained from the model */
    printf("Objective: %g\n", XPRMgetobjval(mod));
    XPRMfindident(mod, "ITEMS", &rvalue); /* Get the model object 'ITEMS' */
    set = rvalue.set;
    for(i=0;i<8;i++)
        printf(" take(%s): %g\n", XPRMgetelsetval(set, i+1, &itemname)->string,
            solution[i]);

    return 0;
}

```

The input data in the C code above is stored and communicated to the model in *sparse format*,

that is, each data item with its index tuple. For retrieving and saving the solution data this example uses *dense format*, that is, just the data items without explicitly specifying the indices. To obtain a printout with the (string) index names, we retrieve separately the set of indices from the Mosel model. Alternatively, we could also use *sparse format* for the solution data similarly to the input data, in which case we did not have to retrieve the index set separately.

The model file `burglar2r.mos` of this example is very similar to the model versions for the *odbc* and *diskdata* I/O drivers in Section 3. The only changes made are to the name of the I/O driver and to the strings `DATA` and `SOL` for the data source and the destination of results respectively.

```

model Burglar2r
  uses 'mmxprs'

  parameters
    DATA=''
    SOL=''
    WTMAX = 102                ! Maximum weight allowed
  end-parameters

  declarations
    ITEMS: set of string      ! Index set for items
    VALUE: array(ITEMS) of real ! Value of items
    WEIGHT: array(ITEMS) of real ! Weight of items
    SOLTAKE: array(ITEMS) of real ! Solution values
  end-declarations

  initialisations from 'raw:'
    [VALUE,WEIGHT] as DATA
  end-initialisations

  declarations
    take: array(ITEMS) of mpvar ! 1 if we take item i; 0 otherwise
  end-declarations

  ! Objective: maximize total value
  MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

  ! Weight restriction
  sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

  ! All variables are 0/1
  forall(i in ITEMS) take(i) is_binary

  maximize(MaxVal)                ! Solve the problem

  ! Solution output
  forall(i in ITEMS) SOLTAKE(i) := getsol(take(i))

  initialisations to 'raw:'
    SOLTAKE as SOL
  end-initialisations

end-model

```

4.3 sysfd: working with system file descriptors

With the *sysfd* driver a file descriptor provided by the operating system can be used in place of a file. File descriptors are returned by the C functions `open` or `fileno`. They also include the default input, output, and error streams (values 0, 1, and 2 respectively) that are opened automatically when a program starts. For instance, to redirect the error stream for a model `mod` to the default output we may write:

```
XPRMsetdefstream(mod, XPRM_F_ERROR, "sysfd:1");
```

4.4 `cb`: output to callback functions

Using the `cb` I/O driver a function is used as a file. In the following example (file `iodrvcb.c`) firstly we redirect the error stream to the default output stream using the `sysfd` I/O driver. We then define a callback function to receive the output stream. Our function simply prints all output lines preceded by Mosel: .

```
#include <stdio.h>
#include "xprm_mc.h"

/* Callback function to handle output */
long XPRM_RTC cbmsg(XPRMmodel model, void *info, char *buf, unsigned long size)
{
/* Note: 'model' is NULL if the stream is used outside of an execution */
printf("Mosel: %.*s", (int)size, buf);
return 0;
}

int main()
{
XPRMmodel mod;
int result;
char outfile_name[40];          /* File name of output stream */

XPRMinit();                    /* Initialize Mosel */
                                /* Using 'sysfd' driver: */
                                /* Redirect error stream to default output */
XPRMsetdefstream(NULL, XPRM_F_ERROR, "sysfd:1");

                                /* Prepare file name for output stream */
                                /* using 'cb' driver: */
                                /* "cb:function pointer[/callback data]" */
sprintf(outfile_name, "cb:%#lx", (unsigned long)cbmsg);

                                /* Set default output stream to callback */
XPRMsetdefstream(NULL, XPRM_F_WRITE, outfile_name);

                                /* Execute = compile/load/run model file
                                'burglar2.mos'
                                Generates the BIM file 'burglar2.bim' */
XPRMexecmod(NULL, "burglar2.mos", NULL, &result, NULL);

return 0;
}
```

The output produced by this program looks as follows:

```
Mosel: Solution:
Mosel: Objective: 280
Mosel: [('camera',1),('necklace',1),('vase',1),('picture',1),('tv',0),
('video',1),('chest',0),('brick',0)]
```

5 User-defined drivers

The contents of this section is quite advanced material and we recommend its reading only to those who are interested in the Mosel Native Interface (NI).

In this section we discuss two examples of user-defined drivers, performing compression and code generation respectively. New drivers are defined via the NI, in the form of *services*. For every driver some or all of a list of standard I/O functions may be defined, depending on its purpose.

The full source code of both examples is provided with the examples of the Mosel distribution. For more detail on the implementation of new I/O drivers see the '[Mosel NI User Guide](#)' and '[Mosel NI Reference Manual](#)'.

5.1 Example: compression

With the *gzip* I/O driver for file compression implemented by the example module *zlib*, compressed files may be used for Mosel input and output. For example, we may *generate and load a compressed BIM file* with the following command:

```
mosel -c "comp mymodel.mos 'example of compression' zlib.gzip:mymodel.gz"
mosel -c "load zlib.gzip:mymodel.gz"
```

To use *compressed data files* in a model we may write:

```
initializations from "zlib.gzip:burglar.dat.gz"
```

Another use of this driver may be to generate *compressed matrix files*:

```
exportprob("zlib.gzip:prob.mat.gz")
```

The I/O driver *gzip* implements an interface to the *gzip* compression format of the *ZLIB* library. The complete example module *zlib* also implements a second driver, *compress*, for a different compression format defined by this library. The driver *gzip* works with physical files, *compress* is a more general driver that works with streams. The *ZLIB* library is required to execute this example: it can be found at <http://www.zlib.org>.

5.1.1 Implementation

A C program defining new I/O drivers has the following components

- **Interface structures:** the DSO interface table, the table of services, the table of drivers, and a table of functions per driver.
- **Module initialization function**
- **Implementation of driver access functions:** opening and closing streams, standard reading and writing, special reading and writing for *initializations* blocks, error handling, deleting and moving files.

A driver must implement at least the stream open functionality and one of the standard or special reading or writing functions; everything else is optional.

The definition of the *interface structures* of module *zlib* is the following. The structure of all tables and the function prototypes are defined by the Mosel NI.

```
/* Functions of the 'gzip' driver */
static void *gzip_open(XPRMcontext ctx, int *mode, const char *fname);
static int gzip_close(XPRMcontext ctx, gzFile gzf, int mode);
static long gzip_read(XPRMcontext ctx, gzFile gzf, void *buffer,
                    unsigned long size);
static long gzip_write(XPRMcontext ctx, gzFile gzf, void *buffer,
                    unsigned long size);
static XPRMiofcttab iodrv_gzip[]=
{
    {XPRM_IOCTL_OPEN, gzip_open},
}
```

```

        {XPRM_IOCTL_CLOSE, gzip_close},
        {XPRM_IOCTL_READ, gzip_read},
        {XPRM_IOCTL_WRITE, gzip_write},
        {XPRM_IOCTL_INFO, "filename"},
        {0, NULL}
    };

    /* Drivers of the zlib module */
static XPRMiodrvtab iodrv_zlib[]=
    {
        {"gzip", iodrv_gzip},
        {NULL, NULL}
    };

    /* Table of services: only IO drivers */
static XPRMdsoerv tabserv[]=
    {
        {XPRM_SRV_IODRVS, iodrv_zlib}
    };

    /* DSO interface: only services */
static XPRMdsointer dsointer=
    {
        0, NULL,
        0, NULL,
        0, NULL,
        sizeof(tabserv)/sizeof(mm_dsoserv), tabserv
    };

static XPRMnifct mm;          /* For storing Mosel NI function table */

```

The previous code extract is best read from bottom to top. It declares an object of type `XPRMnifct` to store the NI function table (to be retrieved during the initialization of the module). The main DSO interface table `dsointer` lists the functionality provided by this module: it only defines services. The table of services, `tabserv`, indicates which services are defined, namely new I/O drivers. The list of I/O drivers is given in the table `iodrv_zlib`. The functionality implemented by the I/O driver `gzip` is listed in the table `iodrv_gzip`. The function bodies of `gzip_open`, `gzip_close`, etc., are printed below. The string set with `INFO` will be used by the 'help' command of Mosel to indicate the usage of this driver (it must be used with the name of a physical file, not with extended filenames).

The *module initialization function* performs the standard initialization of Mosel modules: retrieving the Mosel NI function table and the NI version number and returning the module DSO interface table to Mosel. In addition, we check the version of the ZLIB library.

```

DSO_INIT zlib_init(XPRMnifct nifct, int *interver, int *libver,
                  XPRMdsointer **interf)
{
    const char *zlv;

    mm=nifct;          /* Save the table of Mosel NI functions */
    *interver=XPRM_NIVERS; /* The interface version we are using */

    /* We check the ZLIB library version here */
    zlv=zlibVersion();
    if((zlv==NULL)|| (zlv[0]!=ZLIB_VERSION[0]))
    {
        mm->dispmg(NULL, "ZLIB: wrong version (expect %s got %s).\n", ZLIB_VERSION,
                    (zlv!=NULL)?zlv:"0");
        return 1;
    }
    else
    {
        *libver=XPRM_MKVER(0,0,1); /* The version of the module: 0.0.1 */
        *interf=&dsointer; /* Our interface */
        return 0;
    }
}

```

```
}

```

Below are printed the *functions* implementing the *gzip* driver. Besides the obligatory 'open' function the *gzip* driver implements the functions 'closing a file', 'reading' (= uncompress), and 'writing' (= compress).

```

/**** Open a gzip-file for (de)compression ****/
static void *gzip_open(XPRMcontext ctx, int *mode, const char *fname)
{
  char cmode[16];
  int cml;

  cml=2;
  cmode[0]=((*mode)&XPRM_F_WRITE)?'w':'r';
  cmode[1]=((*mode)&XPRM_F_BINARY)?'b':'t';
  if((*mode)&XPRM_F_APPEND) cmode[cml++]='a';
  cmode[cml]='\0';
  return gzopen(fname,cmode);
}

/**** Close the gzip-file ****/
static int gzip_close(XPRMcontext ctx, gzFile gzf, int mode)
{ return gzclose(gzf); }

/**** Uncompress a block of data ****/
static long gzip_read(XPRMcontext ctx, gzFile gzf, void *buffer,
                    unsigned long size)
{ return gzread(gzf, buffer, size); }

/**** Compress a block of data ****/
static long gzip_write(XPRMcontext ctx, gzFile gzf, void *buffer,
                    unsigned long size)
{ return gzwrite(gzf, buffer, size); }

```

As mentioned earlier, the full *zlib* example module in the module examples of the Mosel distribution defines a second compression driver, *compress*, in a very similar way to what has been shown here for the *gzip* driver.

5.2 Example: code generation

In this section we present a driver that includes the output of Mosel into a C program. It will typically be used for generating a C program that includes the BIM file of a given model, for example using a command like

```
mosel -c "comp mymodel.mos '' export.toC:mymodel.c"
```

As with all Mosel I/O drivers, the *toC* driver may be combined with other drivers. For instance, we may use the *compress* driver of the *zlib* module to obtain a compressed BIM file within the generated C program:

```
mosel -c "comp mymodel.mos '' zlib.compress:export.toC:mymodel.c"
```

5.2.1 Implementation

The module *export* that defines the *toC* I/O driver has the following components (common to all I/O driver modules):

- Interface structures
- Module initialization function
- Implementation of driver access functions

The file `export.c` implementing the module `export` with the I/O driver `toC` starts with the definition of the C code template, divided into three parts:

```
static char prg_part1[]=
"#include <stdio.h>\n"
"#include \"xprm_rt.h\"\n\n"

"static unsigned int bimfile[]={";

static char prg_part2[]=
"0};\n\n"

"int main(int argc,char *argv[])\n"
"{\n"
" char modname[40];\n"
" XPRMmodel mod;\n"
" int rts;\n\n"

" rts=XPRMinit();\n"
" if((rts!=0)&&(rts!=32))\n"
" {\n"
" char msg[512];\n\n"

" XPRMgetlicerrmsg(msg,512);\n"
" fprintf(stderr, \"%s\",msg);\n"
" return 1;\n"
" }\n\n"

" sprintf(modname, \"mem:%#lx/%u\", \n"
"         (unsigned long)bimfile,");

static char prg_part3[]=
"         ");\n\n"
" if((mod=XPRMloadmod(modname,NULL))==NULL)\n"
" return 2;\n"
" if(XPRMrunmod(mod,&rts,NULL))\n"
" return 3;\n"
" else\n"
" return rts;\n"
"}\n";
```

The generated output will be inserted in between parts 1 and 2, its size will be inserted in between parts 2 and 3.

The definition of the NI *interface structures* follows the same scheme like what we have seen in the previous example: the table of driver functions, the table of drivers, the table of services, the main DSO interface table, and an address for storing the NI function table. The `toC` driver implements 'open', 'close', and 'write' functionality and works with generalized files.

```
static void *toc_open(XPRMcontext ctx, int *mode, const char *fname);
static int toc_close(XPRMcontext ctx, s_tocdata *td, int mode);
static long toc_write(XPRMcontext ctx, s_tocdata *td, char *buffer,
                    unsigned long size);

static XPRMiofcttab iodrv_toc[]=
{
    {XPRM_IOCTL_OPEN, (void *)toc_open},
    {XPRM_IOCTL_CLOSE, (void *)toc_close},
    {XPRM_IOCTL_WRITE, (void *)toc_write},
}
```

```

        {XPRM_IOCTL_INFO, "extended_filename"},
        {0, NULL}
    };

    /* Drivers of module 'export': toC */
static XPRMiodrvtab iodrv_export[]=
    {
        {"toC", iodrv_toc},
        {NULL, NULL}
    };

    /* Table of services: only I/O drivers */
static XPRMdsoserv tabserv[]=
    {
        {XPRM_SRV_IODRVS, iodrv_export}
    };

    /* DSO interface: only services */
static XPRMdsointer dsointer=
    {
        0, NULL,
        0, NULL,
        0, NULL,
        sizeof(tabserv)/sizeof(mm_dsoserv), tabserv
    };

static XPRMnifct mm;          /* For storing Mosel NI function table */

```

This module's *initialization function* performs the standard module initialization procedure where the module's interface structure is passed on to Mosel and the NI function table is saved for use by this module.

```

DSO_INIT export_init(XPRMnifct nifct, int *interver, int *libver,
                    XPRMdsointer **interf)
{
    mm=nifct;          /* Save the table of Mosel NI functions */
    *interver=XPRM_NIVERS; /* The interface version we are using */
    *libver=XPRM_MKVER(0,0,1); /* The version of the module: 0.0.1 */
    *interf=&dsointer; /* Our interface */
    return 0;
}

```

The *functions* implemented by this I/O driver are restricted to opening and closing a file and writing to a file. The 'open' and 'close' functions print the predefined part of the generated C file. The function for writing the binary part of the output (`toc_write`) is somewhat tricky since it needs to make sure that all bytes are aligned correctly. This function is called with a buffer as parameter. For every four bytes it prints the corresponding integer. Since the buffer size is not necessarily a multiple of 4 there may be a few bytes left over at the end of the buffer which must be carried over to the next call to this function or to the 'close' function. Below we print the 'open' and 'close' functions. The definition of the 'write' function is left out here, the interested reader may be reminded that the complete module source is a part of the Mosel module examples.

```

static void *toc_open(XPRMcontext ctx, int *mode, const char *fname)
{
    s_tocdata *td;

    /* First time: open actual file */
    if(mm->fopen(ctx, *mode, fname)<0)
    {
        *mode|=XPRM_F_SILENT; /* Error message already displayed */
        /* switch so silent mode */
        return NULL;
    }
    else
    {

```

```

    td=(s_tocdata *)malloc(sizeof(s_tocdata));
    td->total=0;
    td->nb_remain=0;
    td->nb_printed=0;
    mm->printf(ctx, "%s\n", prg_part1); /* Display the beginning of program */
    return td;
}
}

static int toc_close(XPRMcontext ctx, s_tocdata *td, int mode)
{
    if(td->nb_remain>0) /* Send bytes to be written */
    {
        td->total+=td->nb_remain;
        for(;td->nb_remain<4;td->nb_remain++)
            td->map.c[td->nb_remain]=0;
        mm->printf(ctx, "%#x, ", td->map.i);
    }
    mm->printf(ctx, "%s%u%s", prg_part2, td->total, prg_part3);
    return mm->fclose(ctx,mode);
}

```

These two functions work with the following structure for storing information concerning the printed output. Function 'open' initializes the structure (`td`), function 'write' copies the remaining bytes from every buffer into this structure and sets the counters, and function 'close' checks whether there is anything left to be printed before completing and closing the output file.

```

typedef struct
{
    unsigned int total; /* Total number of bytes written so far */
    union
    {
        char c[4];
        unsigned int i;
    } map; /* Mapping between 4 chars and an integer */
    int nb_remain; /* Number of bytes not yet written */
    int nb_printed; /* Number of numbers written on the line */
} s_tocdata;

```

6 Summary

It is now possible to work with very different notions of 'file' in Mosel models and the Mosel libraries. With only minimal changes to his models the user may switch between data sources of different formats. The interaction and exchange of data between a model and the application executing it can be made more immediate and as a consequence, more efficient. It is also possible to avoid the creation of physical intermediate files by performing all operations in memory. The latter may be useful, for instance, in distributed applications.