Reference manual

FICO® Xpress Optimization

XPRD: Mosel remote invocation library

Reference manual

Release 1.4

Last update June 2015

www.fico.com | Make every decision count™

XPRD

Deliverable Version: A

Last Revised: June 2015

Version 1.4

Contents

1 Introduction 1
1.1 Overview . . . 1
1.2 File managers . . . 2

2 Functions of the XPRD library 5
2.1 Contexts and event handling . . . 5
XPRDinit . . . 6
XPRDfinish . . . 7
XPRDqueueempty . . . 8
XPRDgetevent . . . 9
XPRDdropevent . . . 10
XPRDwaitevent . . . 11
XPRDabortwait . . . 12
2.2 Mosel instances management . . . 13
XPRDconnect . . . 14
XPRDdisconnect . . . 15
XPRDconnected . . . 16
XPRDgetxprd . . . 17
XPRDbanner . . . 18
XPRDinstid . . . 19
XPRDsysinfo . . . 20
XPRDsetdefstream . . . 21
XPRDcompmod, XPRDcompmodsec . . . 22
2.3 Model management . . . 24
XPRDloadmod, XPRDloadmodsec . . . 25
XPRDgetmosel . . . 26
XPRDresetmod . . . 27
XPRDrunmod . . . 28
XPRDstoprunmod . . . 29
XPRDgetstatus . . . 30
XPRDgetdata . . . 31
XPRDgetexitcode . . . 32
XPRDgetnumber . . . 33
XPRDgetrmtid . . . 34
XPRDunloadmod . . . 35
XPRDsendevent . . . 36
XPRDsetdata . . . 37
2.4 Remote file access . . . 38
XPRDfflush . . . 39
XPRDfopen . . . 40
XPRDfclose . . . 41
XPRDfread . . . 42
XPRDfskip . . . 43
XPRDfwrite . . . 44
2.5 Connection manager . . . 45

XPRDstart 46
XPRDshutdown 47
XPRDsetmsglev 48
XPRDsetmsgcb 49
2.6 Miscellaneous 50
XPRDsetkeepalive 51
XPRDgetkeepalive 52
XPRDsetfsrvopt 53
XPRDgetfsrvopt 54
XPRDfindxsrvs 55
XPRDsetsshcmd 56
XPRDgetsshcmd 57

Appendix **58**

A Contacting FICO **58**
Product support 58
Product education 58
Product documentation 58
Sales and maintenance 59
Related services 59
About FICO 59

Index **60**

CHAPTER 1

Introduction

The *Mosel remote invocation library (XPRD)* makes it possible to build applications requiring the Xpress technology that run from environments where Xpress is not installed—including architectures for which Xpress is not available. Relying on the *Mosel Distributed Framework* (see Mosel module mmjobs), this self-contained library (*i.e.* with no dependency on the usual Xpress libraries) provides the necessary routines to start Mosel instances either on the local machine or on remote hosts and control them in a similar way as if they were invoked through the Mosel libraries. In particular, the published functionality includes

- redirection of standard streams (input, output and errors);
- compiling and loading of models;
- running and interrupting models.

In addition to these standard operations, the library supports the file handling mechanisms of *mmjobs* (transparent file access between instances) as well as its event signaling system (events can be exchanged between the application and running models).

1.1 Overview

Thanks to the *Mosel Distributed Framework* a Mosel model can start Mosel instances and use them to compile and run other models. The XPRD package implements the protocol used by the Mosel Distributed Framework such that an application using this library can perform the same general operations as a model using the *mmjobs* module: connect a new instance, compile models, load and run bim files, as well as access remote files and exchange events with running models. The following example shows the typical structure of a program using XPRD (for the sake of clarity error handling is not included):

```
{
 XPRDcontext xprd;
 XPRDmosel mosel;
 XPRDmodel model;

 /* Create an XPRD context */
 xprd=XPRDinit();

 /* Start an instance on host 'xpserver' */
 mosel=XPRDconnect(xprd, "xpserver", NULL, NULL, NULL, 0);

 /* Compile model from local source - bim file saved on remote instance */
 XPRDcompmod(mosel, "", "rmt:mymod.mos", "mymod.bim", "");

 /* Load bim file */
 model=XPRDloadmod(mosel, "mymod.bim");
```

```
  /* Run model */
  XPRDrunmod(model, "");

  /* Wait for termination of model before finishing */
  XPRDwaitevent(xprd,-1);
  printf("status: %d exit code:%d\n",
         XPRDgetstatus(model), XPRDgetexitcode(model));
  XPRDunloadmod(model);
  XPRDdisconnect(mosel);
  XPRDfinish(xprd);
}
```

The obvious use of XPRD is when Xpress is not installed on the host running the application: in this case one (or several) remote Mosel instance(s) can be launched on host(s) supporting Xpress. This is a requirement if the application is running on an architecture for which Xpress is not available but may also be useful if the application is executed on a machine with insufficient computational resources. In this scenario, the execution of models may be transfered to dedicated servers or even to some cloud computing facility.

XPRD can also be helpful when the models are to be run on the same host as the application calling the models. In this case, the program could of course use directly the usual Mosel libraries for its optimisation tasks and run models from the same process as the application itself. However, in certain cases it might be preferable to run the optimisation tasks in a separate process in order to preserve the application when resources required by the solution process cannot be predicted or if the models to be run are not coming from a trusted source. For example, an application using XPRD can start Mosel from a process with a limited amount of memory or CPU.

The XPRD package has no dependency on any external library and the Java version is written in pure Java (as opposed to the Mosel Java libraries that rely on native calls): as a consequence, an application using XPRD does not require any supplementary installation task and can be written in pure Java.

1.2 File managers

XPRD acts as a *master model*, and therefore has to process file operations requested from its remote instances (*i.e.* when a model opens a file using the "rmt:" driver). By default, the library handles file requests using the standard operating system routines looking for files from the process' current working directory. For example, if a remote instance asks for the file "rmt:myfile.txt", the library will look for "myfile.txt" in the current directory. In addition to accessing physical files, the I/O driver "sysfd:" is also supported by the library: typically a remote instance uses "rmt:sysfd:1" for its default output and "rmt:sysfd:2" for its default error stream. These streams are automatically routed to the corresponding local file descriptors such that output from remote instances is sent to the usual streams on the calling process.

At the time of creating a Mosel instance using function XPRDconnect it is possible to specify a *file manager* in order to complement or replace the default file handling mechanism. The entry point for this user-provided manager is a function of the following type:

```
void *fmgr(void *fctx, char *fname, int mode,
   XPRDfct_data* sync, XPRDfct_close *close, XPRDfct_skip *skip,
   char *errmsg, int msgsize);
```

This function is called instead of the default file manager whenever a request for opening a file is received from the corresponding instance. The first argument is the data pointer provided when creating the instance; fname is the file to open and mode its opening mode (*e.g.* XPRD_F_INPUT for reading). If this routine returns NULL, the file request is processed using the default procedure as described above. If the value XPRD_FMGR_ERR is returned, the request is rejected and an error message (NUL terminated) may be copied into buffer `errmsg` of size `msgsize`. Any other value is interpreted as a file descriptor pointer to be used with the provided I/O routines `sync`, `skip` and `close`.

The user functions `sync`, `skip` and `close` are used to transfer data from/to the local file and release the resources used by the file descriptor when the file is closed. Only the first function is mandatory (*i.e.* the others can be set to `NULL`). The signature of these routines is as follows:

```
int sync(void *fd, int buf, int bufsize);
int skip(void *fd,int nbtoskip);
int close(void *fd);
```

When the file is open for reading, the function `sync` is expected to copy into buffer `buf` up to `bufsize` bytes of data. The return value should be the number of bytes copied (0 indicating an end of file) or a negative value to report an error condition.
In the case of writing to the file, this function has to get `bufsize` bytes from buffer `buf`. The return value should be `bufsize` if writing is successful, any other value is interpreted as an I/O error.
The optional routine `skip` may be used to skip a number of bytes from a file open for reading (the function is not used on an output stream). Its return value must be positive or `0` in case of success; the special value `-2` indicates the operation is not supported (in which case bytes to skip are read using the `sync` routine) and any other value is interpreted as an I/O error.

In the following example, the file manager `my_open` redirects the pseudo file `"outremote"` to the function `outremote` (that simply displays the text it receives) and keeps the default behaviour for files open for reading and through `"sysfd:"` (redirection to standard streams). Any other queries are rejected.

```
void* XPRD_RTC my_open(void *ctx, char *filename,
    int mode, XPRDfct_data* fct_data, XPRDfct_close* fct_close,
    char *msg, int msglen)
{
 if(strcmp(filename,"outremote")==0)
 {
  if((mode&(XPRD_F_READ|XPRD_F_WRITE))!=XPRD_F_WRITE)
  {
   strncpy(msg, "'outremote' is write only!", msglen);
   return XPRD_FMGR_ERR;
  }
  else
  {
   *fct_data=outremote;
   *fct_close=NULL;
   return (void*)1;
  }
 }
 else
  if((strncmp(filename,"sysfd:",6)==0)||
     ((mode&(XPRD_F_READ|XPRD_F_WRITE))==XPRD_F_READ))
   return NULL;
  else
  {
   strncpy(msg, "access denied", msglen);
   return XPRD_FMGR_ERR;
  }
}

int XPRD_RTC outremote(void *data, char *buf, int size)
{
 printf("REMOTE: %.*s", size, buf);
 return size;
}
```

The above manager has to be passed to XPRD at the time of creating a new instance. In the example below, the error stream of the instance is redirected to the pseudo file `"outremote"`:

```
mosel=XPRDconnect(xprd, "", my_open, NULL, msg, sizeof(msg));
XPRDsetdefstream(mosel, NULL, XPRD_F_ERROR, "rmt:outremote");
```

CHAPTER 2

Functions of the XPRD library

2.1 Contexts and event handling

Each Mosel instance is attached to an *XPRD context*. This structure is also used to handle the queue of events received from the models run on the associated Mosel instances.

XPRDabortwait	Release a thread suspended by a call to XPRDwaitevent.	p. 12
XPRDdropevent	Drop the next event from the queue.	p. 10
XPRDfinish	Release an XPRD context.	p. 7
XPRDgetevent	Retrieve the next event from the queue.	p. 9
XPRDinit	Create a new XPRD context.	p. 6
XPRDqueueempty	Check whether the event queue is empty.	p. 8
XPRDwaitevent	Suspend the execution of the calling thread until an event is available.	p. 11

XPRDinit

Purpose

Create a new XPRD context.

Synopsis

```
XPRDcontext XPRDinit();
```

Return value

The new context or `NULL` in case of error.

Further information

Each context created using this function must be released by a call to `XPRDfinish`.

Related topics

`XPRDfinish`.

XPRDfinish

Purpose

Release an XPRD context.

Synopsis

```
void XPRDfinish(XPRDcontext ctx);
```

Argument

`ctx`	Context to be released

Further information

This routine releases all resources used by the given context: all active connections are closed and the event queue is freed. A context can no longer be used after it has been passed to this function.

Related topics

`XPRDinit`.

XPRDqueueempty

Purpose

Check whether the event queue is empty.

Synopsis

```
int XPRDqueueempty(XPRDcontext ctx);
```

Argument

`ctx`	XPRD context

Return value

1 if the queue is empty, 0 otherwise.

Related topics

`XPRDgetevent`, `XPRDdropevent`.

XPRDgetevent

Purpose

Retrieve the next event from the queue.

Synopsis

```
int XPRDgetevent(XPRDcontext ctx, XPRDmodel *sender, int *cls, double *value);
```

Arguments

`ctx`	XPRD context
`sender`	Pointer to return a reference to the sender of the event
`cls`	Pointer to return the class of the event
`value`	Pointer to return the value of the event

Return value

0 if successful, 1 if the queue is empty.

Related topics

`XPRDqueueempty`, `XPRDdropevent`, `XPRDwaitevent`, `XPRDsendevent`.

XPRDdropevent

Purpose

Drop the next event from the queue.

Synopsis

```
void XPRDdropevent(XPRDcontext ctx);
```

Argument

`ctx`	XPRD context

Further information

This routine has no effect if the queue is empty.

Related topics

`XPRDqueueempty`, `XPRDgetevent`.

XPRDwaitevent

Purpose

Suspend the execution of the calling thread until an event is available.

Synopsis

```
int XPRDwaitevent(XPRDcontext ctx, int timeout);
```

Arguments

`ctx`	XPRD context
`timeout`	maximum wait time (in seconds). A value smaller than 1 will cause an infinite wait

Return value

1 if the time limit has been reached, 0 otherwise.

Further information

If the event queue is not empty this routine returns immediately. This function can be interrupted by a call to `XPRDabortwait` (from a separate thread). In this case the function returns 0 even if the queue is empty.

Related topics

`XPRDqueueempty`, `XPRDabortwait`.

XPRDabortwait

Purpose

Release a thread suspended by a call to `XPRDwaitevent`.

Synopsis

```
void XPRDabortwait(XPRDcontext ctx);
```

Argument

`ctx`	XPRD context

Further information

This routine has no effect if no thread is waiting for the specified queue.

Related topics

`XPRDqueueempty`, `XPRDwaitevent`.

2.2 Mosel instances management

The `XPRDconnect` function starts a *Mosel instance* and returns a `XPRDmosel` object. The method used to create this instance depends on a *connection string* that is interpreted in a similar way as with the `connect` function of the *mmjobs* Mosel module: three I/O drivers can be used to launch a Mosel instance. The first one, `"rcmd:"`, executes a command in a separate process—typically this will be directly `mosel` or a special command to start Mosel on a remote host (*e.g.* the command `ssh`). The second driver, `"xsrv:"`, requires the `xprmsrv` Mosel remote launcher to run on the target machine: the connection is established with such a server through a TCP link. The last driver, `"xssh:"`, is similar to the previous one but establishes the connection to the server through a secure SSH tunnel. The handling of the tunnel is achieved by a separate process: by default, the `xprmsrv` program is used but optionally another SSH client may be selected using `XPRDsetsshcmd`.

`XPRDbanner`	Get the connection banner of a Mosel instance.	p. 18
`XPRDcompmod, XPRDcompmodsec`	Compile a model source file.	p. 22
`XPRDconnect`	Create a new Mosel instance.	p. 14
`XPRDconnected`	Check whether a Mosel instance is still connected.	p. 16
`XPRDdisconnect`	Release a Mosel instance.	p. 15
`XPRDgetxprd`	Get the XPRD context associated to a Mosel instance.	p. 17
`XPRDinstid`	Get the ID of a Mosel instance.	p. 19
`XPRDsetdefstream`	Set default input/output streams.	p. 21
`XPRDsysinfo`	Get system information about the host running a Mosel instance.	p. 20

XPRDconnect

Purpose

Create a new Mosel instance.

Synopsis

```
XPRDmosel XPRDconnect(XPRDcontext ctx, const char *cnstr, XPRDfct_open fmgr, void
       *fctx, char *errmsg, int msglen);
```

Arguments

`ctx`	XPRD context
`cnstr`	Connection string
`fmgr`	File manager routine (can be `NULL`)
`fctx`	Data pointer to be passed to the `fmgr` routine
`errmsg`	Buffer to return error messages
`msglen`	Size of `errmsg`

Return value

A Mosel instance or `NULL` in case of failure.

Example

In the following example 4 Mosel instances are started: `m1` is started in a separate process on the same host; `m2` is launched using the `xprmsrv` server running on host `"myserver"`; `m3` is executed using the `ssh` command on host `"secure"`, and for `m4`, the `xprmsrv` server running on host `"mybox"` is requested to use the context `"xpress"` with password `"mypass"`:

```
m1=XPRDconnect(xdctx,"",NULL,NULL,buf1,256);
m2=XPRDconnect(xdctx,"myserver",NULL,NULL,buf2,256);
m3=XPRDconnect(xdctx,"rcmd:ssh secure mosel -r",NULL,NULL,buf3,256);
m4=XPRDconnect(xdctx,"xsrv:mybox/xpress/mypass",NULL,NULL,buf4,256);
```

Further information

1. An empty connection string `""` is equivalent to `"rcmd:"` (instance started on the same machine in a separate process). Any other string not starting by either `"rcmd:"`, `"xsrv:"` or `"xssh:"` is interpreted as a host name that is prefixed by `"xsrv:"` (instance started on the specified host using the `xprmsrv` protocol). Refer to the Mosel Language Reference Manual, section on module *mmjobs* for further detail on how to use these drivers.

2. In case of failure, the parameter `errmsg` receives the error message reported by the driver used to perform the connection.

3. The optional file manager `fmgr` allows to control file access from the created remote instance: all requests for opening files are passed to this routine. Depending on the return value of the function, the request is rejected, processed by a user provided function or handled by the default file manager (*i.e.* direct access to physical files). Refer to Section 1.2 for further explanation.

4. Function `XPRDstart` is automatically called after a successful connection in order to start (if necessary) the *connection manager*.

5. Default streams of the newly created instance are initialised with `"null:"` for the input (*i.e.* stream disabled); `"rmt:sysfd:1"` for output and `"rmt:sysfd:2"` for errors. These settings can be changed using `XPRDsetdefstream`.

Related topics

`XPRDsetsshcmd`, `XPRDdisconnect`, `XPRDconnected`.

XPRDdisconnect

Purpose

Release a Mosel instance.

Synopsis

```
void XPRDdisconnect(XPRDmosel mosel);
```

Argument

`mosel`	Mosel instance

Further information

1. All models are unloaded (running models are first stopped) before closing the connection and releasing the resources used by the instance.
2. Function `XPRDshutdown` is automatically called during the disconnection procedure.
3. An `XPRDmosel` object can no longer be used after it has been disconnected.

Related topics

`XPRDconnect`, `XPRDconnected`.

XPRDconnected

Purpose

Check whether a Mosel instance is still connected.

Synopsis

```
int XPRDconnected(XPRDmosel mosel);
```

Argument

`mosel`	Mosel instance

Return value

1 if the instance is connected, 0 otherwise.

Further information

1. The connection to a remote instance may be lost due to a network failure or because the corresponding process has terminated: this routine allows to check for this situation.
2. A call to `XPRDdisconnect` is still required to release the local resources used by an instance even if this function reports the instance is no longer active.

Related topics

`XPRDconnect`, `XPRDdisconnect`.

XPRDgetxprd

Purpose

Get the XPRD context associated to a Mosel instance.

Synopsis

```
XPRDcontext XPRDgetxprd(XPRDmosel mosel);
```

Argument

`mosel`	Mosel instance

Return value

The XPRD context to which the instance is associated.

Related topics

`XPRDsysinfo`, `XPRDbanner`.

XPRDbanner

Purpose

Get the connection banner of a Mosel instance.

Synopsis

```
const char* XPRDbanner(XPRDmosel mosel);
```

Argument

`mosel`	Mosel instance

Return value

Message displayed by the instance upon connection.

Related topics

`XPRDsysinfo`, `XPRDgetxprd`.

XPRDinstid

Purpose

Get the ID of a Mosel instance.

Synopsis

```
int XPRDinstid(XPRDmosel mosel);
```

Argument

`mosel`	Mosel instance

Return value

Node number associated to the instance.

XPRDsysinfo

Purpose

Get system information about the host running a Mosel instance.

Synopsis

```
char* XPRDsysinfo(XPRDmosel mosel,int what, char *buf,size_t buflen);
```

Arguments

`mosel`	Mosel instance
`what`	What information to collect:
	`XPRD_SYS_NAME` Name of the operating system
	`XPRD_SYS_VER` Version name of the operating system
	`XPRD_SYS_REL` Release number of the operating system
	`XPRD_SYS_PROC` Processor type
	`XPRD_SYS_ARCH` Processor architecture (32 or 64 bit)
	`XPRD_SYS_NODE` Computer name
`buf`	Buffer to store the information
`buflen`	Size of `buf`

Return value

A reference to `buf` or `NULL` in case of error.

Further information

Several information items can be obtained in a single call by summing up the option codes. In such a case, the resulting string consists in the different items separated by commas. All available information can be retrieved using `XPRD_SYS_ALL`.

Related topics

`XPRDbanner`, `XPRDgetxprd`.

XPRDsetdefstream

Purpose

Set default input/output streams.

Synopsis

```
int XPRDsetdefstream(XPRSmodel mosel, XPRDmodel model, int wmd, const char *filename);
```

Arguments

`mosel`	Reference to a Mosel instance or `NULL`
`model`	Reference to a model or `NULL`
`wmd`	Stream to set. Possible values: `XPRD_F_READ` Default input stream `XPRD_F_WRITE` Default output stream `XPRD_F_ERROR` Default error stream `XPRD_F_LINBUF` Use line buffering
`filename`	Extended file name to be used for the stream.

Return value

0 if successful, 1 otherwise.

Further information

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

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

3. Using an empty string as the file name implies resetting to the original default stream: `"null"` for input; `"rmt:sysfd:1"` for output and `"rmt:sysfd:2` for error.

XPRDcompmod, XPRDcompmodsec

Purpose

Compile a model source file.

Synopsis

```
int XPRDcompmod(XPRDmosel mosel, const char *options, const char *srcfile, const char
        *dstfile, const char *userc);
int XPRDcompmodsec(XPRDmosel mosel, const char *options, const char *srcfile, const
        char *dstfile, const char *userc, const char *passfile, const char *privkey,
        const char *kfile);
```

Arguments

`mosel`	Mosel instance
`options`	Compilation options (may be `NULL`). Possible values:
	`"g"` Include debugging information: in the case of a run time error during the execution of the model the location of the error in the source file may be indicated
	`"G"` Include tracing information: with this option the model can be run through the debugger for an execution step by step
	`"s"` Strip symbols: secure the bim file by removing all private symbol names used in the source model
	`"p"` Parse only: stop after the syntax analysis of the source file, do not compile (no file generated)
	`"bx=prefix"` Package prefix (can be quoted with single or double quotes)
	`"ix=prefix"` Include source prefix (can be quoted with single or double quotes)
	`"S"` Sign the bim file
	`"E"` Encrypt the bim file
	`"F"` The argument `pass` is a file name (not the password itself)
	`"V"` Accept to load signed packages only if their signature can be verified
	`"T"` Accept to load only signed packages with a valid signature
`scrfile`	Name of the source file
`dstfile`	Name of the destination file (may be `NULL`)
`userc`	Commentary text that will be saved as is at the beginning of the output file (may be `NULL`)
`passfile`	Password or password file (for encryption with a password)
`privkey`	Private key file (for bim file signing)
`kfile`	File of public keys (for encryption with public keys)

Return value

Execution status:

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

Example

Ask the Mosel instance `minst` to compile the model `"mymod.mos"` stored locally. The resulting bim file `"mymod.bim"` is saved on the host running this instance:

```
m=XPRDcompmod(minst, "", "rmt:mymod.mos", "mymod.bim", "");
```

Further information

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

2. The source file name may contain environment variable references using the notation `${varname}` (for example,
‘`${XPRESSDIR}/examples/mymodel`’) that are expanded to generate the actual name.

3. When sending a compilation request to a separate Mosel instance, it is important to keep in mind that the operation is performed in the environment of this instance (in particular its current working directory) and file names should be specified appropriately (the `rmt:` I/O driver can be particularly helpful in this context).

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

Related topics

`XPRDloadmod`, `XPRDrunmod`.

2.3 Model management

A *model object* is created by loading a bim file onto a Mosel instance with a call to `XPRDloadmod`. Once a model has been loaded, it can be run (`XPRDrunmod`), send events (`XPRDsendevent`) and possibly be interrupted before its normal termination (`XPRDstoprunmod`). Additional functions provide information about the last execution. Models must be *unloaded* using `XPRDunloadmod` in order to release the resources they use both on the local host and the remote instance.

`XPRDgetdata`	Return the data pointer of a model.	p. 31
`XPRDgetexitcode`	Return the exit code of a model after its execution.	p. 32
`XPRDgetmosel`	Get a reference to the Mosel instance on which a model is loaded.	p. 26
`XPRDgetnumber`	Return the model number.	p. 33
`XPRDgetrmtid`	Return the ID of the model on the remote instance.	p. 34
`XPRDgetstatus`	Return the current status of a model.	p. 30
`XPRDloadmod, XPRDloadmodsec`	Load a Binary Model file onto the specified instance.	p. 25
`XPRDresetmod`	Reset a model.	p. 27
`XPRDrunmod`	Run a model.	p. 28
`XPRDsendevent`	Send an event to a running model.	p. 36
`XPRDsetdata`	Define the data pointer of a model.	p. 37
`XPRDstoprunmod`	Stop a running model.	p. 29
`XPRDunloadmod`	Unload a model.	p. 35

XPRDloadmod, XPRDloadmodsec

Purpose

Load a Binary Model file onto the specified instance.

Synopsis

```
XPRDmodel XPRDloadmod(XPRDmosel mosel, const char *bname);
XPRDmodel XPRDloadmodsec(XPRDmosel mosel, const char *bname, const char *flags, const
       char *passfile, const char *privkey, const char *keys);
```

Arguments

`mosel`	Mosel instance
`bname`	Name of a binary model file
`flags`	Loading options: `"c"` Check signature (if the file is signed) `"V"` If the file is signed, load it only if the signature is valid `"T"` Load only signed files with a valid signature `"F"` The argument `passfile` is a file name (not the password itself)
`passfile`	Password or password file (for encrypted bim files)
`privkey`	Private key file (for encrypted bim files)
`keys`	File of public keys

Return value

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

Example

Load model `"myfile.bim"` stored locally onto the `minst` remote instance:

```
m=XPRDloadmod(minst, "rmt:mymod.bim");
```

Further information

1. This function returns the reference of a new model instance created from a binary model file. The second form of the function will be used to load encrypted and/or signed bim files if additional information has to be provided. While loading a model from a file, Mosel also automatically opens any additional modules that are required by this model.

2. It is important to keep in mind that the operation is performed in the environment of a remote instance (in particular its current working directory) and file names should be specified appropriately (the `rmt:` I/O driver can be particularly helpful in this context).

3. Default streams of the newly created model are inherited from the Mosel instance `mosel`. These settings can be changed using `XPRDsetdefstream`.

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

Related topics

`XPRDrunmod`, `XPRDunloadmod`.

XPRDgetmosel

Purpose

Get a reference to the Mosel instance on which a model is loaded.

Synopsis

```
XPRDmosel XPRDgetmosel(XPRDmodel model);
```

Argument

`mosel`	Mosel instance

Return value

The Mosel instance on which the model is loaded.

XPRDresetmod

Purpose

Reset a model.

Synopsis

```
void XPRDresetmod(XPRDmodel model);
```

Argument

`model`	Reference to a model

Further information

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

Related topics

`XPRDrunmod`, `XPRDunloadmod`.

XPRDrunmod

Purpose

Run a model.

Synopsis

```
int XPRDrunmod(XPRDmodel model, const char *parlist);
```

Arguments

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

Return value

0 if successful, a positive value if the execution cannot be started.

Further information

1. This procedure starts the execution of a model on its Mosel instance: when the procedure returns, the model is not necessarily started (this may be delayed depending on the operating system load) and not necessarily terminated (the second model is executing concurrently to the caller).
2. When the execution of the model is completed (normal termination, interruption after calling `XPRDstoprunmod`, or runtime error) or could not be started, an event of class `XPRD_EVENT_END` is sent to the caller. The execution status is returned via the event value and it can also be obtained using `XPRDgetstatus`. The exit code related to the last execution may be retrieved using `XPRDgetexitcode`.
3. If the same model has to be executed several times concurrently, it must be loaded several times in different model objects.
4. The parameter `parlist` may be used to initialize the model parameters of the model/program (*e.g.* `"PAR1=12,PAR2='tutu'"`).

Related topics

`XPRDloadmod`.

XPRDstoprunmod

Purpose

Stop a running model.

Synopsis

```
void XPRDstoprunmod(XPRMmodel model);
```

Argument

`model`	Model to interrupt

Further information

If the model is not currently running, no operation is performed. Note that the effect of this call may not be immediate and the corresponding model may continue running a few seconds before its effective interruption (for instance, the time required to complete an I/O operation).

Related topics

`XPRDrunmod`.

XPRDgetstatus

Purpose

Return the current status of a model.

Synopsis

```
int XPRDgetstatus(XPRDmodel model);
```

Argument

`model`	Reference to a model

Return value

Model status. Possible values are:

`XPRD_RT_OK`	Normal termination
`XPRD_RT_ERROR`	An error occured during execution
`XPRD_RT_MATHERR`	Mathematical error (*e.g.* division by zero)
`XPRD_RT_I/OERR`	Input/output error (*e.g.* cannot open file)
`XPRD_RT_STOP`	Bit set if execution has been interrupted
`XPRD_RT_FDCLOSED`	Connection to the remote host has been lost
`XPRD_RT_RUNNING`	Model currently running

Further information

When the status is `XPRD_RT_FDCLOSED`, the model is no longer usable and the only possible operation is `XPRDunload` or `XPRDdisconnect` that must be called in order to release local resources used by the model.

XPRDgetdata

Purpose

Return the data pointer of a model.

Synopsis

```
void *XPRDgetdata(XPRDmodel model);
```

Argument

`model`	Reference to a model

Return value

Model data pointer.

Further information

This function returns the data pointer previously set using `XPRDsetdata`.

XPRDgetexitcode

Purpose

Return the exit code of a model after its execution.

Synopsis

```
int XPRDgetexitcode(XPRDmodel model);
```

Argument

`model`	Reference to a model

Return value

Execution status as stated by the Mosel procedure `exit`.

XPRDgetnumber

Purpose

Return the model number.

Synopsis

```
int XPRDgetnumber(XPRDmodel model);
```

Argument

`model`	Reference to a model

Return value

Model order number.

Related topics

`XPRDgetdata`.

XPRDgetrmtid

Purpose

Return the ID of the model on the remote instance.

Synopsis

```
int XPRDgetrmtid(XPRDmodel model);
```

Argument

`model`	Reference to a model

Return value

Model number as returned by the Mosel control parameter `modelnumber`.

XPRDunloadmod

Purpose

Unload a model.

Synopsis

```
int XPRDunloadmod(XPRDmodel model);
```

Argument

`model`	Reference to a model

Return value

0 if successful, 1 otherwise.

Further information

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

Related topics

`XPRDloadmod`.

XPRDsendevent

Purpose

Send an event to a running model.

Synopsis

```
int XPRDsendevent(XPRDmodel model, int class, double value);
```

Arguments

`model`	Model to send the event to
`class`	Event class (must be >1)
`value`	Event value

Further information

1. An event can be received only by a running model that is using the *mmjobs* module: sending an event to a model that is not running or not using *mmjobs* is a no-operation.
2. Events are characterized by a `class` and a `value`. Event class values can be used to indicate the cause of the event (for instance, 2 could mean 'a new solution has been found') and the associated value may specify a property of the given instance (for example an objective value). Except for the special value 1 (`XPRD_EVENT_END`) class values have no predefined meaning.
3. An event of class `XPRD_EVENT_END` (=1) with the model status as the associated event value is automatically sent by each model to its parent when its execution terminates.

Related topics

`XPRDwaitevent`, `XPRDgetevent`.

XPRDsetdata

Purpose

Define the data pointer of a model.

Synopsis

```
void XPRDsetdata(XPRDmodel model, void *data);
```

Arguments

`model`	Reference to a model
`data`	User defined data pointer

Further information

The provided reference is stored in the model structure and can be retrieved at a later stage using `XPRDgetdata`. The data pointer is not used by XPRD and can be employed by the host application for recording model specific information.

2.4 Remote file access

These basic file operation routines allow an application to open a file for reading or writing on a remote host through a connected Mosel instance.

`XPRDfclose`	Close a file that was previously opened with XPRDfopen.	p. 41
`XPRDfflush`	Flush buffer of an output stream.	p. 39
`XPRDfopen`	Open a file on a remote instance.	p. 40
`XPRDfread`	Read a block of data from a remote file.	p. 42
`XPRDfskip`	Skip a block of data from a remote file.	p. 43
`XPRDfwrite`	Write a block of data to a remote file.	p. 44

XPRDfflush

Purpose

Flush buffer of an output stream.

Synopsis

```
int XPRDfflush(XPRDfile f);
```

Argument

`f`	File descriptor

Return value

0 if successful.

Further information

The output buffer is automatically flushed when the file descriptor is closed.

Related topics

`XPRDfwrite`.

XPRDfopen

Purpose

Open a file on a remote instance.

Synopsis

```
XPRDfile XPRDfopen(XPRDmosel mosel, const char *fname, int mode, char *errmsg, int
       msglen);
```

Arguments

`mosel`	Mosel instance
`fname`	File name
`mode`	Open mode (may be combined):

`XPRD_F_BINARY`	Open file in binary mode (default is text mode)
`XPRD_F_INPUT`	Open for reading
`XPRD_F_OUTPUT`	Empty the file and open it for writing
`XPRD_F_APPEND`	Open for writing, appending new data to the end of the file
`XPRD_F_LINBUF`	If open for writing, flushes buffer after end of each line

`errmsg`	Buffer to return error message
`msglen`	Size of `errmsg`

Return value

A file descriptor or `NULL` in case of failure.

Example

Open file `"myfile"` located in the temporary directory of instance `minst` for reading in binary mode:

```
f=XPRDfopen(minst, "tmp:myfile", XPRD_F_BINARY|XPRD_F_INPUT);
```

Further information

1. The specified file path is relative to the working directory of the Mosel instance performing the file operation.
2. File operations are performed under the restrictions of the Mosel instance. For example, if the remote instance does not have write access, this routine will fail to open a file for writing.
3. Just like accessing files from a Mosel model, any I/O drivers supported by the remote instance can be used with this routine. Drivers `"sysfd:"`, `"tmp:"` and `"shmem:"` are therefore available.

Related topics

`XPRDfread`, `XPRDfskip`, `XPRDfwrite`, `XPRDfclose`.

XPRDfclose

Purpose

Close a file that was previously opened with `XPRDfopen`.

Synopsis

```
int XPRDfclose(XPRDfile f);
```

Argument

`f`	File descriptor

Return value

0 if successful, a positive value otherwise.

Further information

Once closed a file descriptor can no longer be used even if the function returns an error code.

Related topics

`XPRDfopen`.

XPRDfread

Purpose

Read a block of data from a remote file.

Synopsis

```
long XPRDfread(XPRDfile f,void *buf, long size);
```

Arguments

`f`	File descriptor
`buf`	Buffer to return the data
`size`	Size of buffer `buf`

Return value

0 in case of end of file; the number of bytes read or a negative value in case of error.

Further information

The amount of data read may be smaller than the amount requested: this is not an error.

Related topics

`XPRDfopen`, `XPRDfskip`.

XPRDfskip

Purpose

Skip a block of data from a remote file.

Synopsis

```
int XPRDfskip(XPRDfile f,int size);
```

Arguments

`f`	File descriptor
`size`	Number of bytes to skip

Return value

Negative values indicate an error.

Related topics

`XPRDfopen`, `XPRDfread`.

XPRDfwrite

Purpose

Write a block of data to a remote file.

Synopsis

```
long XPRDfwrite(XPRDfile f,const void *buf, long size);
```

Arguments

`f`	File descriptor
`buf`	Data to be written
`size`	Size of buffer

Return value

The number of bytes written or a negative value in case of error.

Further information

Output streams are buffered: use `XPRDfflush` to force actual writing of the data currently stored in the buffer.

Related topics

`XPRDfopen`.

2.5 Connection manager

As soon as the first connection is established the *connection manager* is started and it is shut down when all connections have been closed. This manager consists in a background thread handling the communication protocol required by the Mosel Distributed Framework. The functions of this section can be used to control this manager: start and stop independently of active connections as well as handling of messages.

`XPRDsetmsgcb`	Set the message callback.	p. 49
`XPRDsetmsglev`	Change the verbosity level of the library.	p. 48
`XPRDshutdown`	Shut down the connection manager.	p. 47
`XPRDstart`	Start the connection manager.	p. 46

XPRDstart

Purpose

Start the connection manager.

Synopsis

```
int XPRDstart();
```

Return value

0 if successful, a positive value otherwise.

Further information

1. This routine allows the user to start the *connection manager* independently of the active connections as to keep it running if several connections are performed sequentially.
2. This routine is automatically called by `XPRDconnect` after a connection succeeds.
3. The system keeps track of the number of times this routine has been called and the function `XPRDshutdown` must be called the same number of times in order to actually shut down the manager.

Related topics

`XPRDshutdown`.

XPRDshutdown

Purpose

Shut down the connection manager.

Synopsis

```
void XPRDshutdown();
```

Further information

1. Calling this routine shuts down the *connection manager* previously started by a call to `XPRDstart`.
2. This routine is automatically called by `XPRDdisconnect` after the connection has been closed.

Related topics

`XPRDstart`.

XPRDsetmsglev

Purpose

Change the verbosity level of the library.

Synopsis

```
void XPRDsetmsglev(int lev);
```

Argument

`lev`	New verbosity level

Further information

The default verbosity level is 1 (report only error messages). For debugging purpose this routine might be used to display more information.

Related topics

`XPRDsetmsgcb`.

XPRDsetmsgcb

Purpose

Set the message callback.

Synopsis

```
void XPRDsetmsgcb(void *ctx, long (*cbmsg)(void*,void *,char *,unsigned long));
```

Arguments

`ctx`	Context to be passed to the callback routine
`cbmsg`	Message callback. The first argument is always `NULL`; the second corresponds to `ctx`, the two final ones are the message buffer and its length

Further information

By default, messages produced by the library are sent to the default error stream.

Related topics

`XPRDsetmsglev`.

2.6 Miscellaneous

`XPRDfindxsrvs`	Search xprmsrv servers on the local network.	p. 55
`XPRDgetfsrvopt`	Get configuration settings for XPRDfindxsrvs.	p. 54
`XPRDgetkeepalive`	Get KeepAlive settings.	p. 52
`XPRDgetsshcmd`	Get the command used for SSH connections.	p. 57
`XPRDsetfsrvopt`	Set configuration settings for XPRDfindxsrvs.	p. 53
`XPRDsetkeepalive`	Set KeepAlive settings.	p. 51
`XPRDsetsshcmd`	Set the command to use for SSH connections.	p. 56

XPRDsetkeepalive

Purpose

Set KeepAlive settings.

Synopsis

```
int XPRDsetkeepalive(XPRDcontext ctx,int maxfail,int inter);
```

Arguments

`ctx`	XPRD context
`maxfail`	Maximum number of failures before the link is considered broken (≥ 1; default value:2)
`interval`	Interval (in seconds) between two activity checks (≥ 4; default value:60)

Return value

0 if successful, 1 otherwise.

Further information

1. In order to verify if the connection between a client and a server is still active, a *keep alive* message is sent from the server to the client every `interval` seconds. A server will consider the link is down (and close the connection) if no reply has been received after `maxfail+1` *keepalive* messages. Similarly, a client will close the connection to a server that has not sent any message for more than `interval*(maxfail+1)` seconds.
2. Using value `0` for `maxfail` disables the *keepalive* mechanism.
3. This routine can only be called before any connection is created.

Related topics

`XPRDgetkeepalive`.

XPRDgetkeepalive

Purpose

Get KeepAlive settings.

Synopsis

```
void XPRDgetkeepalive(XPRDcontext ctx,int *maxfail,int *inter);
```

Arguments

`ctx`	XPRD context or `NULL` to get default initial values
`maxfail`	Buffer to return the maximum number of failures before the link is considered broken
`interval`	Buffer to return the interval (in seconds) between two activity checks

Further information

Value 0 is returned for both `maxfail` and `interval` when the *keepalive* mechanism is disabled.

Related topics

`XPRDsetkeepalive`.

XPRDsetfsrvopt

Purpose

Set configuration settings for `XPRDfindxsrvs`.

Synopsis

```
void XPRDsetfsrvopt(XPRDcontext ctx,unsigned short port,int nbiter,int delay);
```

Arguments

`ctx`	XPRD context
`port`	UDP port number
`nbiter`	Number of iterations
`delay`	Maximum wait time (in milliseconds)

Further information

The `XPRDfindxsrvs` function uses these parameters as follows: a broadcast message is sent to UDP port `port` up to `nbiter` times. For each of these iterations, a maximum of `delay` milliseconds is waited for answers from remote servers.

Related topics

`XPRDgetfsrvopt`, `XPRDfindxsrvs`.

XPRDgetfsrvopt

Purpose

Get configuration settings for `XPRDfindxsrvs`.

Synopsis

```
void XPRDgetfsrvopt(XPRDcontext ctx,unsigned short *port,int *nbiter,int *delay);
```

Arguments

`ctx`	XPRD context or `NULL` to get default settings
`port`	Buffer to return UDP port number or `NULL`
`nbiter`	Buffer to return the number of iterations or `NULL`
`delay`	Buffer to return the maximum wait time (in milliseconds) or `NULL`

Further information

Default values are returned if the context `ctx` is `NULL`.

Related topics

`XPRDfindxsrvs`, `XPRDsetfsrvopt`.

XPRDfindxsrvs

Purpose

Search xprmsrv servers on the local network.

Synopsis

```
int XPRDfindxsrvs(XPRDcontext ctx,int grp,int maxip,unsigned int *addrs);
```

Arguments

`ctx`	XPRD context or `NULL` to use default settings
`grp`	Group number of the request
`maxip`	Maximum number of addresses to collect (*i.e.* size of `addrs`)
`addrs`	Buffer to return the IP addresses

Return value

The number of IPs stored in `addrs` or `-1` in case of error.

Example

The following example uses this function to find a server and displays its IP address if one is found:

```
struct in_addr addr;
if(XPRDfindxsrvs(NULL,1,1,(unsigned int *)&addr)==1)
 printf("Server found at %s\n",inet_ntoa(addr));
```

Further information

1. This function sends a broadcast message over the local network and waits for replies from running `xprmsrv` servers. A given server will reply only to selected *group* numbers: the `grp` argument specifies this property.
2. The IP addresses of the hosts having replied to the request are returned via the last argument of the procedure in the form of unsigned integers (to be cast as a `struct in_addr` for socket functions). The maximum number of IPs is fixed by `maxip` that cannot be larger than the size of the provided buffer.

Related topics

`XPRDgetfsrvopt`, `XPRDsetfsrvopt`.

XPRDsetsshcmd

Purpose

Set the command to use for SSH connections.

Synopsis

```
int XPRDsetsshcmd(XPRDcontext ctx,const char *sshcmd);
```

Arguments

`ctx`	XPRD context
`sshcmd`	Command starting an SSH client connection

Return value

0 if successful, 1 otherwise.

Further information

This routine specifies which command to use for opening an SSH connection to a remote host as required by the `"xssh:"` I/O driver. The provided string may contain the following special symbols that are replaced before the process is started:

`%h`	the target host
`%p`	port to connect to
`%f`	known host file (it is replaced by `"-"` when no file is provided)

The default value for the parameter is `"xprmsrv -sshclt %h -p %p -kh %f"`

Related topics

`XPRDconnect`, `XPRDgetsshcmd`.

XPRDgetsshcmd

Purpose

Get the command used for SSH connections.

Synopsis

```
const char *XPRDgetsshcmd(XPRDcontext ctx);
```

Argument

`ctx`	XPRD context or `NULL` to get default settings

Related topics

`XPRDsetsshcmd`.

APPENDIX A

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information on the Product Support home page (www.fico.com/support).

On the Product Support home page, you can also register for credentials to log on to FICO Online Support, our web-based support tool to access Product Support 24x7 from anywhere in the world. Using FICO Online Support, you can enter cases online, track them through resolution, find articles in the FICO Knowledge Base, and query known issues.

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

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Sales and maintenance

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

About FICO

FICO (NYSE:FICO) delivers superior predictive analytics solutions that drive smarter decisions. The company's groundbreaking use of mathematics to predict consumer behavior has transformed entire industries and revolutionized the way risk is managed and products are marketed. FICO's innovative solutions include the FICO® Score—the standard measure of consumer credit risk in the United States—along with industry-leading solutions for managing credit accounts, identifying and minimizing the impact of fraud, and customizing consumer offers with pinpoint accuracy. Most of the world's top banks, as well as leading insurers, retailers, pharmaceutical companies, and government agencies, rely on FICO solutions to accelerate growth, control risk, boost profits, and meet regulatory and competitive demands. FICO also helps millions of individuals manage their personal credit health through www.myfico.com. Learn more at www.fico.com. FICO: Make every decision count^{TM}.

Index

B
bim, 23
binary
    model file, 23

C
cloud computing, 2
comment
    user, 22
compile
    model, 22
connection banner, 18
connection manager, 45
    shut down, 47
    start, 46
connection string, 13
context, 5

D
debugging, 22
default streams, 21

E
error stream, 21
event
    abort wait, 12
    drop next, 10
    get next, 9
    send, 36
    wait for, 11
event queue, 8
execute
    model, 28

F
file handling, 2
file manager, 2

I
input stream, 21

K
keepalive, 51, 52

L
load
    model, 25

M
master model, 2
message callback, 49
message level, 48
model
    compile, 22
    data pointer, 31, 37
    exit code, 32
    get Mosel instance, 26
    load, 25
    number, 33
    reset, 27
    run, 28
    status, 30
    stop, 29
    unload, 35
model file
    binary, 23
model object, 24
model parameters, 28
Mosel Distributed Framework, 1
Mosel instance, 13
    check connection, 16
    connection banner, 18
    create, 14
    get, 26
    get context, 17
    ID, 19
    release, 15
    system info, 20

O
output stream, 21

R
remote file
    close, 41
    flush, 39
    open, 40
    read, 42
    skip, 43
    write, 44
remote file access, 38
resetting model, 27
run
    model, 28

S
source file, 22
stream
    set, 21
strip symbols, 22
symbol
    strip, 22

T
tracing, 22

U
unload
    model, 35
user comment, 22

X
XPRD context, 5
    create, 6
    release, 7
XPRD_F_APPEND, 40
XPRD_F_BINARY, 40
XPRD_F_ERROR, 21
XPRD_F_INPUT, 40
XPRD_F_LINBUF, 21, 40
XPRD_F_OUTPUT, 40
XPRD_F_READ, 21
XPRD_F_WRITE, 21
XPRD_RT_ERROR, 30
XPRD_RT_FDCLOSED, 30
XPRD_RT_I/OERR, 30
XPRD_RT_MATHERR, 30
XPRD_RT_OK, 30
XPRD_RT_RUNNING, 30
XPRD_RT_STOP, 30
XPRD_SYS_ARCH, 20
XPRD_SYS_NAME, 20
XPRD_SYS_NODE, 20
XPRD_SYS_PROC, 20
XPRD_SYS_REL, 20
XPRD_SYS_VER, 20
XPRDabortwait, 12
XPRDbanner, 18
XPRDcompmod, 22
XPRDcompmodsec, 22
XPRDconnect, 14
XPRDconnected, 16
XPRDdisconnect, 15
XPRDdropevent, 10
XPRD_EVENT_END, 28
XPRDfclose, 41
XPRDfflush, 39
XPRDfindxsrvs, 55
XPRDfinish, 7
XPRDfopen, 40
XPRDfread, 42
XPRDfskip, 43
XPRDfwrite, 44
XPRDgetdata, 31
XPRDgetevent, 9
XPRDgetexitcode, 32
XPRDgetfsrvopt, 54
XPRDgetkeepalive, 52
XPRDgetmosel, 26
XPRDgetnumber, 33
XPRDgetrmtid, 34
XPRDgetsshcmd, 57
XPRDgetstatus, 30
XPRDgetxprd, 17
XPRDinit, 6
XPRDinstid, 19
XPRDloadmod, 25
XPRDloadmodsec, 25
XPRDmosel, 13
XPRDqueueempty, 8
XPRDresetmod, 27
XPRDrunmod, 28
XPRDsendevent, 36
XPRDsetdata, 37
XPRDsetdefstream, 21
XPRDsetfsrvopt, 53
XPRDsetkeepalive, 51
XPRDsetmsgcb, 49
XPRDsetmsglev, 48
XPRDsetsshcmd, 56
XPRDshutdown, 47
XPRDstart, 46
XPRDstoprunmod, 29
XPRDsysinfo, 20
XPRDunloadmod, 35
XPRDwaitevent, 11