Emdros Programmer's Reference Guide (Preview) – Version 1.2.0.pre231

Ulrik Petersen

December 1, 2006

Contents

1 Introduction

Welcome to the Emdros Programmer's Reference Guide. This guide is designed to help you get up to speed quickly with how to program an application on top of Emdros. Here are some of the topics covered:

- Emdros architecture and how your application fits in.
- API for the MQL language.
- The DTD of the XML output by the MQL engine.
- Tips for how to architect your application.
- Tips for loading an Emdros database with existing data.
- Tips for creating a query-application.

Plus additional goodies.

1.1 Appliccable version

This guide documents Emdros version **1.2.0.pre226** and above, in the 1.2.0.pre-series.

1.2 How to use this guide

Click on any item in the menu on the left. This will expand it and go to that page.

2 Part I: Foundations

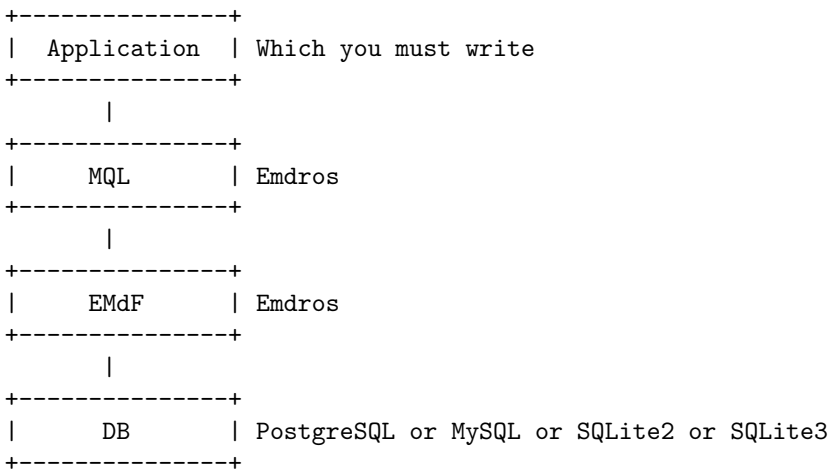
In Part I, we lay the foundation for the rest of the programmer's reference guide. We will look at the following things:

- Emdros architecture and how your application fits in

- Your application and what it minimally needs to include in order to interact with Emdros.
- Libraries and how to use them.

2.1 Emdros architecture

Emdros fits into an architecture as follows:



At the top, we have your application, which in turn rests on Emdros. Emdros is made up of two layers, first the MQL layer, which then rests upon the EMdF layer. The EMdF layer, in turn, interacts with the underlying RDBMS.

This Programmer's Reference Guide is about making you successful in building your application.

2.2 Your application

In this section, we describe two things:

1. Four ways to call Emdros, and
2. The details of how to do so.

2.2.1 Ways of calling Emdros

There are a number of ways you can go about writing your application. Here are four of them:

1. Write your application in C++, interacting directly with the Emdros libraries.
2. Write your application in one of the scripting languages which Emdros supports through SWIG (currently Python, Perl, PHP, Ruby, and Java).

3. Interface via the mql(1) program as a command-line program, or via a system() call or the like.

4. Hook up the mql(1) program to a TCP/IP port using xinetd(8) or inetd(8), interacting with it as if it were a daemon.

We will look at each of these in turn.

Via C++

Benefits Interfacing with Emdros via C++ offers the following benefits:

- Greatest performance:
  - No need to parse the output of the MQL engine. You can use the datastructures returned by MQL directly instead of first dumping them to stdout, then reading them back into memory again. This saves both space and time.
  - No need to start and stop the mql(1) process. Everything is done in-process.
  - Persistence of back-end connection: No need to connect and reconnect to the back-end. This saves time.
- Greatest flexibility: You can use the full EMdF API, rather than "just" the MQL API.

Drawbacks Drawbacks include:

- Greater complexity, due to:
  - Complexity of C++ language,
  - Complexity of Emdros APIs.
- Steeper learning-curve (due to greater complexity).

Via SWIG

Overview SWIG is the "Simplified Wrapper Generator", and allows a programmer to wrap his C or C++ application's interface so that it can be used from a number of scripting languages.

Emdros implements SWIG interfaces for a number of languages. They include:

- Ruby
- Java
- Python
- Perl

- PHP

This means that you can use Emdros from these scripting languages as though you were writing in C++ with almost no performance overhead. The libraries are linked in through dynamic library loading.

Granted, Java is not a scripting language, but is mentioned as such here for simplicity of presentation.

Note: currently, **only Python**-bindings and **Java**-bindings are **available on Windows**. All are available on Linux. It is unknown whether any will work on Solaris.

Benefits Benefits include:

- Having the simplicity and rapid prototyping of scripting languages.
- Losing none of the power of direct access to the Emdros class hierarchy.
- Not having to parse the output from MQL, instead directly accessing the MQLResult object. This reduces performance overhead.

Drawbacks Drawbacks include:

- Slightly slower performance than with pure C++.

Via mql(1) as a command

Overview You can use the mql(1) program for interfacing with Emdros. In C, this is typically done with the system() library-call. Scripting-languages such as Ruby, Perl, and Python have their own ways of calling external programs.

Benefits Benefits include:

- Simplicity: The only APIs to learn are:
  - The MQL language itself (which you must learn anyway),
  - The format of the MQL output,
  - The exit-status codes of the mql(1) program.

Drawbacks Drawbacks include:

- Less performance:
  - Output must be parsed and the datastructures recreated. This takes both space and time.
  - mql(1) process must be started and stopped for each collection of MQL statements you wish to have evaluated. This takes time.
  - Back-end connection is established and broken together with the mql(1) process. This takes time.

- Less flexibility: You can only interface with Emdros via the MQL language, not the EMdF API.

That having been said, it may be possible to keep the mql(1) process running, keeping an open pipe between your application and the mql(1) process. If so, the only performance-problem that remains is the parsing of the output.

Via mql(1) as a daemon

Overview Setting up the mql(1) program as a daemon has much the same benefits and drawbacks as using it as a command.

Benefits Additional benefits include:

- Persistence of connection: You can keep the connection (and thus the mql(1) program) running for as long as you like, thus eliminating the performance-problems relating to starting and stopping the mql(1) process and the back-end connection.
- It may be simpler to connect via a TCP/IP socket than via an external program if you use a scripting language.
- You can connect remotely, and from architectures that do not have the mql(1) program.

Drawbacks Drawbacks include:

- There may be security-related problems with having an mql daemon running on your system.
- Emdros is fundamentally a single-user system. If more than one mql(1) process is spawned at a time, your mileage may vary.

  However, the single-user status only applies to MQL statements that create or update or delete data, not to querying. If your database is in a phase in which it is not created/updated/deleted, but only queried, then you should be able to run mql(1) as a daemon with no problems.

2.2.2 How to call Emdros

Having described the benefits and drawbacks of each method of calling Emdros, let us describe how to actually do it.

Via mql(1) In this section, we will describe how to interface to Emdros via the mql(1) program. This includes both using mql(1) as a command and interfacing to it via xinetd(8)/inetd(8).

Method of interaction

As a command When calling mql(1) as a command, you can either pass the input on a file or on stdin. The output is given on stdout.

You can capture the output by redirecting it to a file (e.g., append "¿ /tmp/x13wd93pq" to the command line). Alternatively, set up a pipe between your application and the mql(1) program. See your system manual pages for how to do this.

See the mql(1) man-page for details of command-line options and command-line syntax.

As a daemon When connecting as a daemon, you must open a TCP/IP stream connecting to the host and port on which you have configured xinetd/inetd to listen for MQL input. Issue your command(s) by writing to the stream and listen on the TCP/IP stream for the answer. Use the QUIT statement to close down the connection.

Output formats

Two choices You have two choices for output, specified by two command-line switches:

- **XML output**, which is best for interaction between programs, and
- **Console output**, which is slightly more human-readable than XML output.

XML output

Benefits Benefits include:

- Has better support for error-handling: It tells you explicitly when something when wrong, in what stage of the MQL compiler, the source of the error (compiler or back-end), and any error message.
- Tells you exactly when a command is being processed and when it is finished.
- Shows the progress of query-queries.

Drawbacks Drawbacks include:

- More verbose, and may take longer to parse.
- More complex in its syntax and more complex to parse.

Later, we will expound the DTD of the XML output.

Console output

Benefits Benefits include:

- Simpler in its syntax. Can be parsed even with AWK.
- Less verbose, which may constitute a performance gain on really slow hardware.

Drawbacks Drawbacks include:

- Less robust error-handling. Is meant for human reading, not for parsing by machines.
- Does not tell you when a statement has finished or when it is being processed.
- Does not have a progress-indicator for query-queries.

Exit status codes The mql(1) program has the following exit codes:

```
 * 0 : Success
 * 1 : Wrong usage
 * 2 : Connection to backend server could not be established
 * 3 : An exception occurred
 * 4 : Could not open file
 * 5 : Database error
 * 6 : Compiler error
```

You can probably only access these if you run mql(1) as a command, not via xinetd/inetd.

xinetd/inetd setup

The basic idea The basic idea of xinetd/inetd is that a super-server (xinetd or inetd) is run, which is configured to listen on certain ports. When an incoming request arrives on a given port, the super-server spawns a background process (in this case, the mql(1) program) and connects that program's stdin/stdout to the TCIP/IP stream.

Configuring xinetd/inetd This means that you must configure xinetd/inetd to spawn the mql(1) program with the correct command-line switches, including whether you want XML or console output.

Included with Emdros, you will find a sample xinetd(8) configuration file, called "xinetd.mql". Please refer to the xinetd(8) documentation for how to set this up.

Next Next, we describe how to interface with Emdros using C++.

Via C++

Easy procedure As of Emdros version 1.1.11, the easiest way of interfacing with Emdros is to use the EmdrosEnv API. This sets up all the machinery described in the next section.

Full procedure The following instructions are **not necessary to follow**. You can simply use the EmdrosEnv API. If you want to know how to do without it, the following instructions tell you that.

1. Create an EMdF database object. This must not be an EMdFDB object, but rather one of the descendants matching the back-end you are running:
   - PgEMdFDB for a PostgreSQL back-end,
   - MySQLEMdFDB for a MySQL back-end.
   - SQLiteEMdFDB for an SQLite 2 back-end.
   - SQLite3EMdFDB for an SQLite 3 back-end.
2. Check that the connection is OK with the connectionOk() method. If not, back up and implement an error-recovery strategy of your choice.
3. Create a EMdFOutput object. This is for specifying:
   - The default encoding (e.g., ISO-8859-1),
   - The std::ostream descendant on which output should be printed (e.g., std::cout), and
   - The output kind (XML or Console).
4. Create a MQLExecEnv from the EMdF database and the EMdFOutput object.
5. Use the executing environment to interact with the MQL layer, and the EMdF database object to interact with the EMdF layer.
6. When you are done, delete the execution environment and the EMdF database object.

Next Next, we will see a code-snippet from the mql(1) program which shows how to use the EmdrosEnv API.

Example The following are some snippets from the mql(1) program. They are placed under the GPL license as given at the beginning of the file src/mql.cpp in the Emdros sourcecode package.

```
#include <emdfdb.h>
#include <emdros_environment.h>
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
```

```
// Not showing exec_file() function

int exec_cin(EmdrosEnv *pEE)
{
  bool bResult;
  std::string strError;
  int nResult;
  if (!pEE->executeStream(std::cin, bResult, true, true)) {
    std::cerr << "FAILURE: Database error executing stdin." << std::endl;
    nResult = 5;
  } else {
    if (!bResult) {
      // std::cerr << "FAILURE: Compiler error executing stdin." << std::endl;
      nResult = 6;
    } else {
      // std::cout << "SUCCESS executing stdin." << std::endl;
      nResult = 7;
    }
  }

  // Return result
  return nResult;
}

int main(int argc, char* argv[])
{
  // Set defaults
  eOutputKind output_kind = kOKConsole;
  std::string initial_db("emdf");
  std::string filename;
  std::string hostname("localhost");
  std::string user("emdf");
  eBackendKind backend\_kind = kSQLite2; // Could also be kPostgreSQL or kMySQL or kSQLite

  // Get default password
  std::string password;
#if USE[PREUNDERSCORE}DEFAULT_PASSWORD
  getDefaultPassword(password);
#else
  password = "";
#endif

  // Parse arguments
  if (!parse_arguments(argc, argv, output_kind,
                       initial_db, filename,
                       hostname, user, password)) {
    print_usage(std::cerr);
    return 1;
```

```
  }

  // Make connection
  EmdrosEnv *pEE = new EmdrosEnv(output_kind, kCSISO_8859_1,
   hostname,
   user, password,
   initial_db,
   backend_kind);
  // Zero-fill password
  zeroFillString(password);

  // Check that we connected
  if (!pEE->connectionOk()) {
    std::cerr << "Connection to backend server could not be established.  Aborting."
              << std::endl;
    delete pEE;
    return 2;
  }

  int nResult = 3;
  try {
    if (filename == "") {
      nResult = exec_cin(pEE);
    } else {
      nResult = exec_file(pEE, filename);
    }
  } catch (EMdFDBException e) {
    std::cerr << "ERROR: EMdFDBException (Database error)..." << std::endl;
    std::cerr << pEE->getDBError() << std::endl;
    std::cerr << pEE->getCompilerError() << std::endl;
  } catch (BadMonadsException e) {
    std::cerr << "BadMonadsException caught.  Program aborted." << std::endl;
  } catch (MQLSymbolTableEntryException e) {
    std::cerr << "MQLSymbolTableEntryException caught.  Program aborted." << std::endl;
  } catch (MQLSymbolTableException e) {
    std::cerr << "MQLSymbolTableException caught.  Program aborted." << std::endl;
  } catch (WrongCharacterSetException e) {
    std::cerr << "WrongCharacterSetException caught.  Program aborted." << std::endl;
  } catch (EMdFOutputException e) {
    std::cerr << "EMdFOutputException caught.  Program aborted." << std::endl;
  } catch (...) {
    std::cerr << "Unknown exception occurred.  Program aborted." << std::endl;
  }

  // Clean up
  delete pEE;
  return nResult;
}
```

Via SWIG

Overview This method was described previously.

Examples and HOW-TOs For **code examples and HOW-TOs**, please see the sources, especially:

- The READMEs in the SWIG directory and the SWIG/*language* directories (SWIG/java, SWIG/ruby, etc.),
- The testing program available for each scripting language in the appropriate SWIG/*language* directory.

Other than that, we refer to the description of C++ for HOW-TOs. The method is the same, even if the language is different.

2.3 Libraries

2.3.1 Introduction

In this section, you will find:

- Information on which libraries to use,
- Information on how to configure your Linux environment for Emdros shared libraries,
- A shared-library HOWTO by Kirk Lowery.

2.3.2 Which libraries to use

Introduction This page is split up into two sections:

- Unix/Linux,
- Windows.

Unix/Linux

Introduction This page gives instructions about which libraries to use on Unix/Linux, and how to use them. But first, some background.

Background On Unix, there are two kinds of libraries:

- static, and
- shared.

Static libraries normally end in ".a", while shared libraries normally end in ".so".

A library named "foo" will generally have a "lib" prefix, so it will be either:

- **libfoo.a** (if static), or
- **libfoo.so** (if shared)

How to link

What When linking your application, you will need to pass a number of switches to the linker. How this is done depends on your compiler and/or linker.

Generally speaking, the **-l** switch tells the linker to link in a certain library. So **-lfoo** links in the **foo** library.

Most linkers will choose a shared library over a static one by default, so -lfoo will link in libfoo.so if it is present.

Where In addition, the linker will need to know where to search for the libraries. Generally speaking, this is done with the **-L** switch. So

```
-L/usr/local/lib/emdros
```

will tell the linker to look in the /usr/local/lib/emdros directory.

LDFLAGS The conventional way of doing this in Makefiles is to set the **LDFLAGS** variable to the string of switches that is necessary.

Order matters The order in which you specify these options usually matters. Generally speaking, -L switches should come before -l switches, and dependent libraries should come before libraries on which they are dependent.

Which libraries to use

Always You will **always** need the following library:

```
-lemdf
```

PostgreSQL If using **PostgreSQL**, you will also need the following library:

```
-lpq
```

libpq is a PostgreSQL library.

MySQL If using **MySQL**, you will also need the following library:

```
-lmysqlclient
```

In addition, you will probably need to tell the linker where to find the mysqlclient library. It is probably located in /usr/lib/mysql:

```
-L/usr/lib/mysql -lmysqlclient
```

In addition, you may find that libmysqlclient depends on libz. So you may need to do the following:

```
-L/usr/lib/mysql -lmysqlclient -lz
```

libmysqlclient is a MySQL library.

SQLite 2 If using **SQLite 2**, you will also need the following library:

```
-lsqlite_emdros
```

This library is found in the same directory as the other Emdros libraries, so there should be no need for extra -L flags.

SQLite 3 If using **SQLite 3**, you will also need the following library:

```
-lsqlite3_emdros
```

This library is found in the same directory as the other Emdros libraries, so there should be no need for extra -L flags.

MQL If using **MQL**, you will need the follwoing libraries:

```
-lmql -lpcre_emdros
```

libmql depends on libemdf and libpcre_emdros.

Summary: PostgreSQL If using **PostgreSQL**, you will need a string such as the following:

```
-L/usr/local/lib/emdros -lmql -lpcre_emdros -lemdf -lpq
```

Summary: MySQL If using **MySQL**, you will need a string such as the following:

```
-L/usr/local/lib/emdros -L/usr/lib/mysql -lmql -lpcre_emdros -lemdf -lmysqlclient
```

Summary: SQLite 2 If using **SQLite 2**, you will need a string such as the following:

```
-L/usr/local/lib/emdros -lmql -lpcre_emdros -lemdf -lsqlite_emdros
```

Summary: SQLite 3 If using **SQLite 3**, you will need a string such as the following:

```
-L/usr/local/lib/emdros -lmql -lpcre_emdros -lemdf -lsqlite3_emdros
```

Windows

Introduction This page describes which libraries to use on Windows.

Look in the win32.mak makefiles in the sourcecode distribution in order to get a feel for how to use them. Also consult the Visual Studio documentation for how to add libraries to your project.

Which libraries to use

Always You will **always** need the following library:

- libemdf.lib

PostgreSQL If using **PostgreSQL**, you will also need the following libraries:

- libpq.lib
- libpq.dll

libpq.lib and libpq.dll are PostgreSQL libraries, and are distributed with the Emdros PostgreSQL binaries.

If your version of PostgreSQL does not match the version from which these libraries came in the binary distribution, follow the instructions in INSTALL.Win32 in the source distribution in order to build your own.

libemdf.lib depends on libpq.lib.

MySQL If using **MySQL**, you will also need the following library:

- mysqlclient.lib

In addition, you may need access to the MySQL header-files.
mysqlclient.lib is a MySQL library.
libemdf.lib depends on mysqlclient.lib.

SQLite 2 If using **SQLite 2**, you will also need the following library:

- sqlite_emdros.lib

This is compiled automatically when you specify in the config.mak file that you want SQLite 2 as a backend.

SQLite 3 If using **SQLite 3**, you will also need the following library:

- sqlite3_emdros.lib

This is compiled automatically when you specify in the config.mak file that you want SQLite 3 as a backend.

MQL If using **MQL**, you will need the following libraries:

- libmql.lib
- libpcre_emdros.lib

libmql.lib depends on libemdf.lib and libpcre_emdros.lib.

Summary: PostgreSQL If using **PostgreSQL**, you will need the following libraries:

- libemdf.lib
- libpq.lib
- libpq.dll
- libmql.lib
- libpcre_emdros.lib

Summary: MySQL If using **MySQL**, you will need the following libraries:

- libemdf.lib
- mysqlclient.lib
- libmql.lib
- libpcre_emdros.lib

Summary: SQLite 2 If using **SQLite 2**, you will need the following libraries:

- libemdf.lib
- libsqlite_emdros.lib
- libmql.lib
- libpcre_emdros.lib

Summary: SQLite 3 If using **SQLite 3**, you will need the following libraries:

- libemdf.lib
- libsqlite3_emdros.lib
- libmql.lib
- libpcre_emdros.lib

2.3.3 How to configure Linux for Emdros use

Introduction This page describes how to set up your linker on Linux so that it finds the Emdros shared libraries.

ld.so The ld.so(8) linker is the system component responsible for linking in shared libraries. By default, it looks in a number of standard directories for shared libraries. There are two ways to configure this:

- Edit the /etc/ld.so.conf file by adding the directories that should be searched, or
- Set the LD_LIBRARY_PATH environment variable to point to the directories that should be searched in addition to the ones in /etc/ldd.conf.

Setting LD_LIBRARY_PATH In order to set the LD_LIBRARY_PATH variable, you can do the following (in the bash shell):

```
export LD_LIBRARY_PATH="\$LD_LIBRARY_PATH:/usr/local/lib/emdros"
```

2.3.4 How to build shared libraries

Attribution The following was contributed by Prof. Dr. Kirk E. Lowery, kloweryatwhi.wtsPARIS.edu. (remove French city).

Shared Library Fundamentals Libraries first came about when complex programs organized function calls in logical groupings. With the advent of GUIs and object-oriented programming, functions began to be designed for use by many programs, and so needed grouping conveniently together into one file of object modules produced by the compiler.

These libraries are called "static" libraries and, on Linux systems, usually carry the "a" extension, e.g., libemdf.a. A programmer wishing to use the functions in a particular library uses the syntax of the function call documented in the header file, e.g., monads.h, called the "Application Program Interface (API)". At link time, the linker makes a copy of the desired functions binary code, and includes it in the final executable binary of the application program. Such a program is called "statically" linked, and is independent of the presence of the original static library. The program can be used on any compatible system whether or not the static library is present, since the program carries all needed code within itself.

As operating systems became more complex, and especially with the appearance of client-server technology, hard disks and RAM became cluttered with many copies of the same functions. Application program binaries were increasing in size by whole orders of magnitude. In order to improve effeciency the concept of "shared" libraries was created.

The idea of a shared library is simple: code that is used by more than one application needs to reside only *once* on the hard disk and in memory. And it does not need to be loaded into memory until called for. It can even be removed from memory if deemed necessary.

For an application such as Emdros, the necessity for and advantages of shared libraries are obvious. Emdros is intended to work with databases, standing in between a client application and the database server itself. Such an environment is inherently multi-tasking and multi-user. The more basic of emdros' function calls (e.g., the creation and manipulation of monads, or mql queries) will be used thousands of time during the session of one user. We don't want thousands of copies of dozens of functions filling up memory, and once the overhead of calling a function and copying it into memory is paid, we don't want to have to pay it over and over again.

Using and Linking with Shared Libraries Let's use the example of Emdros' own application program that uses the Emdros libraries: mql (or, under Windows: mql.exe). Here is a code snippet from mql.cpp where an emdros library function is used:

```
// Make EMdFOutput
EMdFOutput *pOut = new EMdFOutput(kCSISO_8859_1, &std::cout, output_kind);
```

A new instance of the object EMdFOutput is created here and given the name pOut. Accompanying this object are a set of methods (or, functions) which can be used with this object. We don't need to know the details of how EMdFOutput does its magic. All we need to know is found in emdf_output.h:

```
class EMdFOutput {
protected:
  eCharsets m_charset;
  eOutputKind m_output_kind;
  std::ostream *m_pStream;
  int m_current_indent;
  int m_indent_chars;
public:
  EMdFOutput(eCharsets charset, std::ostream *pStream,
             eOutputKind output_kind, int indent_chars = 3);
  ~EMdFOutput();
  // Getting
  eOutputKind getOutputKind(void) const { return m_output_kind; }
  // Output
  void increaseIndent();
  void decreaseIndent();
  void out(std::string s);
  void outCharData(std::string s);
  void newline();
  void flush() { *m_pStream << std::flush; };
  // XML members
  void printXMLDecl();
  void printDTDstart(std::string root_element);
  void printDTDend();
  void startTag(std::string name, bool newline_before = false);
  // Must have pairs of (attribute name, attribute value)
  void startTag(std::string name, const AttributePairList&
                attributes, bool newline_before = false);
  // for tags of type <tag/>
  void startSingleTag(std::string name, bool newline_before = false);
  // for tags of type <tag/>
  void startSingleTag(std::string name, const AttributePairList&
                attributes, bool newline_before = false);
  void endTag(std::string name, bool newline_before = false);
protected:
  void emitAttributes(const AttributePairList& attributes);
};
```

This API tells us all we need in order to create instances of objects of this class, what information we need to give to it, what kind of information we can expect from it, and what functions we can use to manipulate it.

In order to compile the mql application program we issue the command:

```
g++ -g -o .libs/mql mql.o  -L/usr/local/src/emdros/EMdF \
                           -L/usr/local/src/emdros/MQL \
                           -L/usr/local/src/emdros/pcre \
          /usr/local/src/emdros/MQL/.libs/libmql.so -lpcre_emdros \
          /usr/local/src/emdros/EMdF/.libs/libemdf.so \
```

```
-lpq -Wl,--rpath -Wl,/usr/local/lib/emdros
```

The compiler is told the name of the object output file (mql.o), where to place it in the source tree before installation (.libs/mql), and to include debugging information (-g). The "-L" option tells g++ where to find libraries to link to, and the "-l" option tells it which libraries it is to link against, looking for a "libpcre_emdros.a" in the case of "-lpcre_emdros". This is correct, since the pcre library is not compiled as shared, but are linked statically into library functions that need it. In the case of the other libraries, the ".so" extention (Shared Objects) tells the linker that we want to link against these "dynamically," not "statically". "pq" stands for the "postgresql" libraries, which is the database backend chosen for this installation of emdros. "–rpath" is the "runtime path" that will be used to find the shared libraries as the program is executing.

If you are using the GNU Autobuild tools (autoconf, automake, libtool) – and if you are not, you should be! – then the following lines in your Makefile.am will generate the above command, assuming you have properly created the top-level configure.in file:

```
bin_PROGRAMS = mql
mql_LDADD = -L../EMdF -L../MQL -L../pcre -lmql @EMDFLDADD@
mql_DEPENDENCIES = @EMDFDEPS@ ../pcre/libpcre_emdros.a ../MQL/libmql.la
mql_SOURCES = mql.cpp
INCLUDES = -I../include
CLEANFILES = *~ core .deps/*
AM_CXXFLAGS = @CXXFLAGS@ @DEBUGFLAG@
```

Note how libmql is to be linked as a shared library so it is listed as "libmql.la" whereas libpcre_emdros is listed as "libpcre_emdros.a" for static linking.

It is beyond the scope of this HOWTO to get into all the details of the Autobuild tools, but the reader is referred not just to the excellent documentation that comes along with the software, but also to the highly recommended tutorial "The AutoBook"

<http://sources.redhat.com/autobook/>

Making Your Own Shared Library Creating a shared library is pretty straightforward. But I am not going to talk about how to do it manually without using the Autobuild tools, particularly libtool. Unless your situation is trivially simple, libtool makes your life so much easier. I am going to use my experience in modifying the emdros distribution to build shared libraries in addition to the static ones. To make matters even more simple, I will concentrate on just one library: libemdf.

There are two files that concern us in the task. First is configure.in in the top-level emdros directory. The second is EMdF/Makefile.am. I will note the elements needed for building the libraries only.

In configure.in, we need the following:

```
dnl Library versioning
dnl We begin with 0:0:0
LIB_CURRENT=0
LIB_REVISION=0
LIB_AGE=0
AC_SUBST(LIB_CURRENT)
AC_SUBST(LIB_REVISION)
AC_SUBST(LIB_AGE)


dnl Invoke libtool
AC_PROG_LIBTOOL


dnl
dnl Set EMDFLDADD and EMDFDEPS
dnl
if test "x\$BACKEND" = "xpostgresql"; then
  EMDFLDADD="-lemdf -lpq";
  EMDFDEPS="../EMdF/libemdf.la";
else
  EMDFLDADD="-lemdf -lmysqlclient";
  EMDFDEPS="../EMdF/libemdf.la";
fi
```

Let's deal with the easy stuff first. In order to use libtool, we need to tell autoconf that it is going to be used, with the macro AC_PROG_LIBTOOL. We don't need AC_PROG_RANLIB used for static libraries, because libtool handles all that. Because we have a choice of database backends, emdros needs to be told which database it will be used with, and the appropriate emdros libraries to be linked in. These dependencies get propagated down to the lower level makefiles, such as the EMdF subdirectory.

Library Versioning Now a more complex but essential subject is library versioning. Because many libraries form the foundation of many other libraries and programs, e.g., libc, and because these libraries are in constant development and change, a protocol was created to allow multiple versions of the same shared library to exist on the same system, so that application programs (and, indeed, the kernel) can have the version of libraries against which they were compiled. So now the question is, when does one need more than one version of a library? The answer is, when the API changes. If the API does not change, then the application program doesn't care particularly if something "under the hood" changes.

Library versions track the *interface*, which is a set of three entry points into the library. These entry points are arranged in a hierarchy:

```
current interface:revision number:age number
```

The current interface documents a specific way the library functions are called. That means if there is any addition to the library functions, or changes in the way those functions are called, the data type of their parameters, etc., then the interface number of the library must change. If any revisions to the source code of the library is made by fixing bugs, improving performance, even adding functionality (e.g., more rigorous tests made of input data), *but* the prototype of the library functions has not changed, then this is a *revision* of the *current* interface, and the middle number is incremented. The runtime loader will always use the highest revision number of the current interface. Finally, the age number tells us how many previous interfaces are supersets of earlier interfaces, i.e., how many earlier interfaces can be linked by binaries. The age must always be less than or equal to the current interface number.

Quoting from AutoBook, here are the rules for incrementing these three numbers:

1. If you have changed any of the sources for this library, the revision number must be incremented. This is a new revision of the current interface.

2. If the interface has changed, then current must be incremented, and revision reset to '0'. This is the first revision of a new interface.

3. If the new interface is a superset of the previous interface (that is, if the previous interface has not been broken by the changes in this new release), then age must be incremented. This release is backwards compatible with the previous release.

4. If the new interface has removed elements with respect to the previous interface, then you have broken backward compatibility and age must be reset to '0'. This release has a new, but backwards incompatible interface.

Thus, in our example above, since this is the first interface for the shared libraries, it receives the number 0. There are no revisions for this new interface, so revision=0 and age must be 0. Here is a very important principle:

THE SOFTWARE RELEASE VERSION

AND THE SHARED LIBRARY VERSION NUMBERING SCHEMES

HAVE NOTHING TO DO WITH EACH OTHER!

Even though emdros is at Release 1.1.7 when shared libraries support was added, the numbering scheme of the libraries is 0:0:0 and is independent of the release numbers. The library versioning *must* conform to the four rules listed above. If other parts of emdros are changed, but not the libraries, then the library verions remain the same.

Finally, the AC_SUBST macro exports the values of the library versions for substitution in lower-level makefiles.

The Final Step The code for the libemdf library itself is found in EMdF/Makefile.am:

```
pkglib_LTLIBRARIES = libemdf.la
libemdf_la_SOURCES = conn.cpp \
       emdf_wstring.cpp \
       emdfdb.cpp \
       utils.cpp \
       inst.cpp \
       monads.cpp \
       infos.cpp \
       table.cpp \
       string_func.cpp \
       inst_object.cpp \
       emdf_output.cpp
libemdf_la_LDFLAGS = -version-info @LIB_CURRENT@:@LIB_REVISION@:@LIB_AGE@
```

First, we tell automake about our library: we want it installed in the "package" directory for libraries (/usr/local/lib/emdros in this case), that the following list of of libraries are to be made in both static and shared versions, using the "la" extension name. The next macro tells automake what source files are to be used for building libemdf.la. Finally, we tell automake about the version of the library.

That's it, believe it or not! Automake and libtool handle all the rest. Simply invoke the compile. For example:

```
aclocal && automake --add-missing && autoconf && ./configure && make install
```

Now do you see why we so strongly recommend the Autobuild tools? Yes, we thought so! :-)

3 Part II: APIs

Part II contains information on the various APIs necessary for interfacing with Emdros.

The EMdF API is not described in its fullness, since most of it is not strictly necessary for interfacing with Emdros. Interested readers may consult the source code.

3.1 EmdrosEnv

3.1.1 Overview

The EmdrosEnv class is an abstraction of the Emdros API. Use it in your applications as the main interface to Emdros.

3.1.2 C++ interface

```
#include <emdros_environment.h>

/** Backend kind
 *
 * Used to distinguish among backends.
 *
 */
enum eBackendKind {
kBackendNone = 0,/**< No backend selected */
kPostgreSQL = 1, /**< PostgreSQL */
kMySQL = 2,      /**< MySQL */
kSQLite2 = 3,    /**< SQLite 2.X.X */
kSQLite4 = 4     /**< SQLite 3.X.X */
};



class EmdrosEnv {
public:
  EmdrosEnv(std::ostream* output_stream,
             eOutputKind output_kind, eCharsets charset,
             std::string hostname,
             std::string user, std::string password,
             std::string initial_db,
             eBackendKind backend_kind = DEFAULT_BACKEND_ENUM);
  // The following constructor uses std::cout (Standard Out)
  EmdrosEnv(eOutputKind output_kind, eCharsets charset,
             std::string hostname,
             std::string user, std::string password,
             std::string initial_db,
             eBackendKind backend_kind = DEFAULT_BACKEND_ENUM);
  virtual ~EmdrosEnv();

  // Backend-name
  static std::string getBackendName(void);


  // Executing MQL queries
  // See the mqlExecute* functions
  // for information on the parameters.
  bool executeString(const std::string& input, bool& bResult,
                     bool bPrintResult, bool bReportError);
  bool executeFile(std::string filename, bool& bResult,
                   bool bPrintResult, bool bReportError);
  bool executeStream(std::istream& strin, bool& bResult,
                     bool bPrintResult, bool bReportError);
```

```
// Cleaning up
// clean() calls MQLExecEnv::clean().
// It is good to call if you are through with a query's results
// and there is a long time till the next executeXXX call.
void clean();

// Database support
// Call the corresponding EMdFDB methods
bool connectionOk(void);
bool vacuum(bool bAnalyze);
bool getMin_m(monad_m& /* out */ min_m);
bool getMax_m(monad_m& /* out */ max_m);
bool getAll_m_1(SetOfMonads& /* out */ all_m_1);

// Returns the string-representation of an enumeration constant in
// enum enum_name with the value value.
// Just calls the corresponding EMdFDB method.
// bDBOK is true on no DB error.
virtual std::string getEnumConstNameFromValue(long value,
                                              const std::string& enum_name,
                                              /* out */ bool \&bDBOK);


// Transactions
// Call the corresponding EMdFDB methods
bool beginTransaction(void);
bool commitTransaction(void);
bool abortTransaction(void);

//
// Check for results
//

// Check for sheaf
// Returns true on result is sheaf.
// Returns false on no result or result not sheaf
bool isSheaf(void);

// Check for table
// Returns true on result is table.
// Returns false on no result or result not table.
bool isTable(void);

// NOTE on statements:
// If only one MQL query was executed by the last executeXXX
// invocation, then there will be only one statement to get.
// If more than one MQL query was given to one of these methods,
// only the results from the last statement executed will be available.
```

```
// Getting results.  Next invocation of executeXXX deletes the object,
// so you cannot execute a query until you are done processing
// the result
MQLResult* getResult(void); // Return nil on no result
Sheaf* getSheaf(void); // Returns nil on no result or result not a sheaf
FlatSheaf* getFlatSheaf(void); // Returns nil on no result or result not a flat sheaf
Table* getTable(void); // Returns nil on no result or result not a table
Statement *getStatement(void); // Returns nil on no statement

// Take over object.  You now have responsibility for deleting it,
// and it will not be deleted by the next invocation of executeXXX.
MQLResult* takeOverResult(void); // Return nil on no result
Sheaf* takeOverSheaf(void); // Returns nil on no result or result not a sheaf
FlatSheaf* takeOverFlatSheaf(void); // Returns nil on no result or result not a flat she
Table* takeOverTable(void); // Returns nil on no result or result not a table
Statement *takeOverStatement(void); // Returns nil on no statement



// Getting error-messages and info
std::string getDBError(void);
std::string getCompilerError(void);
// Gets the compiler stage that was executed last
int getLastCompilerStage(void);
// clears local DB error message in EMdFDB and local error in MQLError
void clearErrors(void);

// Outputting
// These all output on the stream in local EMdFOutput,
// i.e., the stream you passed to the constructor of EmdrosEnv,
// or stdout if the other constructor was used
void out(std::string str);  // out your own string
void out(MQLResult *pResult);
void out(Sheaf *pSheaf);
void out(Table *pTable);

//
// XML outputting
//

// XML declaration.
// Calls EMdFOutput::printXMLDecl on the local EMdFOutput
void printXMLDecl();

// DTDs
void printDTDStart(std::string root_element);
void printDTDEnd();
void printDTDMQLResult(void);
void printDTDTable(void);
void printDTDSheaf(void);
```

```
  // Accessing members
  MQLExecEnv* getMQLEE(void);
};
```

3.2 EMdFOutput

3.2.1 Overview

The EMdFOutput class is an abstraction of three things:

- A default encoding (e.g., ISO-8859-1),
- A pointer to a std::ostream descendant to use when outputting, and
- An output kind, telling Emdros whether to use console output or XML output.

3.2.2 C++ interface

```
#include <emdf_output.h>

typedef enum {
  kCSASCII,
  kCSISO_8859_1,
  kCSISO_8859_8,
  kCSUTF8
} eCharsets;

typedef enum {
  kOKXML,
  kOKConsole
} eOutputKind;

class EMdFOutput {
public:
  EMdFOutput(eCharsets charset,       // Default encoding
             std::ostream *pStream,   // Output stream
             eOutputKind output_kind, // XML or Console?
             int indent_chars = 3     // No. of indent-spaces for XML
             );
  ~EMdFOutput();
  eOutputKind getOutputKind(void) const { return m_output_kind; }
};
```

3.3 EMdF database

3.3.1 Overview

The EMdF database classes are structured in the following hierarchy:

- EMdFDB
  - PgEMdFDB
  - MySQLEMdFDB
  - SQLiteEMdFDB (for SQLite 2)
  - SQLite3EMdFDB (for SQLite 3)

3.3.2 Boolean return value

For the functions which return a boolean, the return value generally means the following:

- true: No error.
- false: An error occurred. Use the error interface to get to know what the error was. The values of any reference parameters which should have returned something are undefined.

The only exceptions to this rule are the transaction interface methods and the ConnectionOk() method.

3.3.3 Special 'emdf' database name

All of the backends recognize the special database name 'emdf' (without the quotes). This is a valid database name, but will not associate the connection with any database.

Hence, you can use 'emdf' as a default database name used until you issue a 'USE DATABASE' MQL statement or call the EMdFDB::useDatabase() method.

On PostgreSQL, the user is actually connected to the 'template1' database.

On MySQL, the user is connected with no default associated database.

On SQLite (2 and 3), the user is not connected to any database.

3.3.4 C++ interface

emdfdb.h

```
#include <emdfdb.h>

class EMdFDBException {};
class EMdFDBDBError : public EMdFDBException {};

class EMdFDB {
public:
```

```
  //
  // Construction and destruction
  EMdFDB();
  virtual ~EMdFDB();

  // Get backend-name
  static std::string getBackendName(void);

  //
  // Database manipulation
  virtual bool useDatabase(const std::string& db_name);

  // min_m and max_m
  bool getMin_m(monad_m& /* out */ min_m);
  bool getMax_m(monad_m& /* out */ max_m);
  bool getAll_m_1(SetOfMonads& /* out */ all_m_1);

  // Indices
  bool createIndicesOnDatabase(const std::string& database_name);
  bool dropIndicesOnDatabase(const std::string& database_name);
  bool createIndicesOnObjectType(const std::string& object_type_name);
  bool dropIndicesOnObjectType(const std::string& object_type_name);
  bool createIndicesOnObjectType(const std::string& database_name,
                                 const std::string& object_type_name);
  bool dropIndicesOnObjectType(const std::string& database_name,
                               const std::string& object_type_name);


  // DB maintenance
  virtual bool vacuum(bool bAnalyze);

  // Transactions
  virtual bool beginTransaction(void);
  virtual bool commitTransaction(void);
  virtual bool abortTransaction(void);

  //
  // Error messages
  virtual std::string errorMessage(void);
  virtual bool connectionOk(void);
  void clearLocalError(void);
  std::string getLocalError(void);
};
```

PgEMdFDB

```
#include <pgemdfdb.h>

class PgEMdFDB : public EMdFDB {
public:
  PgEMdFDB(std::string host,         // Hostname to connect to
                                     // (e.g., "localhost"),
           std::string user,         // PostgreSQL user (e.g., "emdf")
           std::string passwd,       // PostgreSQL password (e.g., "changeme"),
           std::string database_name // Initial database to connect to
                                     // (e.g., "emdf").
          );
  virtual ~PgEMdFDB();
};
```

MySQLEMdFDB

```
#include <mysqlemdfdb.h>

class MySQLEMdFDB : public EMdFDB {
public:
  MySQLEMdFDB(std::string host,         // Hostname to connect to
                                        // (e.g., "localhost"),
              std::string user,         // MySQL user (e.g., "emdf")
              std::string passwd,       // MySQL password (e.g., "changeme"),
              std::string database_name // Initial database to connect to
                                        // (e.g., "emdf").
             );
  virtual ~MySQLEMdFDB();
};
```

SQLiteEMdFDB

```
#include <sqliteemdfdb.h>

class SQLiteEMdFDB : public EMdFDB {
public:
  SQLiteEMdFDB(std::string database_name, // Initial database to connect to
                                          // (e.g., "emdf").
               std::string database_key, // "Key" to use with encrypted SQLite dbs.
                                          // Is ignored if there is no encryption availabl
```

```
                                          // (can be purchased separately from the Emdros
            );
  virtual ~SQLiteEMdFDB();
};
```

SQLite3EMdFDB

```
#include <sqlite3emdfdb.h>

class SQLite3EMdFDB : public EMdFDB {
public:
  SQLite3EMdFDB(std::string database_name, // Initial database to connect to
                                           // (e.g., "emdf").
                std::string database_key, // "Key" to use with encrypted SQLite dbs.
                                           // Is ignored if there is no encryption availab
                                           // (can be purchased separately from the Emdros
            );
  virtual ~SQLite3EMdFDB();
};
```

3.3.5 Transactions

Overview Transactions are implemented for PostgreSQL and SQLite (2 and 3) only. The MySQL interface returns false on all three member functions.

The beginTransaction member function is special in that its return value, though boolean, does not mean "success/error". Instead, the return value means the following:

- true: The transaction was started,
- false: The transaction was not started.

The other two functions, commitTransaction and abortTransaction, follow the normal pattern of returning true on success, false on error.

Nested transactions are not supported. If a transaction is currently in progress, beginTransaction will return false.

C++ interface

```
#include <emdfdb.h>

class EMdFDB {
public:
```

```
  // Transactions
  virtual bool beginTransaction(void);
  virtual bool commitTransaction(void);
  virtual bool abortTransaction(void);
};
```

C++ example The following shows how to use transactions. pDB is a pointer to the database object.

```
  bool bDoCommit;  // Should we commit later?
  bDoCommit = pDB->beginTransaction(); // Possibly begin transaction

  // Process...

  if (/* something went wrong */) {
    // Recover...
    if (bDoCommit) {
      if (!pDB->abortTransaction()) {
        // Abort failed
      }
    }
    // Exit from function
  }

  // More processing...

  // We're done!

  if (bDoCommit) {
    if (!pDB->commitTransaction()) {
      // Commit failed.
    }
  }
```

3.4 MQL execution environment

3.4.1 Overview

The MQL execution environment is a container for everything the MQL engine needs to know about its environment in order to function.

It is recommended that you create an MQLExecEnv only indirectly, through an EmdrosEnv.

3.4.2 C++ interface

```
#include <mql_execution_environment.h>

#define COMPILER_STAGE_NONE   (0)
#define COMPILER_STAGE_PARSE  (1)
#define COMPILER_STAGE_WEED   (2)
#define COMPILER_STAGE_SYMBOL (3)
#define COMPILER_STAGE_TYPE   (4)
#define COMPILER_STAGE_MONADS (5)
#define COMPILER_STAGE_EXEC   (6)

class MQLExecEnv {
public:
  Statement    *pStatement;  // Only valid after a successful parse.
                             // Deleted and set to nil by clean().
  EMdFDB* pDB;
  EMdFOutput *pOut;
  MQLError *pError;  // Initialized by constructor, deleted by destructor
  MQLExecEnv(EMdFDB* pMyDB, EMdFOutput *pMyOut);
  ~MQLExecEnv();
  int nCompilerStage; // Shows you which compiler stage went wrong
  void clean(void);   // Must be called before each query is executed,
                      // but is called automatically by the
                      // mqlExecuteXXX functions and so, by proxy, the
                      // EmdrosEnv::executeXXX methods
};
```

3.5 MQL Error

3.5.1 Overview

The MQL Error class encapsulates a string which contains the latest error message from the MQL compiler. Any database error must be gotten from the EMdFDB interface.

3.5.2 C++ interface

```
#include <mql_error.h>

class MQLError {
public:
  MQLError();
  ~MQLError();
```

```
  void clearError(void);
  std::string getError(void);
};
```

3.6 MQL language

3.6.1 Overview

The MQL language interface consists of three functions, each of which differs from the others only in the kind of input it takes:

- **mqlExecuteFile** takes a the name of a file
- **mqlExecuteStream** takes a descendant of std::istream
- **mqlExecuteString** takes a std::string

3.6.2 Arguments

The following are the other arguments, common to them all:

- pEE is the execution environment.
- bResult& is a boolean reference returning true on success and false on error.
- bPrintResult is a boolean indicating whether the results should be dumped using the EMdFOutput object in the execution environment (true) or not (false). Set this to false if you plan to use the pStatement of the execution environment for getting the results.
- bReportError is a boolean indicating whether any error message should be reported using the EMdFOutput object in the execution environment. This only has an effect if the output kind is kOKConsole, since any error message is always given in the XML output.

3.6.3 C++ interface

```
#include <mql_execute.h>

extern bool mqlExecuteFile(MQLExecEnv *pEE,
                           std::string filename,
                           bool& bResult,
                           bool bPrintResult,
                           bool bReportError);

extern bool mqlExecuteStream(MQLExecEnv *pEE,
                             std::istream& strin,
                             bool& bResult,
```

```
                                bool bPrintResult,
                                bool bReportError);

extern bool mqlExecuteString(MQLExecEnv *pEE,
                                const std::string& input,
                                bool& bResult,
                                bool bPrintResult,
                                bool bReportError);
```

3.7 Monads

3.7.1 Overview

The monads interface is useful in all kinds of ways, since monads are so fundamental to Emdros.

3.7.2 C++ interface

```
#include <monads.h>

class BadMonadsException {};

class MonadSetElement {
public:
    MonadSetElement(monad_m first, monad_m last);
    MonadSetElement(monad_m monad);
    monad_m first(void) const; // Gets the member variable
    monad_m last(void) const;  // Gets the member variable
    void printConsole(EMdFOutput *pOut) const;
    void printXML(EMdFOutput *pOut) const;

    // Returns true on this and b having the same first_m and
    // this and b having the same last_m.
    bool equals(const MonadSetElement& b) const;

    // Returns true on this object being
    // "lexicographically" before other object.
    // That is, true if and only if
    // this->first_m < other.first_m
    // or (this->first_m == other.first_m
    //     and
    //     this->last_m < other.last_m)
    bool isBefore(const MonadSetElement& other) const;
};
```

```
class SOMConstIterator {
public:
  SOMConstIterator();
  ~SOMConstIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  const MonadSetElement& next(); // Gets current and advances iterator afterwards
  const MonadSetElement& current(); // Gets current without altering iterator
};


class SetOfMonads {
public:
  monad_m first() const;  // First in set
  monad_m last() const;   // Last in set
  SOMConstIterator const_iterator() const;
  std::ostream& putme(std::ostream& s) const;
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput *pOut) const;
  SetOfMonads() {};
  /** Constructor from compact string.
   *
   * Constructs a set from a compact string.  This would previously
   * have been gotten from SetOfMonads::toCompactString().
   *
   * @param compactString The compact string to create from.
   */
  SetOfMonads(const std::string& compactString) : m_first(MAX_MONAD), m_last(0) {
       fromCompactString(compactString); };
  /** Constructor for singleton set.
   *
   * Constructs a set with (initially) only one monad, \p m.
   *
   * @param m The monad to add to the empty set.
   */
  SetOfMonads(monad_m m) { add(m); };
  /** Constructor for single range set.
   *
   * Constructs a set with (initially) a single range, from \p first
   * to \p last.
   *
   * @param first The first monad of the range to add to an empty set.
   * @param last The last monad of the range to add to an empty set.
   */
  SetOfMonads(monad_m first, monad_m last) : m_first(first), m_last(last) {
         monad_ms.push_back(MonadSetElement(first,last)); };
  ~SetOfMonads();
  SetOfMonads& operator=(const SetOfMonads& som); // Not SWIG-wrapped
  std::string toString(void) const; // Get string-representation
  bool part_of(const SetOfMonads& other) const;
  void unionWith(const SetOfMonads& other);
```

```
  void difference(const SetOfMonads& other);
  static SetOfMonads intersect(const SetOfMonads& Aset, const SetOfMonads& Bset);
  void addMSE(MonadSetElement mse);
  void add(monad_m monad);
  void add(monad_m first, monad_m last);
  bool equals(SetOfMonads& other) const; // A proxy for operator==
  bool operator==(SetOfMonads& other) const; // Not SWIG-wrapped
  bool isMemberOf(monad_m m) const;
  bool isEmpty(void) const;
  void getMonad_mVector(std::vector<monad_m>& monad_vec) const; // Not SWIG-wrapped
  void getMonad_mList(std::list<monad_m>& monad_list) const; // Not SWIG-wrapped
  void getGapVector(std::vector<monad_m>& gap_vec) const; // Not SWIG-wrapped
  void removeMSE(const MonadSetElement& mse);
  bool gapExists(monad_m Sm, monad_m& m) const; // Gap exists in
    // set, starting at Sm, finishing at m
  void offset(monad_m m); // Add m to all mse's in set
  void clear(); // Make empty
  std::string toCompactString(void) const;
  void fromCompactString(const std::string& inStr);
  /** Check whether set is a singleton or single range.
   *
   * Returns true if there is only one MonadSetElement in the set.
   * This can be either a singleton or a single range.
   *
   * @return true if there is only one MonadSetElement, false if not.
   */
  bool hasOnlyOneMSE(void) const;
};
```

3.7.3 Examples

A **SetOfMonads** can be used like this:

```
  // Declare and fill som with monads
  SetOfMonads som;
  som.add(1);      // Now is { 1 }
  som.add(3,6);    // Now is { 1, 3-6 }
  som.add(10,13) ; // Now is { 1, 3-6, 10-13 }

  // Get string representation
  std::string stringRepresentation = som.toString();

  // Declare and fill som2 with monads
  SetOfMonads som2;
  som2.add(2,7);  // Now is { 2-7 }

  // Declare and fill som3 with monads
```

```
SetOfMonads som3;
som3.add(2,4);  // Now is { 2-4 }


// Add the monads of som2 to som
som.unionWith(som2) // som is now { 1-7, 10-13 }

// Get set intersection of som and som3
SetOfMonads som4;
som4 = SetOfMonads::intersect(som, som3); // som4 is now { 2-4 }

// Subtract the monads of som2 from som
som.difference(som2); // som is now { 10-13 }
```

The **SOMConstIterator** can be used like this:

```
SetOfMonads som; // Assumed to be initialized from somewhere
SOMConstIterator sci = som.const_iterator();
while (sci.hasNext()) {
  MonadSetElement mse = sci.current();

  // Process mse ...

  // Are there any left?
  if (sci.hasNext()) {
     // Do something if the just-processed mse is not the last
  }

  sci.next();
}

// Or like this:
SetOfMonads som; // Assumed to be initialized from somewhere
SOMConstIterator sci = som.const_iterator();
while (sci.hasNext()) {
  MonadSetElement mse = sci.next();

  // Process mse ...

  // Note how there is no si.next() again, since next()
  // first gets current element and then advances iterator.

}
```

3.8 Statement

3.8.1 Overview

The Statement class is the base class for all MQL statements. It contains the MQLResult object associated with each execution of a statement.

When executing more than one statement at a time in a single call to EmdrosEnv::executeXXX or mqlExecuteXXX, only the results from the last statement executed are available afterwards. Executing more than one query is mostly used when parsing the output from the mql(1) program.

3.8.2 Example

You can get to the results of an MQL statement via the following path:

```
pEE->getStatement()->getResult(); // pEE is the EmdrosEnv environment
```

3.8.3 C++ interface

```
#include <mql_types.h>

// Base class for all statements
class Statement {
public:
  MQLResult* getResult();
};
```

3.9 Table

3.9.1 Overview

The table is the kind of output returned from all MQL statements which return something, except for "SELECT (ALL|FOCUS) OBJECTS", which returns a sheaf.

A table is a list of **TableRow**s. A TableRow, in turn, is a list of strings.

The columns are numbered beginning with 1, not 0.

Java-style iterators are provided for accessing the rows and their contents.

3.9.2 C++ interface

```
#include <table.h>

typedef enum {
```

```
  kTCString,
  kTCInteger,
  kTCID_D,
  kTCMonad_m,
  kTCBool,
  kTCEnum
} TableColumnType;

class TableRowIterator {
public:
  TableRowIterator(TableRow *pMotherTableRow);
  TableRowIterator();
  ~TableRowIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  std::string next(); // Gets current and advances iterator afterwards
  std::string previous(); // Regresses iterator and then gets current
  std::string current(); // Gets current without altering iterator
  std::string getColumn(int column); // Gets column in current without touching iterator.
};

class TableRow : public std::list<std::string> {
public:
  TableRow();
  ~TableRow();
  TableRowIterator iterator();  // Gets iterator pointing to beginning
  std::string getColumn(int column); // Gets column.  First column is 1.
  unsigned int size() const;  // Gets number of columns
};

class TableIterator {
public:
  TableIterator();
  TableIterator(Table *pMotherTable);
  TableIterator(const TableIterator& other);
  ~TableIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  TableRow& next(); // Gets current and advances iterator afterwards
  TableRow& previous();  // Regresses iterator and then gets current
  TableRow& current(); // Gets current without altering iterator
  std::string next(int column);  // Gets column in current and then advances iterator
  std::string getColumn(int column); // Gets column in current without altering iterator.
};


class Table {
public:
  Table();
  ~Table();
  // Reading from a table
  TableIterator iterator();
```

```
  // Find row with string str at column column_index. First column is 1.
  TableIterator find(const std::string& str, int column_index,
                     bool bCompareCaseInsensitively = true);

  // Find row with str1 and str2 at their respective columns. First column is 1.
  TableIterator find(const std::string& str1, int column_index1,
                      const std::string& str2, int column_index2,
                      bool bCompareCaseInsensitively = true);
  void erase(TableIterator& d);
  void clear();

  // First column is 1.
  std::string getColumn(TableIterator& i, int column_index);
  long size() const; // Get number of rows

  // Adding to a table
  void append(const std::string& str);
  void newline();
  void appendHeader(std::string header_name,
                    TableColumnType tc_type,
                    std::string enum_type = "");

  // First column is 1.
  void updateRow(TableIterator& i, const std::string new_value, int column_index);

  // Should be called before append.
  void startNewRow();

  // Printing a table
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput *pOut) const;
  static void printDTD(EMdFOutput *pOut);
};
```

3.9.3 Examples

Iterators (reading from a table) Here is an example of how to use the iterators.

```
  Table *pTable; // Assumed to be initialized from somewhere

  // Get first string of first row
  TableIterator ti = pTable->iterator();
  std::string firstRowCol = pTable->getColumn(ti, 1);  // Note how column 1 is the first.
```

```
// out table
ti = pTable->iterator();
while (ti.hasNext()) {
  // Get iterator for current row
  TableRowIterator tri = ti.current().iterator();

  // out row
  while (tri.hasNext()) {
    // out string plus horizontal tab, and advance iterator
    printf("%s\t", tri.next());
  }

  // out newline
  printf("\n");

  // Advance iterator
  ti.next();
}

// Find something in table
int column_to_search = 2;
TableIterator ti2 = pTable->find("StringToFind", column_to_search);
if (ti2.hasNext()) {
  printf("String found!\n");
  TableRow& tr = ti.current();
  printf("Column 1 = '%s'\n", tr.getColumn(1));
} else {
  printf("String not found.\n");
}
```

Adding to a table Here's an example of how to add to a table:

```
// Create header (not really necessary if you know what you've
// placed into the table; mostly for aesthetics when printing
// and for communicating with other parts of the program)
pTable->appendHeader("first_monad", kTCMonad_m);
pTable->appendHeader("last_monad", kTCMonad_m);

// Start a new row (call this method before starting a new row)
pTable->startNewRow();

// Add row 1
pTable->append(1); // Monad 1 in first_monad
pTable->append(4); // Monad 4 in last_monad
```

```
// Add row 2
pTable->startNewRow();
pTable->append(10); // Monad 10 in first_monad
pTable->append(13); // Monad 13 in last_monad
```

3.10 Sheaf

3.10.1 Overview

The sheaf is the datastructure returned by the "SELECT (ALL|FOCUS) OBJECTS" query. See the MQL User's Guide Chapter 4 for a description of the sheaf.

Briefly, a **sheaf** is a list of straws. A **straw** is a list of matched_objects. A **matched_object** may have an inner sheaf, which makes the datastructure recursive.

A matched_object corresponds to an object_block(_first) or (opt_)gap_block in a topographic MQL query. A straw corresponds to a block_string. A sheaf corresponds to a blocks.

A matched_object may be one of two kinds:

- kMOKID_D : Match of object_block(_first)
- kMOKID_M : Match of (opt_)gap_block

Use the getKind() method to find out which it is, if you don't already know from the query.

A matched_object contains the following information:

- An eMOKind enumeration (either kMOKID_D or KMOKID_M)
- An id_d (valid if the eMOKind enumeration is kMOKID_D)
- An object type (valid if the eMOKind enumeration is kMOKID_D)
- A set of monads (always valid)
- A focus boolean (always valid)
- An inner sheaf (always valid)
- A vector of EMdFValue values, non-empty if the object block had a GET clause.

Java-style iterators are provided.

3.10.2 C++ interface

```
#include <mql_sheaf.h>


typedef enum {
  kMOKNIL_mo,   // Used only while building sheaf, never in finished sheaf
  kMOKEMPTY_mo, // Used only while building sheaf, never in finished sheaf
  kMOKID_D,     // Match of object_block(_first)
  kMOKID_M      // Match of (opt_)gap_block
} eMOKind;

class MatchedObject {
public:
  MatchedObject(const MatchedObject& other);
  ~MatchedObject();
  const SetOfMonads& getMonads(void) const;
  id_d_t getID_D(void) const;
  const Sheaf* getSheaf(void) const;
  bool sheafIsEmpty(void) const;
  eMOKind getKind(void) const;
  /** Return true if this is an ID_M.
   *
   * @return \p true if this is an ID_M matched object,
   * \p false if it is an ID_D matched object.
   */
  bool isID_M(void) const { return m_id_d == 0 || m_id_d == -1; };
  /** Return true if this is an ID_D.
   *
   * @return \p true if this is an ID_D matched object,
   * \p false if it is an ID_M matched object.
   */
  bool isID_D(void) const { return m_id_d > 1 || m_id_d < -1; };
  /** Return true if the block from which this came had the FOCUS keyword.
   *
   * @return \p true if this matched object is in FOCUS,
   * \p false otherwise.
   */
  bool getFocus(void) const;
  MatchedObject& operator=(const MatchedObject& other); // Not SWIG-wrapped
  short int getObjectTypeIndex(void) const { return m_object_type_index; };

  // Returns 0 on index out of range.
  // Is the fastest
  const EMdFValue *getEMdFValue(int index) const;

  // Returns 0 on index out of range.
  // Is slow.
```

```
  const EMdFValue *getEMdFValue(const std::string& feature_name) const;

  // Get the index of a feature name to be used for
  // getEMdFValue(short int index).
  // Returns -1 on not found
  int getEMdFValueIndex(const std::string& feature_name) const;


  // Get list of feature-names in the order in which they appear in
  // the value vector.  Thus this list can be used as a basis for
  // getting the indexes of the EMdFValues to be used with
  // MatchedObject::getEMdFValue().
  StringList getFeatureList(void) const;

  // Get number of values in list of EMdFValue's.
  unsigned int getNoOfEMdFValues(void) const;

  // Get name of object type for this MatchedObject (may be pow_m).
  std::string getObjectTypeName() const;

  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput* pOut) const;

  // See Sheaf::getSOM() for an explanation
  void getSOM(SetOfMonads& som, bool bUseOnlyFocusObjects) const;

};

class StrawConstIterator {
public:
  StrawConstIterator(const Straw *pMotherStraw);
  StrawConstIterator();
  StrawConstIterator(const StrawConstIterator& other);
  ~StrawConstIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  const MatchedObject* next(); // Gets current and advances iterator afterwards
  const MatchedObject* previous(); // Regresses iterator and then gets current
  const MatchedObject* current(); // Gets current without altering iterator
};

class Straw {
private:
  friend class StrawConstIterator;
  monad_m m_last;
  MOList m_list;
public:
  ~Straw();
  Straw(const Straw& other);
  StrawConstIterator const_iterator() const;
  Straw& operator=(const Straw& other); // Not SWIG-wrapped
```

```
  monad_m getLast(void) const { return m_last; };
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput* pOut) const;

  // See Sheaf::getSOM() for an explanation
  void getSOM(SetOfMonads& som, bool bUseOnlyFocusObjects) const;
};

class SheafIterator {
public:
  SheafIterator(ListOfStraws *pMotherList);
  SheafIterator();
  SheafIterator(const SheafIterator& other);
  ~SheafIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  Straw* next(); // Gets current and advances iterator afterwards
  Straw* previous(); // Regresses iterator and then gets current
  Straw* current(); // Gets current without altering iterator
};



class SheafConstIterator {
public:
  SheafConstIterator(const ListOfStraws *pMotherList);
  SheafConstIterator();
  SheafConstIterator(const SheafConstIterator& other);
  ~SheafConstIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  const Straw* next(); // Gets current and advances iterator afterwards
  const Straw* previous(); // Regresses iterator and then gets current
  const Straw* current(); // Gets current without altering iterator
};



class ListOfStraws {
public:
  ~ListOfStraws();
  ListOfStraws(const ListOfStraws& other);
  SheafIterator iterator();
  SheafConstIterator const_iterator() const;
  bool isEmpty() const;
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput* pOut) const;
};

class Sheaf {
public:
  Sheaf(const Sheaf& other);
```

```
  ~Sheaf();
  bool isFail(void) const;
  SheafIterator iterator();
  SheafConstIterator const_iterator() const;
  const ListOfStraws* get_plist(void) const { return m_plist; };
  Sheaf& operator=(const Sheaf& other); // Not SWIG-wrapped
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput* pOut) const;
  // out sheaf's contribution to DTD
  static void printDTD(EMdFOutput* pOut);

  // Gets big-union of sets of monads in sheaf's matched_objects.
  // This is done recursively through all straws and inner sheafs.
  //
  // If  bUseOnlyFocusObjects is true, only matched_objects with their
  // focus boolean set to true will be included. Otherwise, all matched_objects
  // are considered.

  // The method makes no distinction between matched_objects arising from
  // (opt-)gap-blocks and object blocks.
  //
  // You should probably start with an empty set of monads unless you
  // want to include monads that are not in the sheaf.
  virtual void getSOM(SetOfMonads& som, bool bUseOnlyFocusObjects) const;

  // Here is a version which starts with an empty set and returns the result
  // rather than passing it as a parameter.
  virtual SetOfMonads getSOM(bool bUseOnlyFocusObjects) const;
};


////////////////////////////////////////////////////////////
//
// Flat sheaf
//
////////////////////////////////////////////////////////////

class FlatStraw {
public:
  FlatStraw(id_d_t object_type_id, const std::string& object_type_name);
  ~FlatStraw();
  FlatStrawConstIterator const_iterator(void) const;
  void addMO(const MatchedObject *pMO);
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput* pOut) const;
};

class FlatStrawConstIterator {
public:
  FlatStrawConstIterator();
```

```
  FlatStrawConstIterator(const FlatStrawConstIterator& other);
  ~FlatStrawConstIterator() {};
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  MatchedObject *next(); // Gets current and advances iterator afterwards
  MatchedObject *current(); // Gets current without altering iterator
};




class FlatSheafConstIterator {
public:
  FlatSheafConstIterator();
  FlatSheafConstIterator(const FlatSheafConstIterator& other);
  ~FlatSheafConstIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  FlatStraw* next(); // Gets current and advances iterator afterwards
  FlatStraw* current(); // Gets current without altering iterator
};

class FlatSheaf {
public:
  FlatSheaf(); // For "all" object types
  // For only a select few -- isn't SWIG-wrapped
  FlatSheaf(const std::list<std::pair<id_d_t, std::string> >& object_types);
  ~FlatSheaf();
  bool isFail(void) const { return m_bIsFail; };
  void printConsole(EMdFOutput *pOut) const;
  void printXML(EMdFOutput* pOut) const;
  static void printDTD(EMdFOutput *pOut);
  FlatSheafConstIterator const_iterator(void) const;
};


// For all object types in the sheaf.
// Is not SWIG-wrapped. Use the interface in
// MQLResult instead.
extern FlatSheaf *mql_flatten_sheaf(const Sheaf *pSheaf);

// Only for certain object types
// Is not SWIG-wrapped. Use the interface in
// MQLResult instead.
extern FlatSheaf *mql_flatten_sheaf(StringList *pObjectTypeNames, EMdFDB *pDB,
                                    const Sheaf *pSheaf);

// Only for certain object types
// Is not SWIG-wrapped. Use the interface in
// MQLResult instead.
extern FlatSheaf *mql_flatten_sheaf(StringList *pObjectTypeNames, EmdrosEnv *pEnv,
                                    const Sheaf *pSheaf);
```

3.10.3 Example

The iterators are used like this:

```
Sheaf *pSheaf; // Assumed to be initialized from somewhere
SheafConstIterator sci = pSheaf->const_iterator();
while (sci.hasNext()) {
  const Straw *pStraw = sci.current();

  // Process pStraw ...

  sci.next();
}

// Or like this:
SheafIterator si = pSheaf->iterator();
while (si.hasNext()) {
  Straw *pStraw = si.next();

  // Process pStraw ...

  // Note how there is no si.next() again, since next()
  // first gets current element and then advances iterator.

  // NOTE: This is also possible with the const iterators.

}
```

3.11 EMdFValue

3.11.1 Overview

EMdFValue is returned from MatchedObject::getEMdFValue(). It holds an EMdF value, i.e., one of the following:

- Integer
- ID_D
- String
- Enum

- List of Integer (IntegerList).
- List of ID_D

3.11.2 Enums

The enum constants returned are integers, both for kEVEnum and kEVListOfInteger (which is also used for lists of enum constants). You must use ¡page_anchor ID=""¿EmdrosEnv::getEnumConstNameFromValue() in order to retrieve the string-form of the enum.

3.11.3 C++ interface

```
#include <emdf_value.h>

typedef enum {
  kEVInt,
  kEVEnum,
  kEVID_D,
  kEVString,
  kEVListOfInteger, // Also used for enums
  kEVListOfID\_D
} eEVkind;

class EMdFValue {
public:
  // Copy-constructor
  EMdFValue(const EMdFValue& other);

  // kind must be kEVInt, kEVEnum or kEVID_D
  EMdFValue(eEVkind kind, long i);

  // kind becomes kEVString
  EMdFValue(const std::string& str);

  // Kind must be kEVListOfInteger or kEVListOfID\_D
  EMdFValue(eEVkind kind, IntegerList *pIntegerList);

  ~EMdFValue();

  // Get kind
  eEVkind getKind(void) const;

  // Get if it is a string
  const std::string getString(void) const;

  // Get if it is an ID_D
  id_d_t getID_D(void) const;
```

```
  // Get if it is an integer
  long getInt(void) const;

  // Get if it is an enum
  long getEnum(void) const;

  // Get if it is kEVListOfInteger or kEVListOfID\_D
  IntegerList *getIntegerList(void) const;

  // Get a string representation
  void toString(std::string& result) const;

  // Change string-value. Must be kEVString
  void changeString(const std::string&newString);

  // Assignment operator
  EMdFValue& operator=(const EMdFValue& other); // Not SWIG-wrapped
};
```

3.12 MQLResult

3.12.1 Overview

The MQLResult class encapsulates the results of an MQL query. It is either a sheaf or a table. If it is a sheaf, the table-part will be empty. If it is a table, getSheaf() will return nil. If it is a sheaf, getSheaf will return a pointer to the sheaf.

3.12.2 C++ interface

```
#include <mql_result.h>

class MQLResult : public Table {
public:
  ~MQLResult();
  Sheaf *getSheaf(void);
  Table *getTable(void);
  FlatSheaf *getFlatSheaf(void);

  bool isSheaf() const;
  bool isFlatSheaf() const;
  bool isTable() const;

  // out prints either XML or Console output, based on pOut
  void out(EMdFOutput *pOut) const;
```

```
  // printDTD prints the MQLResult's contributions to the DTD
  static void printDTD(EMdFOutput* pOut);

  // For all object types in the sheaf
  bool flatten();

  // Only for the object types in pObjectTypeNames.
  bool flatten(StringList *pObjectTypeNames, EMdFDB *pDB);
};
```

3.13 StringList

3.13.1 Overview

A StringList is for communicating a list of strings between Emdros and the program using Emdros.

The list is ordered, and can be iterated with the normal Emdros-iterators.

```
#include <string_list.h>

class StringListConstIterator {
public:
  StringListConstIterator();
  StringListConstIterator(const StringList *pMotherStringList); // Not SWIG-wrapped.
  StringListConstIterator(const StringListConstIterator& other);
  ~StringListConstIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  std::string next(); // Gets current and advances iterator afterwards
  std::string previous();  // Regresses iterator and then gets current
  std::string current(); // Gets current without altering iterator
};

class StringList {
public:
  // Create empty string list, to be populated later
  // with addStringListNodes.
  StringList();

  // Copy constructor
  StringList(const StringList& other);

  // Assignment operator
  const StringList& operator=(const StringList& other); // Not SWIG-wrapped!

  ~StringList();

  StringListConstIterator const_iterator() const;
```

```
  // Add at beginning of list, pushing everything down one member
  void addStringFront(const std::string& str);

  // Add at end of list
  void addStringBack(const std::string& str);

  bool isEmpty(void) const;
};
```

3.14 IntegerList

3.14.1 Overview

An IntegerList is for communicating a list of integers (C++ "long"s, actually) between Emdros and the program using Emdros. Internally, monad_ms, id_d_ts, and enumeration constant values are all longs. Thus an IntegerList can hold any of those kinds of values.

The list is ordered, and can be iterated with the normal Emdros-iterators.

The interface of IntegerList is very, very similar to StringList. They are even defined using the same template in the same header-file.

```
#include <string_list.h>

class IntegerListConstIterator {
public:
  IntegerListConstIterator();
  IntegerListConstIterator(const IntegerList *pMotherIntegerList); // Not SWIG-wrapped.
  IntegerListConstIterator(const IntegerListConstIterator& other);
  ~IntegerListConstIterator();
  bool hasNext() const; // Is the iterator == end iterator?  Doesn't alter iterator
  long next(); // Gets current and advances iterator afterwards
  long previous();  // Regresses iterator and then gets current
  long current(); // Gets current without altering iterator
};

class IntegerList {
public:
  // Create empty string list, to be populated later
  // with addIntegerFront and/or addIntegerBack.
  IntegerList();

  // Copy constructor
  IntegerList(const IntegerList& other);

  // Assignment operator
  const IntegerList& operator=(const IntegerList& other); // Not SWIG-wrapped!
```

```
  ~IntegerList();

  IntegerListConstIterator const_iterator() const;

  // Add at beginning of list, pushing everything down one member
  void addIntegerFront(long l);

  // Add at end of list
  void addIntegerBack(long l);

  bool isEmpty(void) const;
};
```

3.15 XML output

3.15.1 Overview

This section describes the XML DTD.

3.15.2 DTD

```
<!-- Top-level element is "mql_results" -->
<!DOCTYPE mql_results
  [

  <!-- "mql_results" is zero or more "mql_result"'s
  <!ELEMENT mql_results (mql_result)* >
  <!ATTLIST mql_results >


  <!-- "mql_result" consists of:
       * An optional "mql_progress",
       * A "status"
       * An optional "error"
       * An optional "warning"
       * Either a "table", a "sheaf", or nothing

       An "mql_result" is emitted for each statement in the
       input stream.
  -->
  <!ELEMENT mql_result ((mql_progress)?, status, (error)?,
                        (warning)?, (table|sheaf)?) >
  <!ATTLIST mql_result  >
```

```
<!-- "mql_progress" is just a series of dots emitted at intervals
     while processing a "SELECT (FOCUS|ALL) OBJECTS" query.
-->
<!ELEMENT mql_progress (#PCDATA)* >


<!-- The empty "status" element shows, via its "success" attribute
     whether the statement succeeded or not (i.e., whether an error
     occurred.
-->
<!ELEMENT status EMPTY >
<!ATTLIST status
              success (true|false) #REQUIRED >


<!-- The "error" element is only emitted if an error occurred. -->
<!ELEMENT error (error_source, error_stage,
                 error_message_db, error_message_compiler) >
<!ATTLIST error >

<!-- The "error_source" element, being part of the "error" element,
     is only emitted if an error occurred.  It shows the source of
     the error: whether it was the MQL compiler or the database
     back-end.
-->
<!ELEMENT error_source EMPTY >
<!ATTLIST error_source
              source  (compiler|db) #REQUIRED >


<!-- The "error_stage" element, being part of the "error" element,
     is only emitted if an error occurred.  It shows in which stage
     of the compiler the error occurred.
-->
<!ELEMENT error_stage EMPTY >
<!ATTLIST error_stage
              stage   (none|parse|weed|symbol|type|monads|exec) #REQUIRED >


<!-- The "error_message_db" element, being part of the "error" element,
     is only emitted if an error occurred.  It contains any error-message
     from the back-end.
-->
<!ELEMENT error_message_db (#PCDATA)* >
<!ATTLIST error_message_db >


<!-- The "error_message_compiler" element, being part of the "error"
     element, is only emitted if an error occurred.  It contains any
     error-message from the compiler.
```

```
-->
<!ELEMENT error_message_compiler (#PCDATA)* >
<!ATTLIST error_message_compiler >

<!-- The "warning" element shows the text of any warning from the
     compiler.
-->
<!ELEMENT warning (#PCDATA)* >
<!ATTLIST warning >



<!-- *************
     *** Table ***
     *************
-->

<!-- A "table" is a "theader" followed by zero or more "trow"'s. -->
<!ELEMENT table (theader, trow*) >
<!ATTLIST table >


<!-- A "theader" is zero or more "theadercolumn"'s -->
<!ELEMENT theader (theadercolumn)* >
<!ATTLIST theader >


<!-- A "theadercolumn" is a "tcaption" followed by a "ttype" -->
<!ELEMENT theadercolumn (tcaption, ttype) >
<!ATTLIST theadercolumn >


<!-- A "tcaption" gives the name of a column-header. -->
<!ELEMENT tcaption (#PCDATA)* >
<!ATTLIST tcaption >


<!-- A "ttype" gives the type of a column.  If the type is "enum",
     the "enum_type" attribute gives the name of the enum.
-->
<!ELEMENT ttype EMPTY >
<!ATTLIST ttype
   type (bool|string|integer|id_d|enum) #REQUIRED
   enum_type CDATA #IMPLIED >


<!-- A "trow" is zero or more "tcolumn"'s.  The number of columns in
     a "trow" is always the same throughout the table, and is also
     the same as the number of "theadercolumn"'s in the "theader".
-->
```

```
<!ELEMENT trow (tcolumn)* >
<!ATTLIST trow >


<!-- A "tcolumn" contains the value of a column in a row. -->
<!ELEMENT tcolumn (#PCDATA)* >
<!ATTLIST tcolumn >


<!-- *************
     *** Sheaf ***
     *************
-->

<!-- A "sheaf" is zero or more "straw"'s. -->
<!ELEMENT sheaf (straw)* >
<!ATTLIST sheaf >

<!-- A "straw" is zero or more "matched_object"'s. -->
<!ELEMENT straw (matched_object)* >
<!ATTLIST straw >


<!-- A "matched_object" is a "monad_set" followed by an optional
     "sheaf".  The optional sheaf constitutes a match of the inner
     blocks from the block which gave rise to the matched_object.

     The "focus" boolean tells whether the block in question had the
     "focus" modifier.

     The "object_type_name" attribute is the object type name of the
     object that matched the object_block(_first) which gave rise to
     this matched_object.

     The "id_d" is the object id_d of the object that matched the
     object_block(_first) which gave rise to this matched_object.
-->
<!ELEMENT matched_object (monad_set, (sheaf)?) >
<!ATTLIST matched_object
          object_type_name CDATA #REQUIRED
          focus (false | true) #REQUIRED
          id_d CDATA "nil"
>


<!-- A "monad_set" is one or more "mse"'s." -->
<!ELEMENT monad_set (mse)+ >
<!ATTLIST monad_set >

<!-- An "mse" a "monad set element".  It is a pair of attributes,
```

```
      "first" and "last", each of which gives the first and the last
      monad (a 10-base integer) of the monad set element.  See the
      document "Monad Sets -- Implementation and Mathematical
      Foundations" for an explanation.
  -->
  <!ELEMENT mse EMPTY>
  <!ATTLIST mse
            first CDATA #REQUIRED
            last  CDATA #REQUIRED
  >
]>
```

4 Part III: Tips

In part III, we describe:

- Tips for how to architect your application.
- Tips for loading an Emdros database with existing data.
- Tips for creating a query-application, including:.
  - Tips for how to create an in-memory EMdF database.
- Tips for creating a display-application.

4.1 Your application

4.1.1 Ways you can benefit

Emdros is for storing and retrieving analyzed or annotated text. As such, you are likely to benefit from Emdros in at least one of the following ways:

- Ways you can build an Emdros database:
  - You have an existing text database which you want to import into Emdros.
  - You want to build a database "on the fly" using analyses or annotations you are making.
- Ways you can use an Emdros database:
  - You can display the database contents as they are, based on selection criteria which you make up.
  - You can search the database and display the results.

How you build your application will depend on which of these four goals you have (one or more of them).

4.1.2 Use MQL

The easiest way of writing any application on top of Emdros is probably to take advantage of the MQL language, rather than dealing with the EMdF layer directly. MQL is quite powerful, and allows you to do most things you would want to do with Emdros. Thus however you architect your application, plan on using the MQL User's Guide extensively.

4.1.3 In the following

In the following, we describe:

1. Loading a database
2. Building a query-application
3. Building a display-application

4.2 Loading a database

In order to populate a database, go through the following steps:

1. Create a schema file which contains MQL statements which create all the enumerations and object types and features necessary. Use the CREATE ENUMERATION and CREATE OBJECT TYPE statements for this. See the MQL User's Guide for how to use these statements. The schema file should contain a CREATE DATABASE statement and a USE DATABASE statement at the top of the file.
2. Run this schema file through the mql(1) program.
3. Populate the database. This can be done in two ways:
   (a) Write and run a program which takes any existing data and uses the MQL query language to create new objects. In particular, use the CREATE OBJECT FROM MONADS statement, or the CREATE OBJECTS WITH OBJECT TYPE statement.
   (b) Create the database on-the-fly using some analysis program you have created.

4.2.1 Speeding up population

There are a number of things you can do in order to speed up the population process:

- Generally applicable things, and
- PostgreSQL-specific things.

General

Use CREATE OBJECTS WITH OBJECT TYPE In general, it is faster to use CREATE OBJECTS WITH OBJECT TYPE rather than CREATE OBJECT FROM MONADS. On my machine, it is between 7 and 10 times faster, depending on the backend.

One drawback of CREATE OBJECTS WITH OBJECT TYPE is that you can't get the object id_d back immediately. You would have to query the database (e.g., using SELECT OBJECTS HAVING MONADS IN) to find out the object id_ds of the newly created objects.

If your application requires complex interaction between objects, e.g., setting features based on other objects' id_ds, it may be easier (but certainly not faster) to use CREATE OBJECT FROM MONADS.

Drop indexes In order to speed up queries, one thing you can do is to issue a

```
$ manage_indices --drop databasename
```

command on the command-line before starting the program which populates the database. Then, once all creation has taken place, you can issue the command:

```
$ manage_indices --create databasename
```

in order to recreate the indices. See the manage_indices(1) manual page for more information.

You can also use the MQL statements DROP INDEXES and CREATE INDEXES. See the MQL User's Guide for more information.

PostgreSQL You can also issue a VACUUM command on the database after each part of the database has been loaded, or even while loading the database. The latter can be done in one of two ways:

1. Either with the MQL "VACUUM DATABASE" statement, or
2. With the EMdFDB::Vacuum method if you have the EMdF library linked in.

See the PostgreSQL documentation for details on the VACUUM command. See also the vacuumdb(1) manual page.

Also, you may want to set the option "fsync = false" in the postgresql.conf file (located in /var/lib/pgsql/data on RedHat). See the PostgreSQL documentation for more information. The performance gain, however, may be minimal with PostgreSQL version 7.1 and higher.

4.3 A query-application

If your application has the goal of displaying the results of a SELECT (ALL|FOCUS) OBJECTS query, i.e., if it has the goal of displaying the results of the kind of MQL query which your users will likely be interested in, then the following strategy could be used:

1. Parse the sheaf. Create an internal representation, and extract what you need from that. This will include the monads and object id_ds of the objects in question.
2. Make a big-union of the sets of monads of all the objects you are interested in. You can use the MONAD SET CALCULATION query to build this union. Alternatively, if you have the emdf library linked into your application, you can use the SetOfMonads to calculate the union.
3. Issue a GET OBJECTS HAVING MONADS IN query for each of the object types you need to see (e.g., morpheme) with the monad set which is the union of all monads you are interested in. This will get you all the object id_ds of the objects you are interested in.

   How you go from here depends on how you want to solve the display problem. One strategy could be:

4. Build an in-memory EMdF database with this information.
5. Display the in-memory EMdF database on-screen in a scrolling view, or dump results to a file or stdout.

Next, we describe the design of such an in-memory EMdF database.

4.3.1 An in-memory EMdF database

Benefits Building an in-memory EMdF database has the advantage of being close to the EMdF model, and thus to the data you get back. Also, we have found the EMdF model to be quite powerful when manipulating and displaying data in-memory.

Necessary operations Here are some operations you will likely need:

- Database operations
  - Get object(s) of a given object type at a given monad.
  - Get object by id_d
  - Insert object into database
  - Delete object from database
- Object operations
  - Create object
  - Delete object
  - Get feature value
  - Set feature value

An example application With these operations, you will be able to construct most other operations you may need. For example, if your application domain uses an immediate constituent model, one way of modeling this is to have, in each object type, a feature called "parent", which is an id_d pointing to the parent object. Then a number of desirable operations can easily be implemented in terms of the above operations:

- Get immediate constituents of object
- Get parent of object
- Get siblings (through judicious use of the two above operations)
- is_c_commanding (a boolean)

etc. etc.

An implementation strategy One strategy for implementing an in-memory EMdF database (using C++ terminology as an example) would be the following:

- Have a class which encapsulates the database.
- Have a class which encapsulates object types. Each object type should encapsulate the features it has, as well as their types. Associate each object type with an object type id and store these in the database class.
- Have a class called, e.g., CMonad_d (for "database monad"). This represents a monad_m, and has, for each object type, a list of the object id_ds of the objects which have this monad. In the database class, use either an array or an associative map (e.g., std::map) which associates each monad_m with its CMonad_d.
- Have a class which encapsulates objects, their monads, and their feature values. You can choose either to make the class generic, capable of handling all object types, or to subclass it for each object type in your application domain.
- For each object type, use some sort of associative map (e.g., std::map) which maps id_ds to pointers to object objects, and put these in the database class. You may wish to put all of the objects in one big map, regardless of object type.

This should serve most needs.

Please contribute If you do implement a generic in-memory EMdF database, please consider contributing it to the Emdros project. Contact Ulrik Petersen <ulrikp[at]users.sourceforge.net> with any contributions.

4.4 A display-application

When displaying text from an Emdros database, you need to take the following steps:

1. Find out the object id_ds of the objects which you wish to display. This can either be done with a "SELECT (ALL|FOCUS) OBJECTS" query, or, if you know the monads in which to look, with a "SELECT OBJECTS HAVING MONADS IN" query.

2. Once you have the id_ds you want, proceed from step 2 in the instructions for a query-application.