Studio Modeling Platform™

Embedding MQL Guide

3DEXPERIENCE R2018x

Table of Contents

1 eMatrixMQL

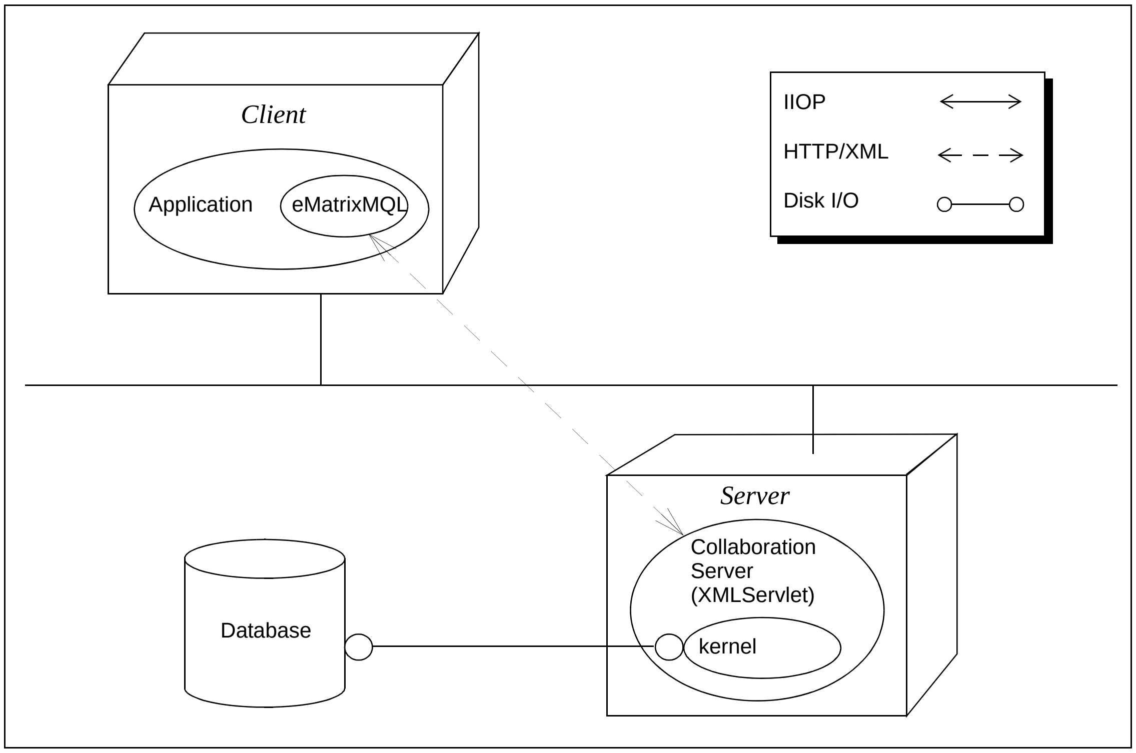
Overview of eMatrixMQL

eMatrixMQL

With eMatrixMQL, an application sends MQL commands to the Live Collaboration Server, which accesses the ENOVIA Live Collaboration database and returns corresponding output.

eMatrixMQL is distributed with the ENOVIA Studio Customization Toolkit.

eMatrixMQL ARCHITECTURE



Requirements

In order to build applications with eMatrixMQL functions, you must have:

- An installed ENOVIA Live Collaboration database, from which to access data
- The eMatrixMQL function library installed

In addition, eMatrixMQL requires:

- The Java 1.3.1 or greater Runtime must be installed
- The path of any calling program must include the jre/bin and jre/bin/class paths of Java
- The environment variable CLASSPATH must include the .jar file that contains the ENOVIA Collaboration Live Collaboration Server classes.

The minimum compiler version for eMatrixMQL functions for Windows is Microsoft Visual C^{++} 6.0 or higher.

Dassault Systèmes Software Prerequisites x86

Before using eMatrixMQL, verify that Dassault Systèmes Software Prerequisites x86 is located in the control panel of your machine. This prerequisite contains Microsoft.VC80.CRT (C++ runtime) and other required libraries.

Usually Dassault Systèmes Software Prerequisites x86 is automatically installed with either the ENOVIA Live Collaboration Server or the ENOVIA Studio Modeling Platform. The prerequisite may not be installed if Microsoft Visual Studio 6.0 Enterprise Edition already exists on the machine. If this is the case, manually run InstallDSSoftwarePrerequisites_x86.msi.

Known Issues

Do Not Use mqlOpen Function More than Once

Using the `mqlOpen` function more than once in a program will cause your program to fail. Do not use `mqlOpen` more than once even if you use `mqlClose` before calling the second `mqlOpen`. To prevent the need for calling `mqlOpen` a second time, don't use `mqlClose` until your program has no further business with ENOVIA Live Collaboration. For example, call `mqlOpen` on initialization and `mqlClose` on termination of the entire Studio Customization Toolkit program.

C++ Studio Customization Toolkit

Installing the ENOVIA Live Collaboration Applet XML Studio Customization Toolkit with the C++ option installs eMatrixMQL libraries and example source code in the INSTALL_DIR\adk\ directory structure. eMatrixMQL is a C++ XML client that can interact with the Live Collaboration Server running on the server.

Refer to the sections that follow for additional requirements for creating and using custom C++ programs.

C++ Studio Customization Toolkit Library Structure

When you install the C++ Studio Customization Toolkit, the following directory structure is created.

```
adk
   | - - -PLATFORM (e.g., win_b64, linux_a64, linux_b64, solaris_a64, solaris_b64)
        |    | - - -code
             |    | - - - bin         Executables and DLLs for the respective platform
             |    | - - - lib         Contains the eMatrixMql static libraries
   | - - -samples
        |- - - eMatrixApplet          Applet wizards, embedded objects, and swing classes
        |- - - eMatrixMql             eMatrixMQL sample source
        |- - - eMatrixMqlTest         Include files and programs that can be used as a test
                                      harness for invoking the eMatrixMQL libraries
```

Setting Up a Client Machine to Run an eMatrixMQL Application

This section describes how to set up a client machine to run an application compiled with eMatrixMQL. It assumes that:

- The client machine does not have the Live Collaboration Server, Studio Customization Toolkit, or any other ENOVIA Live Collaboration component installed.
- The custom application was compiled on a separate server machine that has the Collaboration Server and the C++ Studio Customization Toolkit installed.

The instructions below list the main steps for setting up a Windows client machine for an eMatrixMQL application built on a Windows server machine using the Live Collaboration server Studio Customization Toolkit. The steps are the same for UNIX except:

- The directory references should be / instead of \.
- Instead of win_b64, the subdirectories within the Studio Customization Toolkit installation directories are named according to the platform. See *eMatrixMQL Files (Live Collaboration Server only)*.
- The eMatrixMQL shared library is called eMatrixMql.a for AIX and eMatrixMql.so for Solaris and Linux.

To configure a client machine to run an eMatrixMQL application

1. Copy the Matrix jar file, eMatrixServletRMI.jar from the server machine and paste it to the client machine. You can find the jar file in ENOVIA_INSTALL\PLATFORM\docs\javaserver\.

   Make sure the jar file is the exact same version as the Studio Customization Toolkit used to compile the program.

2. On the client machine, add the path and filename of the ENOVIA Live Collaboration jar file to CLASSPATH.

   The specific name of the file is needed in addition to the full path. Adding only the path produces the error message "Can't find com.matrixone.jdl.MatrixSession".

3. Copy eMatrixMql.dll from the ADK_INSTALL\PLATFORM\code\bin\ directory to the client machine directory that contains the eMatrixMQL application.
4. Copy mxUtil.dll, vgalaxy7u.dll, and vgalaxy7.vr from the ENOVIA_INSTALL\PLATFORM\code\bin\ directory to the client machine directory that contains the eMatrixMQL application.
5. Make sure that the client machine has the correct version of the JDK or the Java Runtime Environment (JRE). Refer to the Program Directory for the given release for supported patch levels.
6. Add the path that includes the Java jvm.dll to the PATH.
7. Now you can run your custom application. For initial testing, you can run the test application that comes with the eMatrixMQL installation. See "*Running the eMatrixMQL Test Application*".

C++ Studio Customization Toolkit Files

eMatrixMQL Files (Live Collaboration Server only)

In the following, PLATFORM = aix_a64 (AIX, Live Collaboration Server only), linux_a64 (Red Hat Enterprise Linux), linux_b64 (Suse Linux), solaris_a64 (Solaris 4), solaris_b64 (Solaris x86), or win_b64 (Windows). All are 64-bit.

The HP 9000 platform (hp9000s800) is no longer supported.

```
ADK_INSTALL\samples\eMatrixMql\eMatrixMql.cpp
ADK_INSTALL\samples\eMatrixMql\eMatrixMql.h
ADK_INSTALL\samples\eMatrixMql\eMatrixMqlUtils.h
ADK_INSTALL\samples\eMatrixMql\StringMacros.h
```

eMatrixMql DLL libraries (Windows only)

```
ADK_INSTALL\PLATFORM\code\bin\eMatrixMql.dll(DLL)
ADK_INSTALL\PLATFORM\code\bin\eMatrixMqlU.dll(Unicode DLL)
```

eMatrixMql Static libraries (Windows only)

```
ADK_INSTALL\PLATFORM\code\lib\eMatrixMql.lib(DLL stub library)
ADK_INSTALL\PLATFORM\code\lib\eMatrixMqlU.lib(Unicode DLL stub
library)
```

eMatrixMql Test Executables

```
ADK_INSTALL\samples\eMatrixMqlTest\eMatrixMqlTest.h
ADK_INSTALL\samples\eMatrixMqlTest\eMatrixMqlTest.cpp
```

```
ADK_INSTALL\PLATFORM\code\bin\eMatrixMQLTest.exe(Test program)
ADK_INSTALL\PLATFORM\code\bin\eMatrixMQLTestU.exe(Unicode test
program)
```

You must also have access to the following Galaxy DLLs via the PATH setting:

```
vgalaxy7u.dll
vgalaxy7.vr
```

Running the eMatrixMQL Test Application

The eMatrixMQL installation includes sample executables (one for unicode and one for non-unicode environments) that you can use as a test harness for invoking the eMatrixMQL libraries. The include files and exe files needed for the test application are in ADK_INSTALL\PLATFORM\code\bin\. This section describes how to set up and run the test application.

Setting Up to Run the Test Application

This section lists the main steps for setting up a Windows system to run the test eMatrixMQL applications. The steps are the same for UNIX except the directory references should be / instead of \.

To install and configure a server to run the test application

1. Install the ENOVIA Live Collaboration Server.
2. Install the Applet Studio Customization Toolkit and choose the C++ Studio Customization Toolkit. Make sure the Studio Customization Toolkit version matches the version of the Live Collaboration Server that you installed.
3. Copy mxUtil.dll, vgalaxy7u.dll, and vgalaxy7.vr from ENOVIA_INSTALL\PLATFORM\code\bin\ to the directory that contains the test application ADK_INSTALL\PLATFORM\code\bin\.

The system can find the DLL if it is located in any directory that is in the operating system's search path for exe files. But placing the DLL in the directory with the test exe files ensures it picks up the correct version, in case you have other versions of the DLL.

4. Add ENOVIA_INSTALL\PLATFORM\docs\javaserver\eMatrixServletRMI.jar to the CLASSPATH.
5. Include Java jvm.dll in the PATH.

Now run the test application using the steps in the following section.

Running the Test Application

To run the test application for eMatrixMQL

1. Create a text file in the ADK_INSTALL\PLATFORM\code\bin\ directory with whatever MQL commands you want to run. This is the script you will pass to the test application.

   For example, you can create a file with the simple command "`version;`" (which works on any database) and save it as version.mql. The resulting ADK_INSTALL\PLATFORM\code\bin\ directory would now contain these files:

   - mxUtil.dll
   - eMatrixMql.dll
   - eMatrixMqlTest.exe
   - eMatrixMqlTestU.exe
   - vgalaxy7u.dll
   - vgalaxy7.vr

- version.mql

2. In the same directory, issue the command to run the test application eMatrixMqlTest.exe. The exe accepts the following arguments:

Command Argument	**Description**	**Example**
eMatrixMqlTest <host server name>	Name of the host server, prefixed with http:// and appended with a colon and the port number and web application name. Make sure the port number is the Live Collaboration Server port (for example, 1099) for the Live Collaboration Server.	//hostname:1099/enovia
-s <mqlscript>	Name of the mql script to pass to the program (stdin).	-s version.mql
-l <outputlog>	Name of log file to send output to (stdout).	-l version.log
-d <level>	Set level of eMatrixMQL diagnostics: 0,1,2; default=1.	-d 1
-h	Display help	-h
-p	Use pending vs. DDE callbacks (=default)	-p
-u <username>[/<pswd>]	Set context to the specified username and password.	-u creator
-v	Turn on verbose output.	-v

For example, to run the version.mql script, you could issue the command:

```
eMatrixMqlTest http://hostname:1099/enovia -s version.mql -l
version.log -d 2 -u creator -v
```

The output window will look as follows:

```
Calling mqlSetVerbose()
Calling mqlSetTrace()
Calling mqlInit()
Calling mqlSetContext() initially
Calling mqlSetContext() again
Calling mqlOpen()
Calling mqlOpenLog()
Calling mqlCommand()
Calling mqlCallback()
Calling mqlOutputLine() through callback
Calling mqlErrors()
Calling mqlCommand()
Calling mqlCallback()
Calling mqlErrors()
Calling mqlCommand()
Calling mqlCallback()
Calling mqlErrors()
Calling mqlCloseLog()
Calling mqlClose()
eMatrixMqlTest Completed.
```

3. If the application runs successfully, you should find the output log file in the same directory. Open the file to check its contents.

   For example, the version.log file would contain something similar to this:

```
Input:version;
Output:10.5-Global
Input:
Input:
```

Troubleshooting

Use this troubleshooting table to help resolve problems with running the test or a custom application.

Error	Remedy
Can’t find or load mxUtil.dll	Copy mxUtil.dll from ENOVIA_INSTALL\PLATFORM\code\bin\ to the same directory as the calling program.
Unable to load eMatrixMql.dll	Copy eMatrixMql.dll from ADK_INSTALL\PLATFORM\code\bin\ to the same directory as the calling program.
Can’t find jvm.dll	The system environment PATH variable is pointing to the wrong jvm.dll. Make sure the JDK or JRE directory that contains the jvm.dll is in your PATH. For example, for JDK 1.6.0_24, the DLL is in the directory \jre\bin\server\.
Can’t create Java vm	The CLASSPATH is pointing to the wrong jvm.dll.
Can't find com.matrixone.jdl.MatrixSession	Make sure the CLASSPATH is defined and includes the full pathname and filename, for example "Set CLASSPATH=\PATHNAME\eMatrixServletRMI.jar"
Dr. Watson errors running the eMatrixMQL application	Be sure that the JRE or JDK is correctly installed and that the system environment PATH variable points to the correct Java binary path.
File checkin / checkout does not work	Need to add the keyword "client" to the checkin and “server” for the checkout command to specify where the operation will occur. For checkin the default is server and for checkout the default is client. Also, the checkout command requires that a target directory specified. Refer to the *MQL Guide* for more information.

Starting a Session

A session is started using the `mqlOpen` command. The `mqlOpen` command will open communications between your client application and MQL or the server. eMatrixMQL uses XML wrapped by HTTP.

In order to track any errors that may occur, it is recommended that a log file be opened as well, using the `mqlOpenLog` command after `mqlOpen` is called. The log file will contain all input to and output from MQL or the server.

A session might begin as follows:

```
    if (mqlSetContext(“WebServer”,”JSmith”,”xyzzy”,”Engineering”) == MQLERROR)
    {
       fprintf(stderr, ”ERROR: Cannot Set Context....”);
    }
```

```
    if (mqlOpen() == MQLERROR)
    {
       fprintf(stderr, ”ERROR: Cannot Open MQL Interface\n”);
       exit(-1);
    }
```

`mqlOpen` returns one of the following:

- **MQLOK (0)** The session has been established.
- **MQLERROR (-1)** The session was unable to be established.

```
    if (mqlOpenLog(“example.log”) == MQLERROR)
    {
       fprintf(stderr, ”ERROR: Cannot Open Log File\n”);
       exit(-1);
    }
```

`mqlOpenLog` returns one of the following:

- **MQLOK (0)** The log file was created without any problems.
- **MQLERROR (-1)** The log file could not be created.

See *mqlOpen*, *mqlClose*, *mqlOpenLog*, *mqlCloseLog* for further information.

Issuing a Command

There is an MQL command length limit of 8K.

Commands are issued to the MQL process using the `mqlCommand` routine. The parameters used to format the commands are just like the C Language `printf` routine, in that it takes multiple parameters and uses format strings to control the layout of the command.

```
    if (mqlCommand( “print businessobject \’%s\’ \’%s\’ \’%s\’ “, type. name.
    revision) == MQLERROR)
    {
       return FAILURE;
    }
```

Assuming that type is equal to ‘Assembly’ and name is equal to ‘Engine’ and revision is equal to ‘A’, this is equivalent to issuing a `print businessobject ‘Assembly’ ‘Engine’ ‘A’` command from the MQL prompt.

In the above example, the return of MQLERROR from `mqlCommand` does NOT mean that the `print businessobject` command failed, but that the `mqlCommand` function failed. Some reasons for failure are (a) the interface has not been initialized or (b) an input error occurred.

`mqlCommand` returns one of the following:

- **MQLOK (0)** The sent command was successfully invoked.
- **MQLERROR (-1)** The sent command was not successfully invoked.

If a callback function is not used when issuing a command, output should be checked with the `mqlPending` command.

The C++ Studio Customization Toolkit interface mqlCommand requires multibyte input to be in Unicode format.

Processing Output with Callbacks

In most programming situations, an application program calls the application programming interface (API), and then the API executes the function called by the application program and returns control, and possibly a value, to the calling program.

In some instances, however, it is necessary for the API to call the application program, either to request additional information or to allow the application program to process some data. The calling of an application program by an API is referred to as a *callback* (thus named because the API is called by the application program and then *calls back*).

Callbacks are often used to handle output or error conditions. To set up a callback, the application program usually *registers*, that is passes to the API, a pointer to a user-defined function. The API generally provides a registration function that accepts a function address as one of its parameters for this purpose. When the specific condition occurs, the API calls the user-defined function (through the registered function pointer) and passes control to that function for execution.

The `mqlCallback` function of eMatrixMQL is a flexible way to extract data from the ENOVIA Live Collaboration database using MQL commands. The `mqlCallback` function requires four parameters:

- A pointer to a function to handle output
- A pointer to an output data area
- A pointer to a function to handle error output
- A pointer to an error output data area

The functions to handle output data and error output data are supplied by the programmer. Simple examples of functions to print data to standard output are given below. They use the eMatrixMQL commands *mqlOutputLine* and *mqlErrorLine*.

```
/* outputCallback.cpp -- function to handle output callback. */
/*****************************************************/
#include <stdio.h>
#include "outputCallback.h"                      /* Function prototype */
#include "eMatrixMQL.h"
/*****************************************************/
#define LINEBUFFERSIZE 8192
/*****************************************************/
int outputCallback(void *outData)
/* Function to handle output callback.
** input: none.
** output: Data from MQL output.
** returns: MQLOK....output callback was handled.
** returns: MQLERROR output callback could not be handled.
*/
{
   char buffer[LINEBUFFERSIZE];                     /* Buffer */
   if (mqlOutputLine(buffer,LINEBUFFERSIZE))
   {
      fprintf(stdout,"%s",buffer);
   }
   return MQLOK;
}
```

```
/* errorCallback.cpp -- function to handle error output callback. */
/************************************************************/
#include <stdio.h>
#include "errorCallback.h"                      /* Function prototype */
#include "eMatrixMQL.h"
/***********************************************************/
#define LINEBUFFERSIZE 8192
/*************************************************************/
int errorCallback(void *errData)
/* Function to handle error output callback.
** input: none.
** output: Data from MQL error output.
** returns: MQLOK.......error output callback was handled.
** returns: MQLERRORerror output callback could not be handled.
*/
{
   char buffer[LINEBUFFERSIZE];                       /* Buffer */
   if (mqlErrorLine(buffer,LINEBUFFERSIZE))
   {
      fprintf(stderr,"%s",buffer);
   }
   return MQLOK;
}
```

The `mqlCallback` call would be:

```
int iStat;
iStat = mqlCallback(outputCallback, NULL, errorCallback, NULL);
```

`mqlCallback` returns one of the following:

- **MQLOK (0)** The callback was successful.
- **MQLERROR (-1)** The callback failed.

See also *mqlCallback*.

Processing Pending Output

MQL output can also be processed using `mqlPending` to determine if output is available for processing.

```
/**************************************************/
int status;
for (;;)
{
   status = mqlPending();
   while (status == MQLOK)
   {
      sleep(1);
      status = mqlPending();
   }
   switch (status)
   {
      case MQLOUTPUTPENDING:
         mqlOutputLine(buffer,size);
                   :
                   :
         break;
      case MQLERRORPENDING:
         mqlErrorLine(buffer,size);
                   :
                   :
         break;
      default:
         break;
   }
}
/**************************************************/
There are four possible returns:
```

- **MQLERRORPENDING (2)** There is error output pending.
- **MQLOUTPUTPENDING (1)** There is pending output.
- **MQLOK (0)** There is no pending output.
- **MQLERROR (-1)** The process timed out.

See also *mqlPending*.

A Small Conversation

Here are two examples of a simple *conversation*, the first using `mqlCallback`, the second using `mqlPending`. The conversation requests the list of vaults using the MQL command `list vault`. All output will be printed to standard out or standard error.

```
/*************************************************/
/* mqlCallback */

   char cmd = "list vault";

   if (mqlCommand(cmd)== MQLERROR)
   {
      fprintf(stderr,"ERROR: mqlCommand() routine has failed\n");
      return FAILURE;
   }

   if (mqlCallback(outputCallback, NULL, errorCallback, NULL) == MQLERROR)
   {
      fprintf(stderr,"ERROR: mqlCallback() routine has failed\n");
      return FAILURE;
   }

   if (mqlErrors())
   {
      fprintf( stderr, "ERROR: MQL Command %s failed\n", cmd );
      return FAILURE;
   }
```

See the example in the *Processing Pending Output* Section, above, for the `outputCallback` and `errorCallback` function definitions.

```
/* mqlPending */

   char cmd = "list vault";
   int errorCnt = 0;
   int outputCnt = 0;
   int status;
   int done;

   if (mqlCommand(cmd) == MQLERROR)
   {
      fprintf(stderr, "ERROR: mqlCommand() routine has failed\n" );
      return FAILURE;
   }
   for (;;)
   {
      status = mqlPending();
      while (status == MQLOK)
      {
         if (mqlEOF())
         {
            done = TRUE;
            break;
         }
         sleep(1);
         status = mqlPending();
      }
      if (done)
         break;

      switch (status)
      {
         case MQLOUTPUTPENDING:
                   mqlOutputLine(buffer,size);
                   fprintf( stdout, "%s\n", buffer );
                   outputCnt++;
                   break;
         case MQLERRORPENDING:
                   mqlErrorLine(buffer,size);
                   fprintf( stderr, "%s\n", buffer );
                   errorCnt++;
                   break;

         default:
                   break;
      }
   }

   if (errorCnt)
   {
      fprintf( stderr, "MQL Command %s failed\n", cmd );
      return FAILURE;
   }
/**************************************************/
```

Library of Functions

Library Listing

The following files are the distributed for use with eMatrixMQL applications:

File	Usage
eMatrixMQL.h	Header File
eMatrixMQL.dll	Windows DLL
eMatrixMQL.lib	Windows DLL stub library (this gets linked to your application vs. the DLL)

Library of eMatrixMQL Functions

This section lists and describes the public functions in the eMatrixMQL library. Syntax and output, as well as additional notes where applicable, are provided in the following section.

Function	Description
mqlCallback	This function is a setup routine to handle MQL or the Collaboration Server output and errors. The user passes in a pointer to a function that returns an integer and a storage location for any results. This is done for both normal and error output.
mqlClose	Ends an open session by terminating the MQL process or the Collaboration Server connection.
mqlCloseLog	Closes the log file opened using mqlOpenLog.
mqlCommand	Sends an MQL command to MQL or the Collaboration Server, using printf style syntax.
mqlDisableLog	Disables writing to the log file.
mqlEnableLog	Enables writing to the log file.
mqlError	Used by mqlCallback to receive error data from MQL or the Collaboration Server.
mqlErrorLine	Used by mqlCallback to receive a line of error data from MQL or the Collaboration Server. It breaks on new line.
mqlErrorMessage	Prints an error message.
mqlErrors	Maintains a counter of the number of times the user-supplied error Callback routine has been called.
mqlExecute	Sends an MQL command to the MQL process or the Collaboration Server session for execution.
mqlInit	Resets all static string variables for the host, user ID, password, and vault to blank values.

Function	Description
mqlInput	Used by mqlCommand to send the contents of the input buffer to MQL or the Collaboration Server.
mqlOpen	Starts a session and establishes communications with MQL or the Collaboration Server. eMatrixMQL uses HTTP/XML.
mqlOpenLog	Opens a log file filename for the current session. It reads the input and output buffers and keeps a record of input to MQL or the Collaboration Server and output from MQL or the Collaboration Server.
mqlOutput	Used by mqlCallback to receive data from MQL or the Collaboration Server.
mqlOutputLine	Used by mqlCallback to receive a line of data from MQL or the Collaboration Server. It breaks on new line.
mqlOutputs	Maintains a counter of the number of times the user-supplied outputCallback routine has been called, by recording the number of outputs from mqlCallback.
mqlPending	Checks for any output data that is waiting to be processed. It is useful for commands that generate a large amount of output, because it begins to process the output data as soon as it is available. mqlCallback is the better function to use.
mqlSetContext	Sets the context with User, Password and Vault parameters. It is the first eMatrixMQL function your program calls. An additional parameter is required for the Collaboration Server host name.
mqlSetQuote	Turns MQL quoting on.
mqlTimeout	Sets the sleep function: sleeps for x seconds.
mqlWriteToLogFile	Writes data to the log file.

Synopsis of eMatrixMQL Functions

The tables that follow provide syntax and additional notes on eMatrixMQL functions.

Function	**mqlOpen**
Synopsis	`int mqlOpen(void);`
Description	This function starts a session and establishes communications with MQL or the Collaboration Server. eMatrixMQL uses HTTP/XML.
Returns	**MQLOK** means that the session has been established. **MQLERROR** means that session has not been established.

Function	**mqlClose**
Synopsis	`int mqlClose(void);`
Description	This function ends an open session by terminating the MQL process.
Returns	**MQLOK** means the session was shutdown successfully. **MQLERROR** means the session shutdown encountered a problem.
Note	Once you have opened a session using mqlOpen, you must call mqlClose to end the session before exiting your program.

.

Function	**mqlnit**
Synopsis	`void mqlInit(void);`
Description	Resets all static string variables for the host, user ID, password, and vault to blank values.
Returns	void

Function	**mqlSetContext**
Synopsis	`void mqlSetContext(MQLBOS_STR_PTR host, MQLBOS_STR_PTR user, MQLBOS_STR_PTR password, MQLBOS_STR_PTR vault);`
Description	Sets the context variables with the User, Password and Vault parameters. It is the first eMatrixMQL function your program calls. The additional parameter, MQLBOS_STR_PTR host, provides the Live Collaboration Server host name.
Returns	void

Function	**mqlOpenLog**
Synopsis	`int mqlOpenLog(char * filename);` `char * filename` is a pointer to a character string containing the path and name of the log file to be opened.
Description	This function opens a log file *(filename)* for the current session. It reads the input and output buffers and keeps a record of input to MQL and the Collaboration Server and output from MQL and the Collaboration Server.
Returns	**MQLOK** means the log file was opened successfully. **MQLERROR** means the log file could not be opened.

Function	**mqlCloseLog**
Synopsis	`int mqlCloseLog(void);`
Description	This function closes the log file opened using mqlOpenLog.
Returns	**MQLOK** means the log file was closed successfully. **MQLERROR** means the log file was not properly closed.

Function	**mqlPending**
Synopsis	`int mqlPending (void);`
Description	This function checks for any output data that is waiting to be processed. It is useful for commands that generate a large amount of output, because it begins to process the output data as soon as it is available. mqlCallback is the better function to use.
Returns	**MQLERRORPENDING (2)** means that error output is pending. **MQLOUTPUTPENDING (1)** means output is pending. **MQLOK (0)** means that no output is pending. **MQLERROR (-1)** means that a time-out has occurred.

Function	**mqlTimeout**
Synopsis	`int mqlTimeout (int seconds);`
Description	Sets the sleep function: sleeps for `int` seconds.
Returns	**MQLOK** means the timeout was set successfully.

Function	**mqlCallback**

Synopsis	`int mqlCallback (int (*outputCallback) (void * outData),`
	`void * outputData, int (*errorCallback) (void * errData),`
	`void * errorData);`
	`int (*outputCallback) (void * outData)` is a pointer to a user-supplied output handling function that returns an integer and has outData as a parameter.
	`void * outputData` is an output data storage location passed to mqlCallback from the output handling function.
	`int (*errorCallback) (void * errData)` is a pointer to a user-supplied error handling function that returns an integer and has errData as a parameter.
	`void * errorData` is the error data storage location passed to mqlCallback from the error handling function.

Description	This function is a setup routine to handle output and errors. The user passes in a pointer to a function that returns an integer and a storage location for any results. This is done for both normal and error output.
Returns	**MQLOK** means that callback completed successfully. **MQLERROR** means that callback failed.
Note	The return values for mqlCallback do not reflect the status of the MQL command for which output is being processed; they reflect the execution status of the mqlCallback function itself.

Function	**mqlOutputs**
Synopsis	`int mqlOutputs (void);`
Description	This function maintains a counter of the number of times the user-supplied outputCallback routine has been called, by recording the number of outputs from mqlCallback.
Returns	The number of times that mqlCallback called the outputCallback routine.
Note	The counter is reset each time mqlCallback is called.

Function	**mqlErrors**
Synopsis	`int mqlErrors (void);`
Description	This function maintains a counter of the number of times the user-supplied errorCallback routine has been called.
Returns	The numbers of times that mqlCallback called the error callback routine.
Note	The counter is reset each time mqlCallback is called.

Function	**mqlInput**
Synopsis	`int mqlInput (char * buffer, int size);`
Description	This function is used by mqlCommand to send the contents of the input buffer to MQL or the Collaboration Server. `char * buffer` is the input buffer. `int size` is a buffer size.
Returns	The size of the buffer written or MQLERROR for failure.

Function	**mqlOutput**
Synopsis	`int mqlOutput(char * buffer, int limit);`
Description	This function is used by mqlCallback to receive data from MQL or the Collaboration Server. `char * buffer` is a pointer to the output buffer. `int limit` is the buffer size limit.
Returns	The size of the buffer received or MQLERROR on failure.

Function	**mqlOutputLine**
Synopsis	`char * mqlOutputLine(char * buffer,int limit);`
Description	This function is used by mqlCallback to receive a line of data from MQL or the Collaboration Server. It breaks on new line. `char * buffer` is a pointer to the output buffer. `int limit` is the buffer size limit.
Returns	A line of data or NULL if nothing could be output.

Function	mqlError
Synopsis	`int mqlError(char * buffer,int limit);`
Description	This function is used by mqlCallback to receive error data from MQL or the Collaboration Server. `char * buffer` is a pointer to the error output buffer. `int limit` is the buffer size limit.
Returns	Number of bytes successfully read or MQLERROR on failure.

Function	**mqlErrorLine**
Synopsis	`char * mqlErrorLine(char * buffer,int limit);`
Description	This function is used by mqlCallback to receive a line of error data from MQL or the Collaboration Server. It breaks on new line. `char * buffer` is a pointer to the error output buffer. `int limit` is a buffer size limit.
Returns	A line of data or NULL if nothing could be read.

Function	**mqlErrorMessage**
Synopsis	`void mqlErrorMessage(MQLBOS_STR_FAR_PTR message, const MQLBOS_STR_FAR_PTR segment);`
Description	Prints formatted error message to stdout.
Returns	void

Function	**mqlCommand**
Synopsis	`int mqlCommand(MQLBOS_STR_FAR_PTR format, ...);`
Description	This function writes an MQL command to MQL or the Collaboration Server using printf style syntax.
Returns	**MQLOK** means that the sent command(s) was successfully invoked. **MQLERROR** means that the sent command(s) could not be invoked.
Note	Do not put a semi-colon (;) at the end of the format string. The mqlCommand function appends a semi-colon automatically. The extra semi-colon will cause mqlCallback to be invoked a second time, resetting the mqlErrors and mqlOutputs counters. When using mqlCommand without a callback function, output should be checked with the mqlPending command

Function	**mqlExecute**
Synopsis	`int mqlExecute(MQLBOS_STR_FAR_PTR format);`
Description	Sends an MQL command to the MQL or the Collaboration Server for execution.
Returns	**MQLOK** means that the command executed successfully. **MQLERROR** means that the command execution failed.

Function	**mqlDisableLog**
Synopsis	`int mqlDisableLog(void);`
Description	This function turns off the log function started with `mqlOpenLog`. Use this to prevent your log file from becoming too large, for example, if you are listing 50,000 objects.
Returns	0 indicates that the log function has been turned off. -1 indicates that the log function has not been turned off.

Function	**mqlEnableLog**
Synopsis	`int mqlEnableLog(void);`
Description	This function turns on the log function again, after a mqlDisableLog function has been called.
Returns	0 indicates that the log function has been turned on. -1 indicates that the log function has not been turned on.

Function	**mqlWriteToLogFile**
Synopsis	`int mqlWriteToLogFile(int dataType, MQLBOS_STR_FAR_PTR buffer, int size);`
Description	This function writes data (i.e., errors) to the log file.
Returns	**MQLOK** means that data was successfully written to the log file. **MQLERROR** means that data was not written to the log file.

Terminating a Session

A session is terminated using the `mqlClose` command. `mqlClose` issues a `quit` command to the MQL process or server session started with `mqlOpen`. Any open log files should also be closed using `mqlCloseLog`.

```
    if (mqlCloseLog() == MQLERROR)
    {
       fprintf(stderr, “ERROR: Can not Close MQL Log File\n”);
       exit(-1);
    }
```

```
    if (mqlClose() == MQLERROR)
    {
       fprintf(stderr, “ERROR: Can not Close MQL Interface\n”);
       exit(-1);
    }
```

`mqlClose` and `mqlCloseLog` have the same possible returns:

- **MQLOK (0)** The session or log file was closed properly.
- **MQLERROR (-1)** The session or log file encountered a problem when attempting to close.

See also *mqlClose* and *mqlCloseLog*.

Conversational Strategies

There are many different strategies a programmer can use to retrieve data from MQL or the server. For example, a list of character pointers can be allocated to hold the data from a command and expand as needed.

```
/**************************************************/
#define ALLOCATIONBLOCK 64
/* List. */
typedef struct stringList stringList;

struct stringList
{
   int size;
   int allocated;
   int iter;
   char **base;
};
/**************************************************/
stringList * createStringList(void)

/* Function to create string list.
** input: none.
** output: none.
** returns: stringList
*/
{
   stringList *list = NULL;
   char **base = NULL;
   if (base = (char **)malloc(ALLOCATIONBLOCK*sizeof(char *)))
   {
      if (list = (stringList *)malloc(sizeof(stringList)))
      {
         list->size = 0;
         list->allocated = ALLOCATIONBLOCK;
         list->iter = -1;
         list->base = base;
      }
      else
      {
         if (base) free((void *)base);
      }
   }
   return list;
}
Functions to append to the stringList can also be added.
/****************************************************/

int appendStringList(stringList * list,char * string)

/* Function to append item to string list.
** input: list.....list pointer.
** input: string.....string pointer.
** output: none.
** returns: 0......item was successfully appended.
** returns: -1.....item could not be appended.
```

```
*/

{

   int item;
   char **newbase = NULL;

      if (list && list->base && string)
   {
      if (list->size >= list->allocated)
      {
         if (newbase = (char **)malloc((list->allocated
+ALLOCATIONBLOCK)*sizeof(char *)))
         {
            memcpy((char *)newbase,(char *)list->base, list-
allocated*sizeof(char *));
            free((void *)list->base);
            list->base = newbase;
            list->allocated += ALLOCATIONBLOCK;
         }
      }
      item = list->size;

      if(list->base[item]=(char *)malloc(strlen(string)+1))
      {
         strcpy(list->base[item],string);
         list->size++;
         return 0;
      }
   }
   return -1;
}

/*****************************************************/
An output function for use with mqlCommand would look like this:
/* stringListOutputCallback.cpp -- function to handle string list output
** callback. */

/********************************************************/

#include <string.h>
#include "stringListOutputCallback.h"
#include "stringList.h"
#include "eMatrixMQL.h"

/********************************************************/

#define LINEBUFFERSIZE 8192

/********************************************************/

int stringListOutputCallback(void * list)

/* Handle string list output callback.
** input: list......string list.
** output: none.
** returns: MQLOK.........output was successfully handled.
** returns: MQLERROR.......output could not be handled.
*/
```

```
{

   char buffer[LINEBUFFERSIZE];                  /* Buffer */
   char *eol;          /* End of line */

   if (mqlOutputLine(buffer,LINEBUFFERSIZE))
   {
      if (eol = strchr(buffer,' \n'))
      *eol = '\0';
         if (strlen(buffer))
            appendStringList((stringList *)list,buffer);
   }
   return MQLOK;

}
```

Modular Packaging

The use of MQL commands through the eMatrixMQL interface leads to easy modularization of functions. Any MQL sequence can be placed in a *wrapper* function for easy re-use. It is recommended that the `mqlClose` function be placed in a function that would be used in place of `exit()`.

```
/**************************************************/
void ExitApplication( int status )
{

   if (mqlCloseLog() == MQLERROR)
   {
      fprintf(stderr,"ERROR: Can Not Close Log File \n");
      exit(status);
   }
   if (mqlClose() == MQLERROR)
   {
      fprintf(stderr,"ERROR: Can Not Close MQL Interface\n");
      exit(status);
   }

   exit(status);
}

In the application, an error condition could then be handled easily.
if (strcmp(objectType, "Assembly") != 0 )
{
   fprintf(stderr, "Can't find an Assembly!\n");
   ExitApplication( NO_ASSEMBLY );

}
```

Another common MQL usage is to test the existence of an object.

```
/* testObject.cpp -- function to test if an eMatrix object exists.*/

/*******************************************************/
#include <stdio.h>
#include "testObject.h"
#include "eMatrixMQL.h"
/*******************************************************/
#define SUCCESS (0)
#define FAILURE (1)
#define ERROR (-1)
#define LINEBUFFERSIZE 8192
int ignoreOutputCallback(void * outData);
int ignoreErrorCallback(void * errData);
/*******************************************************/

int testObject(char * type, char * name, char * revision, char * user)

/* Test for object in Matrix.
** input: type.........object type.
** input: name.........object name.
** input: revision.....object revision.
** output: none.
** returns: SUCCESS.......object already exists.
** returns: FAILURE.......object does not exist.
** returns: ERROR.........error.
*/
{
   if (mqlCommand("print bus \'%s\' \'%s\' \'%s\' select name dump",  type,
name, revision)
         ==  MQLERROR)
   {
      return ERROR;
   }
   if (mqlCallback( ignoreOutputCallback, NULL,  ignoreErrorCallback, NULL)
      == MQLERROR )
   {
      return ERROR;
   }
   /* If mqlErrors returns zero (0) the object exists return success. */
   if (mqlErrors() == 0)
   {
      return SUCCESS;
   }
   else
   {
      return FAILURE;
   }
}
/********************************************************/
int ignoreOutputCallback(void * outData)

/* Function to handle output callback.
** input: none.
** output: none.
** returns: MQLOK..output callback was handled.
** returns: MQLERROR..output callback could not be handled.
*/

{
   char buffer[LINEBUFFERSIZE];                   /* Buffer */

   if (mqlOutputLine(buffer, LINEBUFFERSIZE))
   {
       /* ignore any output */
   }

   return MQLOK;
}
/********************************************************/

int ignoreErrorCallback(void * errData)

/* Function to handle error callback.
** input: none.
** output: none.
** returns: MQLOK..error callback was handled.
** returns: MQLERROR..error callback could not be handled.
*/

{
   char buffer[LINEBUFFERSIZE];                   /* Buffer */

   if (mqlErrorLine(buffer, LINEBUFFERSIZE))
   {
      /* ignore any error output */
   }

   return MQLOK;

}
The function could be used as follows:
int status;

status = testObject("Assembly", "AAAA", "1", "Fred");

switch (status)
{
   case FAILURE :
      createObject("Assembly", "AAAA", "1", "Fred");
      break;

   case SUCCESS :
      fprintf( stderr, "Object: %s %s %s exists\n",
         "Assembly", "AAAA", "1", "Fred");
      break;

   case ERROR :
      fprintf( stderr, "eMatrixMQL Failure\n" );
      break;

   default :
      fprintf( stderr, "Unknown Return Code %d\n", status );
      break;
}
```

Working with a DLL

Using a DLL is a convenient method of accessing MQL from another application, since you can perform runtime linking with your C or C++ program, or call it externally from another program:

DLL	Format	Use With	Protocol
eMatrixMQL.dll	64-bit	eMatrixMQL	HTTP/XML

Using DLLs with C or C^{++}

When using eMatrixMQL.dll with a C or C^{++} program, you can use the `LoadLibrary()` function to access any of the exported functions, or link in the DLL Stub Library (recommended) as long as the appropriate DLL is in the same path as the executable for your program. When statically linking, eMatrixMQL.h must be *included*, and the .lib files must appear in the project.

The include file for eMatrixMQL is eMatrixMQL.h.

Compiling and Linking

Compiling a Custom Application

To Compile Using Visual C++

1. Create a new project as “Win32 Console Application” in Visual studio.
2. Add the custom *.cpp source code file(s)
3. Link with eMatrixMql.lib. Under Project Settings -> Link tab, add the library to the end of the line “Object/library modules”.
4. Add the directory where your header files are located under “preprocessor, additional include directories”.
5. Under Project Settings -> C/C++ tab, set **Category** drop down list to **Preprocessor**, and add the path to the header file(s)
6. Make sure the eMatrixMql.dll is available to the compiler, which might mean to place it in the default directory

Distributing a Custom Application

To run your test application from the client machine

1. Be sure that the full JRE (or JDK) distribution is installed, and that the system environment PATH variable points to the location of the Java binaries, for example:
   ```
   Set PATH=%PATH%;\JAVA_HOME\jre\bin\server
   ```
2. Be sure that all Components needed are on the client computer:
   - the program you made (e.g., "test.exe")
   - eMatrixMQL.dll: in the same directory as the above program, or in another location (but be sure to set your PATH environment variable to point to it)
   - mxUtil.dll, vgalaxy7u.dll, and vgalaxy7.vr: copy from ENOVIA_INSTALL\PLATFORM\code\bin\ to the same directory, or to the other location (but be sure to set your PATH environment variable to point to it).
   - eMatrixServletRMI.jar: put this anywhere, and be sure to define your CLASSPATH var to point to it, including full path and filename, (set CLASSPATH=/programs/eMatrixServletRMI.jar)

Compiling and Linking with UNIX

Makefiles will vary slightly on each UNIX platform. The following is an example Makefile for a Red Hat workstation:

```
##########################################################

CC = cc

ADK_INSTALL = /usr/DassaultSystemes/enoviaV6R2013/adk
ENOVIA_INSTALL = /usr/DassaultSystemes/enoviaV6R2013/server
JAVA_HOME = /usr/java/jdk1.6.0_24

CFLAGS = -fPIC –D_POSIX_ -D_LINUX_SOURCE –I. –I$(ADK_INSTALL)/
samples/eMatrixMql
```

```
LDFLAGS = -L$(ADK_INSTALL)/linux_a64/code/bin
-L$(ENOVIA_INSTALL)/linux_a64/code/bin -L$(JAVA_HOME)/jre/lib/
amd64/server

LIBS = -leMatrixMql -ljvm -lmxUtil -lvgalaxy7u

SRCS = example1.cpp \
       testObject.cpp \
       errorCallback.cpp \
       ignoreErrorCallback.cpp \
       ignoreOutputCallback.cpp

OBJS = example1.o \
       testObject.o\
       errorCallback.o \
       ignoreErrorCallback.o\
       ignoreOutputCallback.o

all: example1

example1: $(OBJS)
       cc -o example1 $(LDFLAGS) $(CFLAGS) $(OBJS) $(LIBS)

clean:
       rm -f $(OBJS) example1

#########################################################
```

Note: The library supplied for the eMatrixMQL interface is in object format (.o). Using the -l option will NOT work. Use the full path name instead.

Compiling and Linking on Windows

When using the eMatrixMQL.dll, you can link any of the eMatrixMQL functions using dllimport, as shown below:

```
#include <windows.h>

#ifdef MatrixMQL
__declspec (dllimport) int mqlSetContext(char *, char *, char *,
char *);
#endif

__declspec (dllimport) int mqlOpen(void);
__declspec (dllimport) int mqlClose(void);
__declspec (dllimport) int mqlExecute(char *);
__declspec (dllimport) int mqlPending(void);
__declspec (dllimport) int mqlOutput(char *,int);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrev
Instance, LPSTR
lpszCmdLine, int nCmdShow)
{
   int nResult, status;
   char * retbuf = "";
   char * host = "WebServer";
   char * userID = "JSmith";
   char * password = "xyzzy";
   char * vault = "Engineering";

#ifdef EMATRIXMQL
mqlSetContext(host, userID, password, vault);
#endif

   nResult = mqlOpen();
   mqlExecute("set context user creator");
   status = mqlPending();
   mqlExecute("list vault");
   mqlOutput (retbuf, 1000);
   MessageBox (NULL, retbuf, "Title", MB_OK);
   mqlClose();

   return 1;
}
```

Sample Code

Sample code for eMatrixMQL is provided on the distributed CDs. The filenames and locations are listed below.

eMatrixMQLTest Sample Code

The eMatrixMQLTest sample code can be found on the Studio Customization Toolkit distributed CD. The following files are included:

- eMatrixMQLTest.cpp
- eMatrixMQLTest.h

Example makefile for Studio Customization Toolkit

The following is an example using C++ Studio Customization Toolkit classes to create a command line interface to access the database via the collaboration server.

This example (eMatrixMqlTest.cpp) is for the Red Hat Enterprise Linux platform.

```
#
# Makefile.gnu - example makefile for eMatrixMqlTest example
(compatible with GNU make)
#
# Usage : make -f Makefile.gnu
#

ADK_INSTALL = /home/ds-hchang/enovia/R213/adk
ENOVIA_INSTALL = /usr/DassaultSystemes/enoviaV6R2013/server
JAVA_HOME = /usr/java/jdk1.6.0_24

EXE       = eMatrixMqlTest
CXX       = cc
CXXFLAGS = -fPIC -D_POSIX_ -D_LINUX_SOURCE -I. -I$(ADK_INSTALL)/
samples/eMatrixMql
LDFLAGS = -L$(ADK_INSTALL)/linux_a64/code/bin
-L$(ENOVIA_INSTALL)/linux_a64/code/bin -L$(JAVA_HOME)/jre/lib/
amd64/server
LIBS     = -leMatrixMql -ljvm -lmxUtil -lvgalaxy7u

all: $(EXE)

$(EXE):
   $(CXX) $(CXXFLAGS) $(LDFLAGS) $(LIBS) $(EXE).cpp -o $(EXE)
```

Index