A Ruby extension for InterSystems M/Caché/IRIS and YottaDB

mg_ruby

M/Gateway Developments Ltd.
Chris Munt

Revision History:

23 January 2020
20 January 2021
16 February 2021
14 March 2021

Contents

1 Introduction

mg_ruby is an Open Source Ruby extension providing direct access to InterSystems **Caché**, **IRIS** and the **YottaDB** database. It will also work with other M-like databases.

Although this document refers to ***InterSystems Caché*** throughout, it can be assumed that the same applies to ***InterSystems IRIS***.

2 Pre-requisites

It is assumed that you have the following three components already installed:

- Ruby http://www.ruby-lang.org

Either:

- InterSystems Caché or IRIS http://www.intersystems.com

Or:

- YottaDB https://www.yottadb.com
- A Web Server (if Ruby is to be used for developing web applications).

3 Installing mg_ruby

There are three parts to **mg_ruby** installation and configuration.

- The Ruby extension (**mg_ruby.so**).
- The database (or server) side code: **zmgsi** (*The DB Superserver*).
- A network configuration to bind the former two elements together.

The *DB Superserver* are required for:

- Network based access to databases.

Two M routines need to be installed **(%zmgsi** and **%zmgsis**). These can be found in the *Service Integration Gateway* (**mgsi**) GitHub source code repository.

https://github.com/chrisemunt/mgsi

Note that it is not necessary to install the whole *Service Integration Gateway (SIG)*, just the two M routines held in that repository, unless of course you intend to connect **mg_ruby** to the database via the SIG

3.1 InterSystems Caché or IRIS

Log in to the %SYS Namespace and install the **zmgsi** routines held in **/isc/zmgsi_isc.ro**.

```
do $system.OBJ.Load("/isc/zmgsi_isc.ro","ck")
```

Change to your development Namespace and check the installation:

```
do ^%zmgsi

M/Gateway Developments Ltd - Service Integration Gateway
Version: 4.0; Revision 16 (11 February 2021)
```

3.2 YottaDB

The instructions given here assume a standard 'out of the box' installation of **YottaDB** (version 1.30) deployed in the following location:

```
/usr/local/lib/yottadb/r130
```

The primary default location for routines:

```
/root/.yottadb/r1.30_x86_64/r
```

Copy all the routines (i.e. all files with an 'm' extension) held in the GitHub **/yottadb** directory to:

```
/root/.yottadb/r1.30_x86_64/r
```

Change directory to the following location and start a **YottaDB** command shell:

```
cd /usr/local/lib/yottadb/r130
./ydb
```

Check the installation:

```
do ^%zmgsi

M/Gateway Developments Ltd - Service Integration Gateway
Version: 4.0; Revision 16 (11 February 2021)
```

Note that the version of **zmgsi** is successfully displayed.

3.3 Setting up the network service: the DB Superserver

The default TCP server port for the DB Superserver (**zmgsi)** is **7041**. If you wish to use an alternative port then modify the following instructions accordingly. Ruby code using the **mg_ruby** functions will, by default, expect the database server to be listening on port **7041** of the local server (localhost). However, **mg_ruby** provides the functionality to modify these default settings at run-time. It is not necessary for the web Ruby installation to reside on the same host as the database server.

For InterSystems DB Servers, the DB Superserver is started from the DB Server command prompt. For YottaDB the DB Superserver can either be started from the DB Server command prompt or managed by the ***xinetd*** service.

3.3.1 Starting the DB Superserver from the DB Server command prompt

For InterSystems DB Servers the DB Superserver should be started in the %SYS Namespace.

```
do start^%zmgsi(0)
```

To use a server TCP port other than 7041, specify it in the start-up command (as opposed to using zero to indicate the default port of 7041).

3.3.2 Starting YottaDB Superserver processes via xinetd

Instead of starting the DB Supersever from the DB Server command prompt, network connectivity to **YottaDB** can be managed via the **xinetd** service. First create the following launch script (called **zmgsi_ydb** here):

```
/usr/local/lib/yottadb/r130/zmgsi_ydb
```

Content:

```
#!/bin/bash
cd /usr/local/lib/yottadb/r130
export ydb_dir=/root/.yottadb
export ydb_dist=/usr/local/lib/yottadb/r130
export
ydb_routines="/root/.yottadb/r1.30_x86_64/o*(/root/.yottadb/r1.30_x86_6
4/r /root/.yottadb/r) /usr/local/lib/yottadb/r130/libyottadbutil.so"
export ydb_gbldir="/root/.yottadb/r1.30_x86_64/g/yottadb.gld"
$ydb_dist/ydb -r xinetd^%zmgsis
```

Note that you should, if necessary, modify the permissions on this file so that it is executable. For example:

```
chmod a=rx /usr/local/lib/yottadb/r130/zmgsi_ydb
```

Create the **xinetd** script (called **zmgsi_xinetd** here):

```
/etc/xinetd.d/zmgsi_xinetd
```

Content:

```
service zmgsi_xinetd
{
     disable         = no
     type            = UNLISTED
     port            = 7041
     socket_type     = stream
     wait            = no
     user            = root
     server          = /usr/local/lib/yottadb/r130/zmgsi_ydb
}
```

- Note: sample copies of **zmgsi_xinetd** and **zmgsi_ydb** are included in the /unix directory of the **mgsi** GitHub directory (https://github.com/chrisemunt/mgsi).

Edit the services file:

```
/etc/services
```

Add the following line to this file:

```
zmgsi_xinetd          7041/tcp                          # ZMGSI
```

Finally restart the **xinetd** service:

```
/etc/init.d/xinetd restart
```

3.4 Installing and using the mg_ruby component

Either build the Ruby component (**mg_ruby.so**) from source (/src/mg_ruby.c) or use a pre-built module from the distribution (if available for your platform).

Windows:

The pre-built **mg_ruby.so** module should be copied to the appropriate location in the Ruby file system. For example, using an 'out of the box' Ruby 2.7 installation this will be:

```
C:\Ruby27-x64\lib\ruby\site_ruby\2.7.0\x64-msvcrt
```

Having done this, **mg_ruby** is ready for use.

3.4.1 Building the source code

mg_ruby is written in standard C. For Linux systems, the Ruby installation procedure can use the freely available GNU C compiler (gcc) which can be installed as follows.

Ubuntu:

```
apt-get install gcc
```

Red Hat and CentOS:

```
yum install gcc
```

Apple OS X can use the freely available **Xcode** development environment.

Under Windows, Ruby is built with the Open Source **MSYS2** Development kit and if you plan to build **mg_ruby** from the supplied source code it is recommended that you select the pre-built '**Ruby+Devkit**' option when downloading Ruby for Windows. This package will install Ruby together with all the tools needed to build **mg_ruby**

The Ruby Extension installer can be used to build and deploy **mg_ruby**. You will find the setup scripts in the /src directory of the distribution.

UNIX and Windows using the MSYS2 Development Toolkit (most installations):

The commands listed below are run from a command shell. For Windows, the MSYS2 command shell provided with the **'Ruby+Devkit'** distribution should be used. For example, with the 'out of the box' Ruby 2.7 installation this will be found at: `C:\Ruby27-x64\msys64\msys2`.

```
ruby extconf.rb
make
make install
```

Windows using the Microsoft Development Toolkit:

```
ruby extconf.rb
nmake
nmake install
```

3.4.2 Using mg_ruby

Ruby programs may refer to, and load, the **mg_ruby** module using the following directive at the top of the script.

```
require 'mg_ruby'
mg_ruby = MG_RUBY.new()
```

Having added this line, all methods listed provided by the module can be invoked using the following syntax.

```
mg_ruby.<method>
```

It is not necessary to name your instance as 'mg_ruby'. For example, you can have::

```
require 'mg_ruby'
<name> = MG_RUBY.new()
```

Then methods can be invoked as:

```
<name>.<method>
```

For web development, the distribution also contains the following XMLHTTP script file:

```
/usr/mgwsi/java/mg_client.js
```

Copy the XMLHTTP file into your applications root directory. For example:

```
C:\Inetpub\wwwroot\
```

4 Ruby Arrays and M Globals

The M database is made up of any number of *global variables*. Each global variable (or *global*) roughly equates to a table in a relational database. The data within each global is organized as a B-Tree structure. The key to each *global node* (or *record*) can be divided up into any number of individual sub-key items (or *subscripts*). The ability to divide-up the key to a global node in this manner gives the M database its *multidimensional* characteristics.

This section will describe the conventions used within **mg_ruby** to map M globals on to simple numerically indexed Ruby Arrays. Two data constructs are used to represent M data.

1. The key (or subscripts) to a global node are held in simple numerically indexed Ruby Arrays. By convention, position zero in the key array contains the number of subscripts.

2. Sets of global nodes (or records) are held in Ruby Arrays. With Arrays, the *size()* method can be used to determine the number of records held. Within individual array records, the M global subscripts and data are concatenated together (in that order) to create a compound data record containing all the individual parts. The **mg_ruby** module provides methods to create and manage these data constructs. The *m_local* suite of methods are used to maintain locally held sets of M records.

5 Method Reference for mg_ruby

The methods provided by the core **mg_ruby** component allow you to directly manipulate the M database from within the Ruby scripting environment. Methods are also supplied to allow you to directly call M extrinsic functions (and methods) and M procedures that are capable of generating sections of HTML form data.

All **mg_ruby** methods are implemented according to the following patterns.

- Methods working to local data (ma_local_* methods): The first argument is always the name of a records array. Further arguments represent subscripts and data for a record.

- Methods working directly to M: The first argument is always the name of either a M global or a function/procedure depending on the nature of the operation. Further arguments are either passed as subscripts to a global or arguments to a M function (or method), depending on the context.

5.1 Connecting to the database

By default, and assuming TCP based connectivity is used, the **mg_ruby** methods will address M host '**localhost'** listening on TCP port **7041**. The following methods allow you to redefine the default host, and to address hosts other than the default.

5.1.1 Modifying the default M Host

```
mg_ruby.m_set_host(netname, port, username, password)
```

In addition to addressing the default host, other M servers can be specified on a per-call basis as shown in this document's many examples. You can specify the M host to be used within an instance of the **mg_ruby** module using the '**m_set_host'** method.

For example:

```
mg_ruby.m_set_host("MyCachéServer", 7041, "", "")
```

All **mg_ruby** method calls in the page will now target M server: `MyCachéServer`.

Scope

The name of the host is scoped in accordance with the instance of the **mg_ruby** module in use.

5.1.2 Modifying the default M NameSpace

```
mg_ruby.m_set_uci(uci)
```

This method will change the Namespace associated with the current **mg_ruby** instance.

For example:

```
mg_ruby.m_set_uci("USER")
```

5.1.3 Modifying the DB Server response timeout

```
mg_ruby.m_set_timeout(timeout)
```

This method will change the DB Server response timeout associated with the current server handle. The timeout should be specified in seconds.

For example (set the timeout to 60 seconds):

```
result = mg_ruby.m_set_timeout(60)
```

5.1.4 Non-network based access to the database via its API

As an alternative to connecting to the database using TCP based connectivity, **mg_ruby** provides the option of high-performance embedded access to a local installation of the database via its API.

5.1.4.1 InterSystems Caché or IRIS.

Use the following functions to bind to the database API.

```
mg_ruby.m_set_uci(namespace)
mg_ruby.m_bind_server_api(dbtype, path, username, password,
                          envvars, params)
```

Where:

```
namespace:  Namespace.
dbtype:     Database type ('Cache' or 'IRIS').
path:       Path to database manager directory.
username:   Database username.
password:   Database password.
envvars:    List of required environment variables.
params:     Reserved for future use.
```

Example:

```
mg_ruby.m_set_uci("USER")
result = mg_ruby.m_bind_server_api("IRIS", "/usr/iris20191/mgr",
                                   "_SYSTEM", "SYS", "", "")
```

The bind function will return '1' for success and '0' for failure.

Before leaving your Ruby application, it is good practice to gracefully release the binding to the database:

```
mg_ruby.m_release_server_api()
```

Example:

```
mg_ruby.m_release_server_api()
```

5.1.4.2 YottaDB

Use the following function to bind to the database API.

```
mg_ruby.m_bind_server_api(dbtype, path, username, password,
                          envvars, params)
```

Where:

```
dbtype:     Database type ('YottaDB').
path:       Path to the YottaDB installation/library.
username:   Database username.
password:   Database password.
envvars:    List of required environment variables.
params:     Reserved for future use.
```

Example:

This assumes that the YottaDB installation is in: `/usr/local/lib/yottadb/r130`
This is where the **libyottadb.so** library is found.
Also, in this directory, as indicated in the environment variables, the YottaDB routine interface file resides (zmgsi.ci in this example). The interface file must contain the following lines:

```
sqlemg: ydb_string_t * sqlemg^%zmgsis(I:ydb_string_t*, I:ydb_string_t *,
I:ydb_string_t *)
sqlrow: ydb_string_t * sqlrow^%zmgsis(I:ydb_string_t*, I:ydb_string_t *,
I:ydb_string_t *)
sqldel: ydb_string_t * sqldel^%zmgsis(I:ydb_string_t*, I:ydb_string_t *)
ifc_zmgsis: ydb_string_t * ifc^%zmgsis(I:ydb_string_t*, I:ydb_string_t *,
I:ydb_string_t*)
```

Moving on to the Ruby code for binding to the YottaDB database. Modify the values of these environment variables in accordance with your own YottaDB installation. Note that each line is terminated with a linefeed character, with a double linefeed at the end of the list.

```
envvars = "";
envvars = envvars + "ydb_dir=/root/.yottadb\n"
envvars = envvars + "ydb_rel=r1.30_x86_64\n"
envvars = envvars + "ydb_gbldir=/root/.yottadb/r1.30_x86_64/g/yottadb.gld\n"
envvars =envvars +
"ydb_routines=/root/.yottadb/r1.30_x86_64/o*(/root/.yottadb/r1.30_x86_64/r
/root/.yottadb/r) /usr/local/lib/yottadb/r130/libyottadbutil.so\n"
envvars = envvars + "ydb_ci=/usr/local/lib/yottadb/r130/zmgsi.ci\n"
envvars = envvars + "\n"


result = mg_ruby.m_bind_server_api("YottaDB",
                                   "/usr/local/lib/yottadb/r130",
                                   "", "", envvars, "")
```

The bind function will return '1' for success and '0' for failure.

Before leaving your Ruby application, it is good practice to gracefully release the binding to the database:

```
mg_ruby.m_release_server_api()
```

Example:

```
mg_ruby.m_release_server_api()
```

5.2 Direct access to the M database

The **mg_ruby** methods in this section give you direct access to the M commands for manipulating global data. A M global is essentially a multi-dimensional associative array that's held in permanent storage.

There are two forms for each of these methods: those that express a global reference as a variable number of input arguments are prefixed by 'm_' and those for which the keys are expressed as a Ruby list are prefixed by 'ma_'.

5.2.1 Set a global node.

```
result = mg_ruby.m_set(<global>, <keys…>, <data>)
result = mg_ruby.ma_set(<global>, <keylist>, <data>)
```

Types:

```
result            (String)
global            (String)
```

```
keys              (Any)
keylist           (List)
data              (String)
```

By convention, the last argument is always the global node's data record.

Example 1:

M command:

```
Set ^Customer(1234)="Chris Munt"
```

Equivalent **mg_ruby** methods:

```
mg_ruby.m_set("^Customer", 1234, "Chris Munt")

Or:

key = [1, "1234"]
mg_ruby.ma_set("^Customer", key, "Chris Munt")
```

5.2.2 Retrieve the data from a global node.

```
data = mg_ruby.m_get(<global>, <keys…>)
data = mg_ruby.ma_get(<global>, <keylist>)
```

Types:

```
data              (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M command:

```
Set data=$Get(^Customer(1234))
```

Equivalent **mg_ruby** methods:

```
data = mg_ruby.m_get("^Customer", 1234)

Or:

key = [1, "1234"]
data = mg_ruby.ma_get("^Customer", key)
```

5.2.3 Check that a global node exists.

```
defined = mg_ruby.m_defined(<global>, <keys…>)
defined = mg_ruby.ma_defined(<global>, <keylist>)
```

Types:

```
defined           (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M command:

```
Set defined=$Data(^Customer(1234))
```

Equivalent **mg_ruby** methods:

```
defined = mg_ruby.m_defined"^Customer", 1234)
```

Or:

```
key = [1, "1234"]
defined = mg_ruby.ma_defined("^Customer", key)
```

5.2.4 Delete a global node.

```
result = mg_ruby.m_delete(<global>, <keys…>)
result = mg_ruby.ma_delete(<global>, <keylist>)
```

Types:

```
result            (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M command:

```
Kill ^Customer(1234))
```

Equivalent **mg_ruby** method:

```
mg_ruby.m_delete("^Customer", 1234)
```

Or:

```
key = [1, "1234"]
mg_ruby.ma_delete("^Customer", key)
```

5.2.5 Get the next key value for a global node.

```
next = mg_ruby.m_order(<global>, <keys…>)
next = mg_ruby.ma_order(<global>, <keylist>)
```

Types:

```
next              (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M command:

```
Set nextID=$Order(^Customer(1234))
```

Equivalent **mg_ruby** methods:

```
nextID = mg_ruby.m_order("^Customer", 1234)
```

Or:

```
key = [1, "1234"]
nextID = mg_ruby.ma_order("^Customer", key)
```

Example 2 (Parse a global in order):

M command:

```
Set nextID=""
For  Set nextID=$Order(^Customer(1234)) Quit:nextID=""  Do
. Write "<br>", nextID," = ",$Get(^Customer(nextID))
. Quit
```

Equivalent **mg_ruby** method:

```
key = ""
while ((key = mg_ruby.m_order("^Customer", key) != "")
   puts key + " = " + mg_ruby.m_get("^Customer", key))
end
```

Or:

```
key = [1, “”]
while (mg_ruby.ma_order("^Customer", key) != “”)
   puts key[1] + " = " + mg_ruby.ma_get("^Customer", key))
end
```

5.2.6 Get the previous key value for a global node.

```
prev = mg_ruby.m_previous(<global>, <keys…>)
prev = mg_ruby.ma_previous(<global>, <keylist>)
```

Types:

```
prev              (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M command:

```
Set prevID=$Order(^Customer(1234),-1)
```

Equivalent **mg_ruby** methods:

```
prevID = mg_ruby.ma_previous(“^Customer”, 1234)
```

Or:

```
key = [1, "1234"]
prevID = mg_ruby.ma_previous(“^Customer”, key)
```

Example 2 (Parse a global in reverse order):

M command:

```
Set nextID=””
For  Set nextID=$Order(^Customer(1234), -1) Quit:nextID=””  Do
. Write "<br>", nextID," = ",$Get(^Customer(nextID))
. Quit
```

Equivalent **mg_ruby** method:

```
key = ""
while ((key = mg_ruby.m_previous("^Customer", key) != "")
   puts key + " = " + mg_ruby.m_get("^Customer", key))
end
```

Or:

```
key = [1, “”]
while (mg_ruby.ma_previous("^Customer", key) != “”)
   puts key[1] + " = " + mg_ruby.ma_get("^Customer", key))
end
```

5.2.7 Increment the value of a global node.

```
result = mg_ruby.m_increment(<global>, <keys…>, <increment_value>)
```

Types:

```
result            (String)
global            (String)
keys              (Any)
increment_value   (Number)
```

Example:

M command:

```
Set value=$Increment(^Global(“counter”))
```

Equivalent **mg_ruby** method:

```
result = mg_ruby.m_increment("^Global", "counter", 1)
```

This will increment the value of global node **^Global("counter")**, by 1 and return the new value.

5.3 Manipulate the local representation of M data in Ruby

The **mg_ruby** methods in this section create and manipulate the local Ruby data structures that represent M data in the Ruby environment. These (ma_local) methods will be used extensively in the examples shown in subsequent sections.

5.3.1 Set a local node.

```
result = mg_ruby.ma_local_set(<records>, <index>,
                              <keylist>,
                              <data>)
```

Types:

```
result            (String)
records           (List)
index             (int)
keylist           (List)
data              (String)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to assign the next available index number
      or overwrite the existing key value.
-2:
      Instruct the method to assign the next available index number
      at the end of the record set regardless of whether or
      not the key value exists.
```

It should be noted that this method will not automatically insert records into the set in M-order. Use the ‘ma_local_sort’ method to order the contents of a records List.

Example 1:

M command:

```
Set Customer(1234)=”Chris Munt”
```

Equivalent **mg_ruby** method:

```
key = [1, "1234"]
mg_ruby.ma_local_set(Customer, -1, key, “Chris Munt”)
```

5.3.2 Retrieve the data from a local node.

```
data = mg_ruby.ma_local_get(<records>, <index>,
                            <keylist>)
```

Types:

```
data              (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to locate the record required based
      on the key value supplied.
```

Example 1:

M command:

```
Set data=$Get(Customer(1234))
```

Equivalent **mg_ruby** method:

```
key = [1, "1234"]
data = mg_ruby.ma_local_get(Customer, -1, key)
```

5.3.3 Check that a local node exists.

```
defined = mg_ruby.ma_local_data(<records>, <index>,
                                <keylist>)
```

Types:

```
defined           (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to locate the record required based
```

```
      on the key value supplied.
```

Example 1:

M command:

```
Set defined=$Data(Customer(1234))
```

Equivalent **mg_ruby** method:

```
key = [1, "1234"]
defined = mg_ruby.ma_local_data(Customer, -1, key)
```

5.3.4 Delete a local node.

```
result = mg_ruby.ma_local_kill(<records>, <index>,
                               <keylist>)
```

Types:

```
result            (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to locate the record required based
      on the key value supplied.
```

Example 1:

M command:

```
Kill Customer(1234))
```

Equivalent **mg_ruby** method:

```
key = [1, "1234"]
result = mg_ruby.ma_local_kill(Customer, -1, key)
```

5.3.5 Get the next key value for a local node.

```
next = mg_ruby.ma_local_order(<records>, <index>,
                              <keylist>)
```

Types:

```
next              (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The current index number in the records List (if known).
-1:
      Instruct the method to locate the next record based
      on the current key value supplied.
```

Example 1:

M command:

```
Set nextID=$Order(Customer(1234))
```

Equivalent **mg_ruby** method :

```
key = [1, "1234"]
nextID = mg_ruby.ma_local_order(Customer, -1, key)
```

Example 2 (Parse a local record set in order):

M command:

```
Set nextID=""
For  Set nextID=$Order(Customer(1234)) Quit:nextID=""  Do
. Write "<br>", nextID
. Quit
```

Equivalent **mg_ruby** method:

```
key = [1, ""]
while (mg_ruby.ma_local_order(Customer, -1, key) != -1)
   puts '<br>' + key[1]
end
```

5.3.6 Get the previous key value for a local node.

```
prev = mg_ruby.ma_local_get(<records>, <index>,
                            <keylist>)
```

Types:

```
prev              (String)
records           (List: Use List or Vector)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The current index number in the records List (if known).
-1:
      Instruct the method to locate the next record based
      on the current key value supplied.
```

Example 1:

M command:

```
Set prevID=$Order(Customer(1234), -1)
```

Equivalent **mg_ruby** method:

```
key = [1, "1234"]
prevID = mg_ruby.ma_local_previous(Customer, -1, ref key)
```

Example 2 (Parse a local record set in reverse order):

M command:

```
Set prevID=""
For  Set prevID=$Order(Customer(1234), -1) Quit:prevID=""  Do
. Write "<br>", prevID
. Quit
```

Equivalent **mg_ruby** method:

```
key = [1, ""]
while (mg_ruby.ma_local_previous(Customer, -1, key) != -1)
   puts '<br>' + key[1]
end
```

5.3.7 Sort the contents of a local records set.

```
data = mg_ruby.ma_local_sort(<records>)
```

Types:

```
data              (String)
records           (List)
```

This method will sort the contents of a records List into M order. The default M server will be used to perform the sort operation.

Example 1:

mg_ruby method:

```
result = mg_ruby.ma_local_sort(records)
```

5.3.8 Merge a Ruby array to a global.

```
result = mg_ruby.ma_merge_to_db(<global>, <keylist>,
                                <records>, <options>)
```

Types:

```
result            (String)
global            (String)
keylist           (List)
records           (List)
options           (String)
```

The ‘options’ argument can currently take the following value:

```
ks
```

This means ‘**K**ill at **S**erver’. If this option is selected, the M global will be deleted at the level specified within the merge function before the actual merge is performed.

Example 1:

M commands:

```
Set custList(1234)=”Chris Munt”
Set custList(1235)=”Rob Tweed”
Merge ^Customer=custList
```

Equivalent **mg_ruby** method:

```
custList = [] # Clear records List
key = [1, "1234"]
mg_ruby.ma_local_set(custList, -1, key, "Chris Munt")
key = [1, "1235"]
mg_ruby.ma_local_set(custList, -1, key, "Rob Tweed")

key = [0] # No fixed key for merge global
result = mg_ruby.ma_merge_to_db("^Customer", key, custList, "")
```

In both cases the following records will be created in M:

```
^Customer(1234)="Chris Munt"
^Customer(1235)="Rob Tweed"
```

Example 2:

M commands:

```
Set custInvoice(1)="1.2.2001"
Set custInvoice(2)="5.3.2001"
Merge ^CustomerInvoice(1234)=custInvoice
```

Equivalent **mg_ruby** method:

```
custInvoice = [] # Clear records List
key = [1, "1"]
mg_ruby.ma_local_set(custInvoice, -1, key, "1.2.2001")
key = [1, "2"]
mg_ruby.ma_local_set(custInvoice, -1, key, "5.3.2001")

key = [1, "1234"] # Fixed key for merge global
result = mg_ruby.ma_merge_to_db("^CustomerInvoice", key,
                                custInvoice, "")
```

In both cases, the following records will be created in M:

```
^CustomerInvoice(1234,1)="1.2.2001"
^CustomerInvoice(1234,2)="5.3.2001"
```

Example 3 (Using the 'ks' option):

Existing M database:

```
^CustomerInvoice(1234,1)="1.2.2001"
^CustomerInvoice(1234,2)="5.3.2001"
```

```
^CustomerInvoice(1234,3)="7.9.2001”
```

M commands:

```
Set custInvoice(3)=”8.9.2001”
Set custInvoice(4)=”12.11.2001”
Kill ^CustomerInvoice(1234)

Merge ^CustomerInvoice(1234)=custInvoice
```

Equivalent **mg_ruby** method:

```
custInvoice = [] # Clear records List
key = [1, "3"]
mg_ruby.ma_local_set(custInvoice, -1, key, "8.9.2001")
key = [1, "4"]
mg_ruby.ma_local_set(custInvoice, -1, key, "12.11.2001")

key = [1, "1234"] # Fixed key for merge global
result = mg_ruby.ma_merge_to_db("^CustomerInvoice", key,
                                custInvoice, "ks")
```

In both cases, after this operation, M will hold the following records:

```
^CustomerInvoice(1234,3)="8.9.2001"
^CustomerInvoice(1234,4)="12.11.2001”
```

Example 4 (Dealing with multi-dimensional arrays):

M commands:

```
Set custInvoice(1234,1)=”1.2.2001”
Set custInvoice(1234,2)=”5.3.2001”
Set custInvoice(1235,7)=”7.6.2002”
Set custInvoice(1235,8)=”1.12.2002”
Merge ^CustomerInvoice=custInvoice
```

Equivalent **mg_ruby** method:

```
custInvoice = [] # Clear records List
key = [2, "1234", “1”]
mg_ruby.ma_local_set(custInvoice, -1, key, "1.2.2001")
key = [2, "1234", “2”]
mg_ruby.ma_local_set(custInvoice, -1, key, "5.3.2001")
key = [2, "1235", “7”]
mg_ruby.ma_local_set(custInvoice, -1, key, "7.6.2002")
key = [2, "1235", “8”]
mg_ruby.ma_local_set(custInvoice, -1, key, "1.12.2002")

key = [0] # No fixed key for merge global
```

```
result = mg_ruby.ma_merge_to_db("^CustomerInvoice", key,
                                custInvoice, "")
```

In both cases, after this operation, M will hold the following records:

```
^CustomerInvoice(1234,1)=”1.2.2001”
^CustomerInvoice(1234,2)=”5.3.2001”
^CustomerInvoice(1235,7)=”7.6.2002”
^CustomerInvoice(1235,8)=”1.12.2002”
```

5.3.9 Merge a global to a Ruby array.

```
result = mg_ruby.ma_merge_from_db(<global>, <keylist>,
                                  <records>, <options>)
```

Types:

```
result            (String)
global            (String)
keylist           (List)
records           (List)
options           (String)
```

The last ‘options’ argument is reserved for future use.

Example 1:

M database:

```
^Customer("1234")="Chris Munt"
^Customer("1235")="Rob Tweed”
```

M command:

```
Merge custList=^Customer
```

Equivalent **mg_ruby** method:

```
custList = [] # Clear records List

key = [0] # No fixed key for merge global
result = mg_ruby.ma_merge_from_db("^Customer", key, custList, “”)
```

In both cases, the following records will be created in Ruby:

```
custList[1] = “1234” + “Chris Munt”
custList[2] = “1235” + “Rob Tweed”
```

To get the number of records:

```
custList.count
```

Example 2:

M database:

```
^CustomerInvoice(1234,1)="1.2.2001"
^CustomerInvoice(1234,2)="5.3.2001”
```

M commands:

```
Merge custInvoice=^CustomerInvoice(1234)
```

Equivalent **mg_ruby** method:

```
custInvoice = [] # Clear records List

key = [1, "1234"] # Fixed key for merge global
result = mg_ruby.ma_merge_from_db("^Customer", key,
                                  custInvoice, “”)
```

In both cases, the following records will be created in Ruby:

```
custInvoice[1] = "1" + "1.2.2001"
custInvoice[2] = "2" + "5.3.2001”
```

Example 3 (Dealing with multi-dimensional arrays):

M database:

```
^CustomerInvoice(1234,1)=”1.2.2001”
^CustomerInvoice(1234,2)=”5.3.2001”
^CustomerInvoice(1235,7)=”7.6.2002”
^CustomerInvoice(1235,8)=”1.12.2002”
```

M commands:

```
Merge custInvoice=^CustomerInvoice
```

Equivalent **mg_ruby** method:

```
custInvoice = [] # Clear records List

key = [0] # No fixed key for merge global
result = mg_ruby.ma_merge_from_db("^CustomerInvoice", key,
```

```
                                    custInvoice, "")
```

In both cases, the following records will be created in Ruby:

```
custInvoice[1] = "1234" + "1" + "1.2.2001"
custInvoice[2] = "1234" + "2" + "5.3.2001"
custInvoice[3] = "1235" + "7" + "7.6.2002"
custInvoice[4] = "1235" + "8" + "1.12.2002"
```

5.4 Direct access to M functions and procedures

M provides a rich scripting language (formally known as MUMPS or M) for developing function and procedures. The following methods are supplied by the mg_ruby module for the purpose of directly accessing M functions and procedures within the Ruby environment.

A note on terminology: M procedures and functions are contained within blocks of code known as routines. A function or procedure is referred to using the following syntax:

```
<FunctionName>^<Routine>
```

For example:

```
MyFunction^MyRoutine
```

A routine name of 'MyRoutine' will be used throughout the following examples.

Input arguments to M functions can be expressed as a variable number of input arguments for which the Ruby method is prefixed by 'm_'. Input arguments to M functions can also be supplied as Ruby lists in which case the Ruby method is prefixed by 'ma_'. The latter form must be used if arguments are to be passed by reference (i.e. M will be modifying their values).

5.4.1 Call a M extrinsic function.

```
result = mg_ruby.m_function(<function>, <arguments…>)

result = mg_ruby.ma_function(<function>, <arguments>,
                             <no_arguments>)
```

Types:

```
result            (String)
function          (String)
arguments         (Any)
argumentlist      (List)
no_arguments      (int)
```

The following methods are used to create arguments to a function call.

Add an item to the arguments list:

```
mg_ruby.ma_arg_set(<arguments>, <arg_no>, <item>, <by_ref))
```

Where:

```
arguments   (List)
      Arguments array.  Holds physical data and associated properties.
arg_no      (int)
      Argument number.
item  (List or String)
      Argument
by_ref      (int)
      By reference flag - set to 1 for pass-by-reference; 0 for
      pass-by-value.
```

Example 1 (A simple function call):

M procedure:

```
GetTime()   ; Get the current date and time in M internal format
            Set result=$Horolog
            Quit result
            ;
```

This function returns the date and time in M's internal format.

Equivalent **mg_ruby** method:

```
time = mg_ruby.m_function("GetTime^MyRoutine")
```

Or:

```
time = mg_ruby.ma_function("GetTime^MyRoutine", arguments, 0)
```

Example 2 (Passing arguments by reference):

M procedure:

```
GetDateDecoded(dateDisp)      ; Get the current date in decoded form
                              Set result=+$Horolog
                              Set dateDisp=$ZD(result,2)
                              Quit result
                              ;
```

This function returns the date in M's internal format. In addition, it will return the date in a human-readable format (i.e. decoded).

Equivalent **mg_ruby** method:

```
mg_ruby.ma_arg_set(arguments, 1, "", 1)
date = mg_ruby.ma_function("GetDateDecoded^MyRoutine",
                           arguments, 1)
dateDisp = arguments[1]
```

Notice that the 'by reference' flag is set in the 'ma_arg_set' method.

Example 3 (Another simple function call):

M procedure:

```
GetCust(custID)   ; Return the customer name
                  Set cust=$Get(^Customer(custID))
                  Quit cust
                  ;
```

Equivalent **mg_ruby** method:

```
cust = mg_ruby.m_function("GetCust^MyRoutine", "1234")
```

Or:

```
mg_ruby.ma_arg_set(arguments, 1, "1234", 0)
cust = mg_ruby.ma_function("GetCust^MyRoutine", arguments, 1)
```

Example 4 (Passing an array from Ruby to M):

M procedure:

```
ProcCustList(custList)  ; Return the number of active customers
                  Set custID="",activeCust=0
                  For  Set CustID=$Order(^CustList(custID)) Do
                  . If $Data(^CustOrderStatus(custID))="Active" Do
                  .. Set activeCust=activeCust+1
                  Quit activeCust
                  ;
```

Equivalent **mg_ruby** method:

```
custList = []
key = [1, "1234"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1235"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1236"]
mg_ruby.ma_local_set(custList, -1, key, "")

mg_ruby.ma_arg_set(arguments, 1, custList, 0)
activeCust = mg_ruby.ma_function("ProcCustList^MyRoutine",
                                 arguments, 1)
```

Example 5 (Passing an array from M to Ruby):

M procedure:

```
ActCList(custList) ; Return a list of active customers
                   Set custID="",activeCust=0
```

```
                    For  Set CustID=$Order(^CustOrderStatus(custID))
Quit:custID=””  Do
                    . If $Data(^CustOrderStatus(custID))=”Active” Do
                    .. activeCust=activeCust+1
                    .. Set custList(custID)=$Get(^Customer(custID))
                    Quit activeCust
                    ;
```

Equivalent **mg_ruby** method:

```
custList = []
mg_ruby.ma_arg_set(arguments, 1, custList, 1)
activeCust = mg_ruby.ma_function(“ActCList^MyRoutine”,
                                  arguments, 1)
```

Example 6 (Using a M function to modify a Ruby array):

M procedure:

```
GetCustNames(custList)  ; Return the names for a list of customers
                  Set custID=””,result=””
                  For  Set CustID=$Order(^CustList(custID)) Do
                  . Set custList(custID)=$Get(^Customer(custID))
                  Quit result
                  ;
```

Equivalent **mg_ruby** method:

```
custList = []
key = [1, "1234"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1235"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1236"]
mg_ruby.ma_local_set(custList, -1, key, "")

mg_ruby.ma_arg_set(arguments, 1, custList, 1)
activeCust = mg_ruby.ma_function(“GetCustNames^MyRoutine”,
                                  arguments, 1)
```

5.4.2 Return a block of HTML from a M function.

```
connection_handle = mg_ruby.ma_html(<function>,
                                    <arguments>,
                                    <no_arguments>)

buffer = mg_ruby.ma_get_stream_data(<connection_handle>)
```

Types:

```
connection_handle (int)
function          (String)
function          (String)
arguments         (List)
no_arguments      (int)
```

Usage:

```
connection_handle = mg_ruby.ma_html_ex(<function>,
                                       <arguments>,
                                       <no_arguments>)
buffer = mg_ruby.ma_get_stream_data(<connection_handle>)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(<connection_handle>)
end
```

Example 1:

M procedure:

```
MyHtml      ; Return some HTML to Ruby
            Write "<p>This text was returned from M"
            Write "<br>You can return as text much as you like …"
            Quit
            ;
```

Equivalent **mg_ruby** method:

```
connection_handle = mg_ruby.ma_html_ex("MyHtml^MyRoutine",
                                       arguments,
                                       0)
buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

Example 2:

M procedure:

```
GetHTML(custID)   ; Return some HTML to Ruby
                  Write "<p>This text was returned from M"
                  Write "<br>You can return as text much as you like …"
                  Write "<br>A single argument '",custID,"' was passed
from Ruby"
                  Quit
                  ;
```

Equivalent **mg_ruby** method:

```
mg_ruby.ma_arg_set(arguments, 1, "1234", 0)
connection_handle = mg_ruby.ma_html_ex("MyHtml^MyRoutine",
                                       arguments,
                                       1)
buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

Example 3 (Passing an array from Ruby to M):

M procedure:

```
GetCTable(custList)     ; Return a table of customers to Ruby
                  Write "<table>"
                  Set custID=""
                  For  Set CustID=$Order(^CustList(custID))
Quit:custID=""  Do
                  . Set custName=$Get(^Customer(custID))
                  . Set custList(custID)=custName ; Return name to Ruby
                  . Write "<tr><td>",custID,"</td>"
                  . Write "<td>",custName,"</td></tr>"
                  Write "</table>"
                  Quit
                  ;
```

Equivalent **mg_ruby** method:

```
custList = []
key[0] = "1";
key[1] = "1234";
mg_ruby.ma_local_set(custList, -1, key, "")
```

```
key[1] = "1235";
mg_ruby.ma_local_set(custList, -1, key, "")
key[1] = "1236";
mg_ruby.ma_local_set(custList, -1, key, "")

mg_ruby.ma_arg_set(arguments, 1, custList, 0)
connection_handle = mg_ruby.ma_html_ex("GetCTable^MyRoutine",
                                       arguments,
                                       1)
buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

5.5 Transaction Processing

M DB Servers implement Transaction Processing by means of the methods described in this section.

5.5.1 Start a Transaction

```
result = mg_ruby.m_tstart()
```

On successful completion this method will return zero, or an error code on failure.

Example:

```
result = mg_ruby.m_tstart()
```

5.5.2 Determine the Transaction Level

```
result = mg_ruby.m_tlevel()
```

Transactions can be nested and this method will return the level of nesting. If no Transaction is active this method will return zero. Otherwise a positive integer will be returned to represent the current depth of Transaction nesting.

Example:

```
tlevel = mg_ruby.m_tlevel()
```

5.5.3 Commit a Transaction

```
result = mg_ruby.m_tcommit()
```

* On successful completion this method will return zero, or an error code on failure.

Example:

```
result = mg_ruby.m_tcommit()
```

5.5.4 Rollback a Transaction

```
result = mg_ruby.m_trollback()
```

On successful completion this method will return zero, or an error code on failure.

Example:

```
result = mg_ruby.m_trollback()
```

5.6 Direct access to Caché methods

Caché provides an Object Oriented development environment (Caché Objects). The following methods are supplied by the mg_ruby module for the purpose of directly accessing Caché class methods from within the Ruby environment.

This section will show the same examples used in the previous section (Caché functions and procedures), but implemented as methods of an object class.

The methods described here will belong to a class called 'MyUtilities.MyClass'. This translates to a Caché package name of 'MyUtilities' and a class name of 'MyClass'. Of course, in a real application the methods described would be contained within a class more appropriate to the functionality they implement.

Input arguments to M methods can be expressed as a variable number of input arguments for which the Ruby method is prefixed by 'm_'. Input arguments to M methods can also be supplied as Ruby lists in which case the Ruby method is prefixed by 'ma_'. The latter form must be used if arguments are to be passed by reference (i.e. M will be modifying their values).

5.6.1 Call a Caché ClassMethod.

```
result = mg_ruby.m_classmethod(<class_name>,
                               <method_name>,
                               <arguments…>)

result = mg_ruby.ma_classmethod(<class_name>,
                                <method_name>,
                                <argumentlist>, <no_arguments>)
```

Types:

```
result            (String)
class_name        (String)
method_name       (String)
arguments         (Any)
argumentlist      (List)
no_arguments      (int)
```

Example 1 (A simple method):

M/Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetTime()
```

```
{
      ; Get the current date and time in Caché's internal format
      Set result=$Horolog
      Quit result
}
```

This method returns the date and time in Caché's internal format.

Equivalent **mg_ruby** method:

```
date = mg_ruby.m_classmethod("MyUtilities.MyClass", "GetTime")

Or

date = mg_ruby.ma_classmethod("MyUtilities.MyClass", "GetTime",
                              arguments, 0)
```

Example 2 (Passing arguments by reference):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetDateDecoded(dateDisp)
{
      ; Get the current date in decoded form
      Set result=+$Horolog
      Set dateDisp=$ZD(result,2)
      Quit result
}
```

This method returns the date in Caché's internal format. In addition, it will return the date in a human-readable format (i.e. decoded).

Equivalent **mg_ruby** method:

```
mg_ruby.ma_arg_set(arguments, 1, "", 1)
date = mg_ruby.ma_classmethod("MyUtilities.MyClass",
                              "GetDateDecoded",
                              arguments, 1)
dateDisp = arguments[1]
```

Example 3 (Another simple method):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …
```

```
ClassMethod GetCust(custID)
{
      ; Return the customer name
      Set cust=$Get(^Customer(custID))
      Quit cust
}
```

Equivalent **mg_ruby** method:

```
cust = mg_ruby.m_classmethod("MyUtilities.MyClass", "GetCust",
                             "1234")
```

Or

```
mg_ruby.ma_arg_set(arguments, 1, "1234", 0)
cust = mg_ruby.ma_classmethod("MyUtilities.MyClass", "GetCust",
                              arguments, 1)
```

Example 4 (Passing an array from Ruby to Caché):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod ProcCustList(custList)
{
      ; Return the number of active customers
      Set custID="",activeCust=0
      For  Set CustID=$Order(^CustList(custID)) Do
      . If $Data(^CustOrderStatus(custID))="Active" Do
      .. Set activeCust=activeCust+1
      Quit activeCust
}
```

Equivalent **mg_ruby** method:

```
custList = []
key = [1, "1234"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1235"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1236"]
mg_ruby.ma_local_set(custList, -1, key, "")

mg_ruby.ma_arg_set(arguments, 1, custList, 0)
activeCust = mg_ruby.ma_classmethod("MyUtilities.MyClass",
                                    "ProcCustList",
                                    arguments, 1)
```

Example 5 (Passing an array from Caché to Ruby):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod ActCList(custList)
{
      ; Return a list of active customers
      Set custID="",activeCust=0
      For  Set CustID=$Order(^CustOrderStatus(custID)) Quit:custID=""
Do
      . If $Data(^CustOrderStatus(custID))="Active" Do
      .. activeCust=activeCust+1
      .. Set custList(custID)=$Get(^Customer(custID))
      Quit activeCust
}
```

Equivalent **mg_ruby** method:

```
custList = []
mg_ruby.ma_arg_set(arguments, 1, custList, 1)
activeCust = mg_ruby.ma_classmethod("MyUtilities.MyClass",
                                    "ActCList",
                                     arguments, 1)
```

Example 6 (Using a Caché method to modify a Ruby array):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetCustNames(custList)
{
      ; Return the names for a list of customers
      Set custID="",result=""
      For  Set CustID=$Order(^CustList(custID)) Do
      . Set custList(custID)=$Get(^Customer(custID))
      Quit result
}
```

Equivalent **mg_ruby** method:

```
custList = []
key = [1, "1234"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1235"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [1, "1236"]
mg_ruby.ma_local_set(custList, -1, key, "")
```

```
mg_ruby.ma_arg_set(arguments, 1, custList, 0)
activeCust = mg_ruby.ma_classmethod("MyUtilities.MyClass",
                                    "GetCustNames",
                                    arguments, 1)
```

5.6.2 Return a block of HTML from a Caché ClassMethod.

```
connection_handle = mg_ruby.ma_html_classmethod_ex(<class_name>,
                                                   <method_name>,
                                                   <arguments>,
                                                   <no_arguments>)

buffer = mg_ruby.ma_get_stream_data(<connection_handle>)
```

Types:

```
connection_handle (int)
class_name        (String)
method_name       (String)
arguments         (List)
no_arguments      (int)
```

Usage:

```
connection_handle = mg_ruby.ma_html_classmethod_ex(<class_name>,
                                                   <method_name>,
                                                   <arguments>,
                                                   <no_arguments>)

buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

Example 1:

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod MyHtml()
{
      ; Return some HTML to Ruby
      Write "<p>This text was returned from Caché"
      Write "<br>You can return as text much as you like …"
      Quit
}
```

Equivalent **mg_ruby** method:

```
connection_handle =

mg_ruby.ma_html_classmethod_ex("MyUtilities.MyClass",
                                                    "MyHtml",
                                                    arguments,
                                                    0)
buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

Example 2:

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetHTML(custID)
{
      ; Return some HTML to Ruby
      Write "<p>This text was returned from Caché"
      Write "<br>You can return as text much as you like …"
      Write "<br>A single argument '",custID,"' was passed from Ruby"
      Quit
}
```

Equivalent **mg_ruby** method:

```
mg_ruby.ma_arg_set(arguments, 1, "1234", 0)
connection_handle = mg_ruby.ma_html_classmethod_ex(
                                                    "MyUtilities.MyClass",
                                                    "GetHTML",
                                                    arguments,
                                                    1)

buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

Example 3 (Passing an array from Ruby to Caché):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetCTable(custList)
{
```

```
      ; Return a table of customers to Ruby
      Write "<table>"
      Set custID=""
      For  Set CustID=$Order(^CustList(custID)) Quit:custID=""  Do
      . Set custName=$Get(^Customer(custID))
      . Set custList(custID)=custName ; Return name to Ruby
      . Write "<tr><td>",custID,"</td>"
      . Write "<td>",custName,"</td></tr>"
      Write "</table>"
      Quit
}
```

Equivalent **mg_ruby** method:

```
custList = []
key = [0, "1234"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [0, "1235"]
mg_ruby.ma_local_set(custList, -1, key, "")
key = [0, "1236"]
mg_ruby.ma_local_set(custList, -1, key, "")

mg_ruby.ma_arg_set(arguments, 1, custList, 0)
connection_handle = mg_ruby.ma_html_classmethod_ex(
                                                "MyUtilities.MyClass",
                                                "GetCTable",
                                                arguments,
                                                1)
buffer = mg_ruby.ma_get_stream_data(connection_handle)
while (buffer != "")
   puts buffer
   buffer = mg_ruby.ma_get_stream_data(connection_handle)
end
```

5.7 Direct access to Ruby and M functions from the browser

There are many situations in web application programming where it is desirable to directly access server-side functionality from the context of the browser environment. For example, such a facility could be use for performing individual field validation and simple table lookups. For trivial operations of this sort it is rather expensive to have to submit the whole form to the server in order to communicate with the database. Browser components are supplied with mg_ruby to provide the capability of accessing Ruby and, subsequently, M functionality directly through browser-based scripting (for example, JavaScript).

Using the XMLHTTP script (mg_client.js)

The XMLHTTP script file (JavaScript) is included in the mg_ruby distribution:

```
/js/mg_client.js
```

The procedure for installing this file in the hosting web server environment is the same as for the Java Applet. The examples shown below are based on the XMLHTTP script being installed in the web server's documents root directory.

This represents the functionality of the mg_client browser component implemented in standard JavaScript. The new XMLHTTP functionality contained within the latest browsers is used to implement the connectivity between the browser environment and M (via Ruby /mg_ruby).

The rationale for providing this implementation is because of ongoing problems (both technical and commercial) with Java technology embedded in the newer browsers. It is therefore felt that customers should be offered a non-Java-based equivalent in order to secure their applications for the future.

Using the functions contained within 'mg_client.js' is straightforward. Indeed the interface is identical to that provided by the functionally equivalent Java Applet.

The following procedure should be used:

1. Include the JavaScript file in the hosting page instead of the Java Applet:

```
<script language="JavaScript" src="/mg_client.js"></script>
```

Example:

```
<HTML>
<HEAD><TITLE>My Form</TITLE>
<script language="JavaScript" src="/mg_client.js"></script>
```



```
</HEAD>
<BODY>

etc ...
```

2. Having initialized the JavaScript environment for 'mg_client.js' as described in the previous step, the internal functions can be used in exactly the same way as they would have been used with the Java Applet.

The function names (being plain JavaScript functions) are not qualified with a name. Therefore, what would have been ...

```
result = mg_client.server_proc(<URL>)
```

... for 'mg_client.class' (or mg_client.jar) is now ...

```
result = server_proc(<URL>)
```

... for 'mg_client.js'

The example shown (and described) in the previous section is shown below, recoded to use the XMLHTTP script instead of the Java Applet. Again, Ruby is used to script the form.

```
require "cgi"
require 'mg_ruby'

cgi = CGI.new
mg_ruby = MG_RUBY.new()

begin

key = Array(2)

puts
puts 'Status: 200 OK'
puts 'Content-Type: text/html'
puts

if cgi.has_key?("m_fun") then
   m_funx = cgi.params['m_fun']
   m_fun = m_funx[0]
   m_argx = cgi.params['m_arg']
   m_arg = m_argx[0]
   if m_fun != "" then
      if m_fun == "NameLookup" then
         key[0] = 1
         key[1] = m_arg
         puts mg_ruby.ma_return_to_client(mg_ruby.ma_get("^MGWCust",
key))
         exit()
      end
   end
end

puts '<html>'
puts '<head>'

puts '<script language="JavaScript" src="/m_client.js"></script>'

puts '<script LANGUAGE = "JavaScript">'

puts 'function NameLookup(FormObject, value) {'
puts '   FormObject.value =
server_proc("/GetName.rb?m_fun=NameLookup&m_arg=" + value);'
puts '   return;'
puts '}'

puts '</script>'

puts '<TITLE>Ruby to M - applet demo</TITLE>'

puts '</head>'
puts '<body>'
puts '<form>'

puts '<h1>Ruby to M - applet demo</h1>'

puts 'Customer No <INPUT TYPE=TEXT NAME=id SIZE=30
ONCHANGE="NameLookup(form.name, this.value)">'
puts 'Name <INPUT TYPE=TEXT NAME=name SIZE=30>'
```

```
puts '<p>'

puts 'Setup database when we load form ...'


key = [1, "1"]
mg_ruby.ma_set("^MGWCust", key, "Chris Munt")
key[1] = "2"
mg_ruby.ma_set("^MGWCust", key, "Rob Tweed")

puts '<hr>'

puts '</form>'
puts '</body>'
puts '</html>'


rescue RuntimeError
  puts "RuntimeError: " + $!
  raise
rescue StandardError
  puts "StandardError: " + $!
  raise
rescue TypeError
  puts "TypeError: " + $!
  raise
end
```

5.8 Handling error conditions

```
error = mg_ruby.m_get_last_error()
```

Occasionally it is necessary for an **mg_ruby** method to return an error condition after failing to complete the prescribed task. For example, a target M server may be unavailable or there may be a problem with the supporting network.

Use the above method to return the last error message. The internal error message variable will only be reset as a result of a call to this method. Therefore, this method can be placed at the end of a series of **mg_ruby** method calls in order to see whether there were any error conditions encountered in processing the form.

On error, the **mg_ruby** methods will return an error code. The value of the error code is minus 1 by default (-1). This will be returned as either a string ("-1") or number (-1) depending on context. The corresponding error message can be obtained using the 'ma_get_last_error' method. For example:

```
key = [0]
error_code = mg_ruby.ma_kill("^MGWCust", key)
if (error_code == "-1")
   error_message = mg_ruby.m_get_last_error()
   puts "<br>ERROR: ",  error_code, " ", error_message
end
```

Of course, in some cases the error code of '-1' may clash with legitimate return values. If this is anticipated to be the case, you can specify your own error code using the following method:

```
mg_ruby.m_set_error_code(error_code)
```

For example:

```
mg_ruby.m_set_error_code("-7") # Reset error code
key = [0]
error_code = mg_ruby.ma_kill("^MGWCust", key)
if (error_code == "-7")
   error_message = mg_ruby.m_get_last_error()
   puts "<br>ERROR: ",  error_code, " ", error_message
end
```

Scope

The public global variables mentioned in this section are scoped in accordance with the instance of the **mg_ruby** module in use.

In addition to the facilities mentioned above, it is good practice to use the built-in exception handling functionality provided by Ruby. For example:

```
begin

# code block

rescue RuntimeError
  puts "RuntimeError: " + $!
  raise
rescue StandardError
  puts "StandardError: " + $!
  raise
rescue TypeError
  puts "TypeError: " + $!
  raise
end
```

5.9 Handling Ruby strings that exceed the maximum size allowed under M

M-based systems impose a hard limit on the length of string that can be assigned to global nodes and variables in programs. For example, in Caché the maximum string length is 32767 Bytes.

Ruby does not impose any such limit and the following convention can be used to trade 'oversize' strings between M and Ruby. Oversize strings are broken up into individual sections in M as follows:

```
variable = <First Section>
variable(extra, <Section Number>) = <Subsequent Section>
```

For example, take a string that exceeds the maximum length allowed in M by a factor of three:

```
variable = <First Section>
variable(extra, 1) = <Second Section>
variable(extra, 2) = <Third Section>
```

The same convention applies to arrays:

```
array("key") = <First Section>
array("key", extra, 1) = <Second Section>
array("key", extra, 2) = <Third Section>
```

The special variable '**extra**' is set by the **mg_ruby** engine. Its default value is **ASCII 1**. There is a possibility that this special data marker will clash with your data particularly where arrays are concerned. For example, you may have data keyed by the default value of ASCII 1. If this is the case, you can change the value of extra by editing its value in **mg_ruby** core routine: `VARS^%ZMGWSIS`

```
VARS  ; Public system variables
      Set extra=$C(1)
```

It should be noted however that this variable can only be reassigned on a per-installation basis in the above procedure. It should not be dynamically changed in other code.

mg_ruby will handle the transformation of oversize values to and from this form between Ruby and M and vice versa. The Ruby scripts are unaffected by these transformations.

Example 1 (Pass an oversize Ruby variable to Caché):

M procedure:

```
MyFun(arg)  ; Accept an oversize variable
            Set s1=$Get(arg)
            Set s1=$Get(arg(extra,1))
            Set s1=$Get(arg(extra,2))
            Quit 1
            ;
```

Equivalent Ruby method:

```
var = "large value …….."
mg_ruby.ma_arg_set(arguments, 1, var, 0)
result = mg_ruby.ma_proc("MyFun^MyRoutine", arguments, 1)
```

Example 2 (Pass an oversize M variable to Ruby – by reference):

M procedure:

```
MyFun(arg)  ; Accept an oversize variable
            Set arg="first section …….."
            Set arg(extra,1)="second section …….."
            Set arg(extra,2))="third section …….."
            Quit 1
            ;
```

Equivalent Ruby method:

```
var = "large value …….."
mg_ruby.ma_arg_set(arguments, 1, var, 1)
result = mg_ruby.ma_proc("MyFun^MyRoutine", arguments, 1)
var = arguments[1]
```

Example 3 (Pass an oversize M variable to Ruby – by return value):

Note the use of the '**`oversize`**' flag to instruct the **mg_ruby** engine to expect additional sections of an oversize return value to be held in global node **`^WORK($Job,0,`** where **`$Job`** is the M process ID returned by the M environment.

M procedure:

```
MyFun()     ; Return an oversize variable
            Set result="first section …….."
            Set ^WORKJ($Job,0,extra,1)="second section …….."
            Set ^WORKJ($Job,0,extra,2))="third section …….."
            Set oversize=1
            Quit result
            ;
```

Equivalent Ruby method:

```
result = mg_ruby.ma_proc("MyFun^MyRoutine", arguments, 0)
```

Example 4 (Pass an oversize Ruby array node to M):

M procedure:

```
MyFun(array) ; Accept an oversize array node
             Set s1=$Get(array("key to long string")
             Set s2=$Get(array("key to long string",extra,1)
             Set s3=$Get(array("key to long string",extra,2)
             Quit 1
             ;
```

Equivalent Ruby method:

```
var = "large value …….."
key[0] = 1
key[1] = "key to long string"
mg_ruby.ma_local_set(records, -1, key, " large value ……..")
mg_ruby.ma_arg_set(arguments, 1, records, 0)
result = mg_ruby.ma_proc("MyFun^MyRoutine", arguments, 1)
```

Example 5 (Pass an oversize M array node to Ruby):

M procedure:

```
MyFun(array) ; Accept an oversize array node
             Set array(“key to long string”)=”Section 1”
             Set array(“key to long string”,extra,1”=”Section 2”
             Set array(“key to long string”,extra,2) ”)=”Section 3”
             Quit 1
             ;
```

Equivalent Ruby method:

```
var = “large value …….”
key[0] = 1
key[1] = "key to long string"
mg_ruby.ma_local_set(records, -1, key, " large value …….")
mg_ruby.ma_arg_set(arguments, 1, records, 1)
result = mg_ruby.ma_proc(“MyFun^MyRoutine”, arguments, 1)
```

5.10 Handling large Ruby arrays/lists in M

```
mg_ruby.m_set_storage_mode(mode)
```

In addition to imposing limits on the maximum length of string that can be used, M also limits the amount of memory (or *partition* space) that each process can use. Ruby, on the other hand, does not impose such limits but must, of course, work within the system limits imposed by the hosting computer. If large Ruby arrays are sent to M, it may be necessary to specify that these arrays be held in a permanent storage within the M environment (i.e. a workfile) rather than in memory. The amount of memory that M allows for each process will depend on individual configurations.

The 'storage_mode' facility gives **mg_ruby** software some control over how array data is projected to the M environment.

Mode 0

```
mg_ruby.m_set_storage_mode(0)
```

This is the default. Ruby arrays are projected into (and out of) the M environment as simple memory-based arrays.

Mode 1

```
mg_ruby.m_set_storage_mode(1)
```

Ruby arrays are projected into (and out of) the M environment as equivalently structured global arrays (in permanent storage). These globals are structured as follows:

```
^WORKJ($Job,argn,
```

Where:

$Job – The Cache process ID (supplied by the M environment).
argn - The argument number in the function call.

Example 1 (Pass a Ruby array into M global storage):

M procedure:

```
MyFun(array) ; Process an array held in a global
             Set key=""
             For  Set key=$Order(^WORKJ($Job,1,key)) Quit:key=""  Do
             . Set data=$Get(^WORKJ($Job,1,key))
             . Quit
             Quit 1
             ;
```

Equivalent Ruby method:

```
mg_ruby.m_set_storage_mode(1)
records = [] # Clear records array
key[0] = 1
key[1] = "key 1"
mg_ruby.ma_local_set(records, -1, key, "value 1")
key(1) = "key 2"
mg_ruby.ma_local_set(records, -1, key, "value 2")
mg_ruby.ma_arg_set(arguments, 1, records, 0)
result = mg_ruby.ma_proc("MyFun^MyRoutine", arguments, 1)
```

Example 2 (Pass a M array held in global storage to Ruby):

M procedure:

```
MyFun(array) ; Process an array held in a global
             Set ^WORKJ($Job,1,"key 1")="Value 1"
             Set ^WORKJ($Job,1,"key 2")="Value 2"
             Set ^WORKJ($Job,1,"key to long string")="Section 1"
             Set ^WORKJ($Job,1,"key to long string",extra,1)="Section 2"
             Set ^WORKJ($Job,1,"key to long string",extra,2)="Section 3"
             Quit 1
             ;
```

Equivalent Ruby method:

```
mg_ruby.m_set_storage_mode(1)
records = [] # Clear records array
mg_ruby.ma_arg_set(arguments, 1, records, 1)
result = mg_ruby.ma_proc("MyFun^MyRoutine", arguments, 1)
```

Scope

The value of ‘storage_mode' is scoped in accordance with the instance of the **mg_ruby** module and page handle in use.

6 License

http://www.mgateway.com
Email: cmunt@mgateway.com