Refinitiv Management Classes Version 2.2.0

TUTORIAL GUIDE

Document Version: 2.2.0
Date of issue: December 2020
Document ID: RMC220UMTUT.200

REFINITIV™

Contents

1 Introduction

1.1 Scope

This tutorial guide explains the example programs provided for the Refinitiv Management Classes. It helps the user to get started with writing an application using both Refinitiv Management Classes consumer and publisher APIs.

This manual is directed towards engineers and system administrators who should have, at a minimum, a degree of familiarity with the Refinitiv Real-Time Distribution System and a working knowledge of C++.

1.2 Organization of Manual

- Section 1 – Introduction: Introduction to this document, including a glossary.
- Section 2 – Product Description: Briefly describes the purpose of the Refinitiv Management Classes and gives an overview of managed and managing applications.
- Section 3 – Managed Application: A sample program is used to explain how to create a managed application.
- Section 4 – Managing Application: A sample program is used to explain how to create a managing application.

1.3 Conventions Used in this Manual

Various text font styles have been employed to clarify information presented in this manual.

- A user entry appears in this manual in a bold font as follows: **`User entry appears in this style`**.
- Variables (i.e., information that must be supplied by the user) are indicated in italics. In the following example, ***`DEVICE`*** is variable whose value must be determine and set by the user:

  `tar -xvf` ***`DEVICE`***

- Directory names and file names usually appear in a bold italic font as follows: ***/etc/passwd***

1.4 Glossary

TERM	DESCRIPTION
IPC	Inter Process Communication
Node	A device connected to a network cable, usually referring to a server or workstation. The node may be directly or indirectly attached to the Refinitiv Real-Time Distribution System.
Managed Application	A management-enabled application whose role is mainly, but not limited to, that of a Publisher.
Managing Application	A management-enabled application whose role is mainly, but not limited to, that of a Consumer.
Refinitiv Real-Time Distribution System Node	A workstation or device executing any of the Refinitiv Real-Time Distribution System software components. Most notably these components are any Sink Source Library Infrastructure component such as the Refinitiv Real-Time Advanced Distribution Server or Refinitiv Real-Time Advanced Data Hub.
RTDS	Refinitiv Real-Time Distribution System
Triarch	Trading Room Information Architecture

Table 1: Glossary

2 Product Description

Refinitiv Management Classes is an interface API that provides a fast and efficient method of IPC, the shared memory methodology. Developers are able to instrument their applications with Refinitiv Management Classes API, which offers a standard manageability architecture.

Refinitiv Real-Time Distribution System components are instrumented with Refinitiv Management Classes API to support management capability.

A managed application (also known as a publisher application) is an application that publishes statistical information and configuration variables to a piece of shared memory by using Refinitiv Management Classes Publisher API. A managing application (also know as a consumer application) monitors and controls these management data by using Refinitiv Management Classes Consumer API.

The managed application uses managed objects to publish statistical, performance and control parameters. The managing application subscribes to these managed objects and receives notification in the form of callbacks as the information changes. In the case of control parameters, the managing application is allowed to alter the value of these parameters. As a result, the managing application alters the behaviors of the managed application through changes of the managed parameters.

In the following diagram, RMC stands for Refinitiv Management Classes.

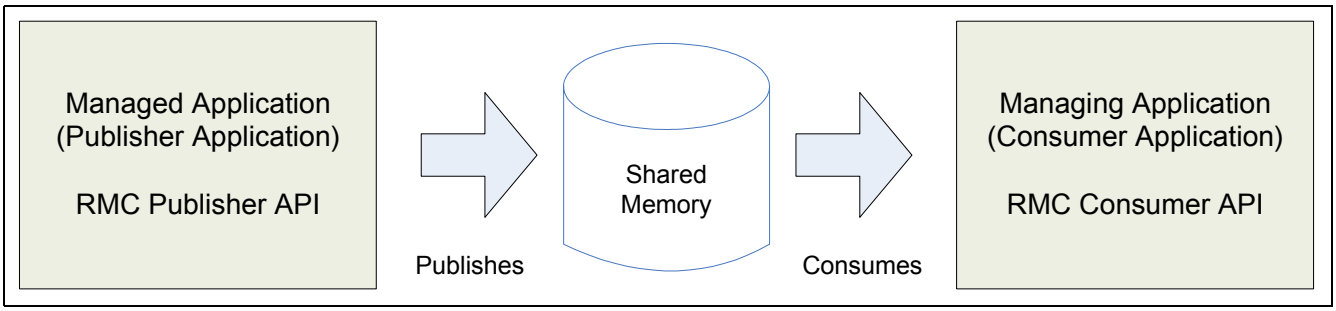

Figure 1. Shared Memory

Figure 1 illustrates the interaction between a managed application and a managing application. Managed objects are made available to the managing application by communicating via the shared memory.

Typically, the shared memory is divided into segments. Each segment is being managed and controlled by a server with a unique shared memory key. The server may partition the shared memory in order to use the shared memory segment for more than one purpose. This model is a one-server-to-n-client relationship where the number of clients is determined by the server.

3 Managed Application

3.1 Overview

Managed applications are perceived as a collection of managed objects. Each managed object contains a number of managed variables. To make an object, along with its variables, available to managing applications, the object must become a descendant of **RTRPublicObject**. Within the same managed application, a managed object may instantiate other managed objects to be its children. This relationship forms one or more managed object trees. A managed object with no parent objects is considered the root of the tree.

Refinitiv Management Classes API provides several types of managed variables. Each of these has a different set of attributes and characteristics. For more information, refer to the Refinitiv Management Classes Developer Guide.

This section uses a sample problem to explain the basics of creating a managed application.

Hereafter in this document, the term “public object” refers to “managed object” and “public variable” refers to “managed variable”.

3.2 Instrument an Application

This section explains the **main()** function of the publisher application. The purposes of the main() function of the program is to:

- Initialize a segment of shared memory
- Retrieve the statistical information of the shared memory
- Instantiate a root managed object
- Enable an event loop

3.2.1 Initializing a Segment of Shared Memory

Firstly, the managed application must acquire a segment of shared memory. To allocate a shared memory segment, the managed application instantiates an instance of **RTRServerSharedMemoryRoot** using a specified shared memory key. An instance of **RTRServerSharedMemoryRoot** can be created by:

```
#include “rtr/shrdmem.h”
RTRServerSharedMemoryRoot shm(context, name, mk, sk, s, long, m);
```

Where:

- ***context*** is the ID context.
- ***name*** is the ID name.
- ***mk*** is the shared memory key.
- ***sk*** is the semaphore key.
- ***s*** is the number of semaphores.
- ***long*** is the number of required user bytes.
- ***m*** is the maximum number of clients.

For example,

```
RTRServerSharedMemoryRoot shm(appId, "sharedmem", "99", "99", 5, 30000, 2);
```

This will allocate 30000 bytes of shared memory with “99” as the shared memory key.

If the memory associated with the given shared memory key already exists, the memory server will examine that piece of memory to determine whether or not it can safely be reinitialized. If the shared memory header matches the existing one – for example, the header has the same size, same context and same semaphore key - and it is no longer in use, the server will re-initialize the existing memory.

If the memory allocation process is unsuccessful but it has not been timed-out, the server will periodically retry the allocation process.

After a successful shared memory allocation, the server will perform a handshaking ritual with every connected client in order to verify their existence. If a client fails to acknowledge the server, the server will clean up the connection and make it available to other waiting or future clients.

To allocate storage for managed objects and managed variables of the server, **`RTRShmMOServerMemPool`** is instantiated. The object structure defined by the application is reproduced in shared memory so that it can be cloned as managed objects and variables in the client process.

For example:

```
RTRShmMOServerMemPool moPool(shm, *(shm.semaphoreSet()), 3, 10000);
```

This will allocate storage for managed objects and variables to be used by three clients, each with 10000 bytes.

If no memory (or no memory of the requested type) is available, the server will return a null pointer to the caller (usually an instance of an object or variable implementation class). It is assumed that the caller will detect this and allocate memory from the heap as necessary. The server also detects and processes parameter operations requested by clients.

3.2.2 Determining the Statistics of the Shared Memory

Once a shared memory segment is acquired, the statistics of the heap memory need to be checked as well.

There are two ways to retrieve the shared memory statistics of a shared memory segment. The first is to use the **RTRSharedMemoryStats** class directly as follows:

```
RTRSharedMemoryStats _memoryStats (
    appId,
    "MemoryStats",
    *(shm.header()),
    RTRTRUE,
    5
    );
```

This will publish the statistics with the update interval of 5 seconds.

The other approach is to use the **MyHelperAdmin** helper class instead to publish the statistics of the shared memory.

```
MyHelperAdmin admin(class_id, appId);
if( admin.error() )
{
    cerr << "publisher: Initialization Failed -Check Log-" << admin.text() << endl;
    cerr << flush;
    return 0;
}
```

MyHelperAdmin reads the arguments and extracts the relevant parameters from the given configuration file, in this case, **distribution.cnf**. It then uses **RTRTriarchProcessAdmin** to set up the shared memory distribution of the managed objects and variables. Upon encountering an error, **MyHelperAdmin** will display an error text and return.

3.2.3 Instantiating a Root Managed Object

Once the memory processing is done, developers can publish their managed objects.

```
RootObject root(appId, "Publisher", "1.0");
```

For more information about **RootObject**, please refer to the Section Section 3.3.3.

3.2.4 Enabling an Event Loop

Refinitiv Management Classes provides a class for an event-dispatching loop. The purpose is to allow application components to use system resources (I/O, timers) in a cooperative way, without being dependent on any particular implementation of a “main loop”.

```
RTRSelectNotifier::run();
```

`RTRSelectNotifier` acts as a while loop which blocks the process and waits for I/O and timer events. **`RTRSelectNotifier`** utilizes the **`select()`** system call to call back at the requested time interval, unless an I/O event occurs first.

One way to break the main loop is by interrupting the process by pressing CTRL-C. When the main loop exits, the program adds some text to the log file and then returns gracefully.

```
// RTRTermSignalHandler sigHandler;
event.setText("Terminated.");
event.log();

return 0;
```

3.3 The Publisher Application

This section provides more details of how objects, along with their variables, get published into shared memory.

3.3.1 The Structure of the Publisher Application

The structure of the Publisher application is illustrated in the following diagram:

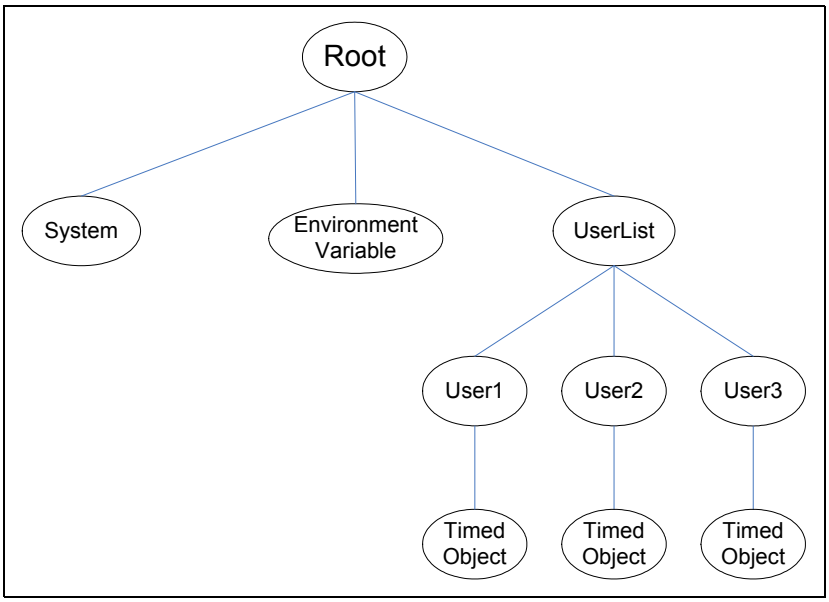

Figure 2. Publisher Application Structure

There can be a number of classes for the public objects in a managed application. Public objects with the same class identification are grouped together and are recognized to be in the same category. There are a number of classes to which the public objects are assigned in this example program. They include:

- UserList
- User
- EnvironmentVariable
- System
- TimedObject

Class identification is assigned when a public object is instantiated. For example,

```
SysInfo::SysInfo(RTRPublicObject& parent,const char* cname)
:RTRPublicObject(parent, cname, "System Information", "System",RTRManagedObject::Normal),
_cwd(*this, "Current Directory", "directory", getCurrentDirectory(), RTRFALSE)
{
}
```

This is a constructor of the **`SysInfo`** object. When an object of type **`SysInfo`** is created, it is assigned to be in the category “System”. In addition, the object is initialized to a Normal state.

3.3.2 Public Variables

Public variables are used by a publisher (a managed application) to publish manageable data. For example, a managed application may decide to publish a set of variables related to a particular socket connection (e.g. number of writes/reads, number of overflows, and number of messages sent/received). There are 11 types of public variables as listed in the following table.

VARIABLE CATEGORY	VARIABLE TYPE	C++ CLASS NAME
Boolean	Boolean	RTRPublicBoolean
Boolean	booleanConfig	RTRPublicBooleanConfig
Numeric	Counter	RTRPublicCounter
Numeric	Gauge	RTRPublicGauge
Numeric	gaugeConfig	RTRPublicGaugeConfig
Numeric	Numeric	RTRPublicNumeric
Numeric	LargeNumeric	RTRPublicLargeNumeric
Numeric	numericConfig	RTRPublicNumericConfig
Numeric	numericRange	RTRPublicNumericRange
String	String	RTRPublicString
String	stringConfig	RTRPublicStringConfig

Table 2: Types of Managed Variables

These public variables can be modified by the managing application if their options are set to be modify-enabled. However, four of them are an exception. The public variables, which include **numeric**, **largeNumeric**, **counter**, and **numericRange** types, have read-only attributes. So they are not allowed to be updated by the managing application.

3.3.3 Public Objects: RootObject

In **root.C** and **root.h**, the **RootObject** class inherits **RTRManagedProcess** which provides a minimum set of variables related to process information. These variables include:

- processID
- groupID
- userID
- time
- path
- version
- instance

RTRManagedProcess is a descendant of **RTRPublicObject**. Therefore, the public variables listed above will automatically be published into a piece of shared memory. Its constructor is as follows:

```
RTRManagedProcess(
    const RTRObjectId& appId,
    const char *execName,
    const char *subClassName,
    const char *description,
    const char *version,
    MOState startState=RTRManagedObject::Normal
    ;
```

The following figure illustrates the **Root** object.

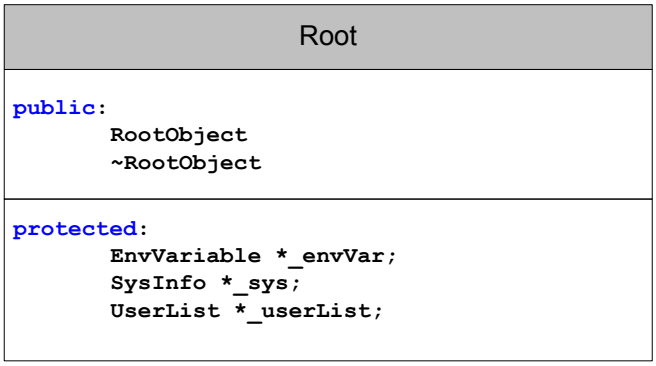

Figure 3. The Root Object

The root object starts from the Normal state. **RootObject** contains three other public objects, which are **UserList**, **SysInfo** and **EnvVariable**:

```
/* create child objects */
_userList = new UserList(*this, "userList");
_sys = new SysInfo(*this, "systemInfo");
_envVar = new EnvVariable(*this, "envVar");
```

3.3.4 Public Objects: UserList

In **userList.C** and **userList.h**, the **`UserList`** class inherits **`RTRPublicObject`** with the class identification, “UserList”. It is initialized to start from the init state with six control parameters (public variables):

- _deleteUser
- _createUser
- _goNormal
- _goRecovering
- _goWaiting
- _goDead

These variables are of **`RTRPublicBoolean`** type. In this scenario, they are represented as control parameters and are configured to be modify-enabled to allow the user (consumer application) to alter the behavior of the Publisher application.

The following figure illustrates the **`UserList`** object.

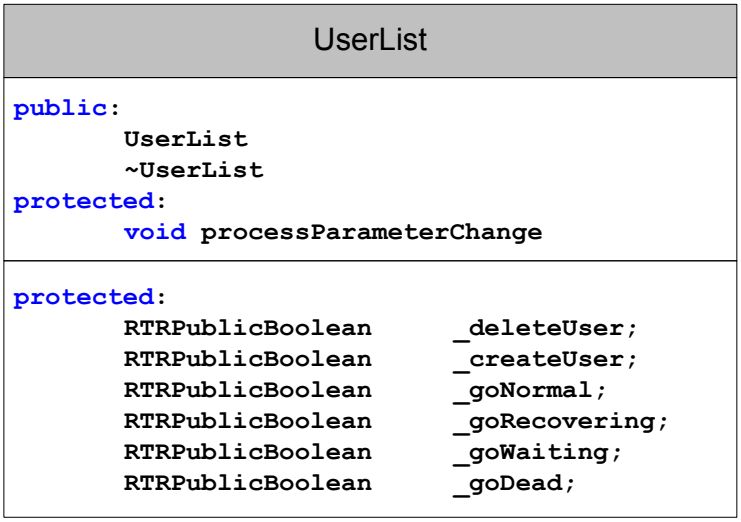

Figure 4. The UserList Object

3.3.4.1 Altering the Behaviors

When one of these control parameters changes its value, the publisher will be informed with the changed variable and the **`processParameterChange()`** function will be executed.

An example of the **`processParameterChange()`** function is:

```
void UserList::processParameterChange( RTRManagedVariable & var)
{
    RTRString text;
    RTRMgmtEvent event;

    if(var.name()=="Delete Users")
    {
        for (int i=0;i<3;i++)
        {
            if (_user[i])
            {
                delete _user[i];
```

```
                _user[i]=0;
            }
        }
        _deleteUser = RTRTRUE;
        _createUser = RTRFALSE;
    }

    if(var.name()=="Create Users")
    {
        _user[0] = new User(*this,"user1");
        _user[1] = new User(*this,"user2");
        _user[2] = new User(*this,"user3");
        _createUser = RTRTRUE;
        _deleteUser = RTRFALSE;
    }

    if(var.name()=="Set Normal State")
    {
        _goNormal = RTRTRUE;
        _goRecovering = RTRFALSE;
        _goWaiting = RTRFALSE;
        _goDead = RTRFALSE;
        markNormal("The object goes Normal");
    }

    if(var.name()=="Set Recovering State")
    {
        _goNormal = RTRFALSE;
        _goRecovering = RTRTRUE;
        _goWaiting = RTRFALSE;
        _goDead = RTRFALSE;
        markRecovering("The object goes Recovering");
    }

    if(var.name()=="Set Waiting State")
    {
        _goNormal = RTRFALSE;
        _goRecovering = RTRFALSE;
        _goWaiting = RTRTRUE;
        _goDead = RTRFALSE;
        markWaiting("The object goes Waiting");
    }

    if(var.name()=="Set Dead State")
    {
        _goNormal = RTRFALSE;
        _goRecovering = RTRFALSE;
        _goWaiting = RTRFALSE;
        _goDead = RTRTRUE;
        markDead("The object goes Dead");
```

```
    }

    text.append(var.name());
    text.append(" has been triggered.");
    event.setIdentifier(_instanceId);
    event.setSeverity("Info");
    event.setText(text);
    event.log();
}
```

In this example, the name of the public variable whose value has been updated is first checked before performing further operation. The public variables **_deleteUser** and **_createUser** are used to signal the process to delete and renew the child objects of the current object respectively.

The other four public variables are used to update the state of the public object, "**UserList**". As specified in the constructor of the **UserList** class, the **UserList** object is initialized to the normal state. **_goNormal**, **_goRecovering**, **_goWaiting**, and **_goDead** Boolean public variables are associated with the functions **markNormal**, **markRecovering**, **markWaiting**, and **markDead** from **RTRPublicObject**. Upon the execution of these functions, the state of the object is set to an appropriate value.

3.3.4.2 UserList States

The possible states of a public object are:

STATE	DESCRIPTION
Init	Initializing service
Normal	Normal service
ManualRecovery	In a service interrupted state but attempting to recover normal service manually.
AutoRecovery	In a service interrupted state but attempting to recover normal service automatically.
Dead	In an unrecoverable error state

Table 3: States of Public Objects

A public object is free to change states as requested. However, once transitioned from the **Init** state, the object cannot revert back. In addition, the **Dead** state is unrecoverable. If a parent object transitions to the **Dead** state, its child objects will transition to the **Dead** State as well.

The **UserList** object has three children, which are called **user1**, **user2**, and **user3**, as folllows:

```
_user[0] = new User(*this,"user1");
_user[1] = new User(*this,"user2");
_user[2] = new User(*this,"user3");
```

3.3.5 Public Objects: User

In **user.C** and **user.h**, the User object contains 11 public variables. Each of them represents an example of the 11 variable types as listed below.

```
protected:

    RTRPublicBoolean _bool;
    RTRPublicString _str;
    RTRPublicNumeric _num;
    RTRPublicLargeNumeric _lnum;
    RTRPublicCounter _cnt;
    RTRPublicGauge _gauge;
    RTRPublicNumericRange _numrng;
    RTRPublicStringConfig _strcfg;
    RTRPublicNumericConfig _numcfg;
    RTRPublicBooleanConfig _boolcfg;
    RTRPublicGaugeConfig _gaugecfg;
```

3.3.5.1 Altering the Behaviors

When each of these control parameters (which is modify-enabled) changes its value, the User object will be notified of the change and executes the **`processParameterChange()`** function. In this example, this function is implemented to create a log specifying which public variable has been updated as follows:

```
void User::processParameterChange( RTRManagedVariable & var)
{
    RTRString text;
    RTRMgmtEvent event;

    text.append("The value of the managed variable ");
    text.append(var.name());
    text.append(" is now ");
    text.append(var.toString());
    text.append(".");

    /* log the change of the managed variable */
    event.setIdentifier(_instanceId);
    event.setSeverity(RTRMgmtEvent::Info);
    event.setText(text);
    event.log();
}
```

The following figure illustrates the **User** object.

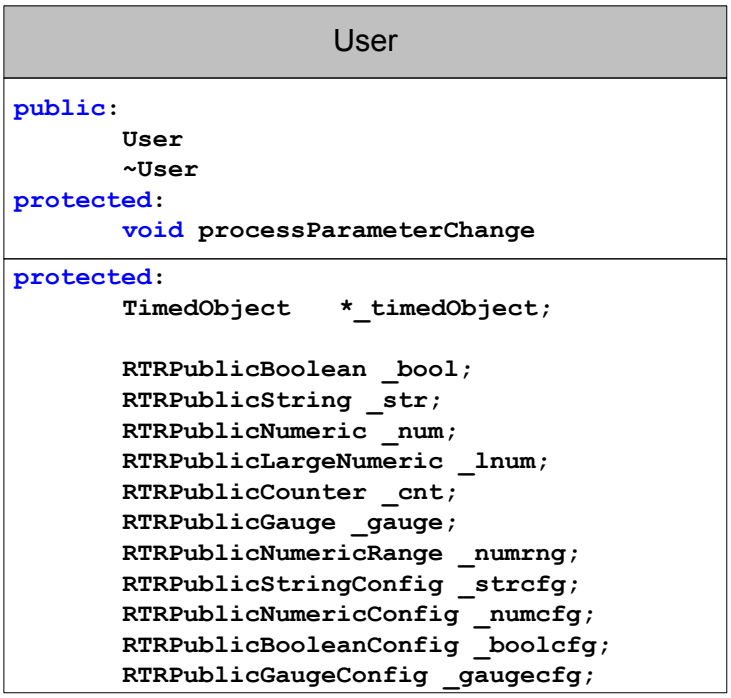

Figure 5. The User Object

3.3.5.2 Child of User

The **User** object also has a child, which is an object of class **TimedObject**.

```
protected:

    TimedObject*_timedObject;
```

3.3.6 Public Objects: TimedObject

In **timedObject.C** and **timedObject.h**, the class **TimedObject** is described. **TimedObject** inherits both **RTRPublicObject** and **RTRTimerCmd**. The following figure illustrates the **TimedObject** object.

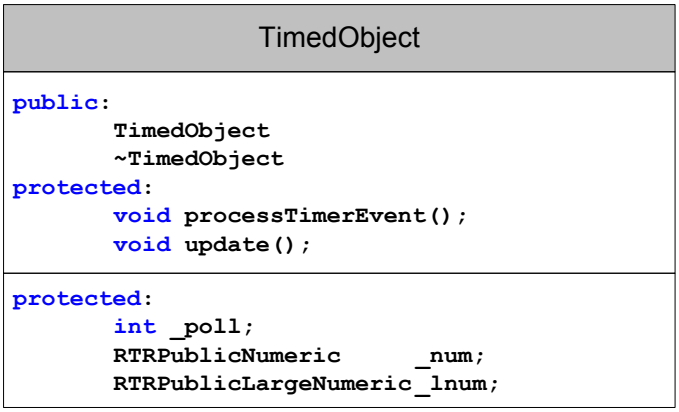

Figure 6. The TimedObject Object

```
#include "timedObject.h"

TimedObject::TimedObject(RTRPublicObject& parent, const char* cname)
    :RTRPublicObject(parent, cname, "Value Changing Object", "TimedObject", RTRManagedObject::Normal),
    _num(*this, "dynamic", "Changing value", 65),
    _lnum(*this, "large dynamic", "Changing value", 123456789101112),
    _poll(5)
{
    /* set and activate the timer to go off after _poll seconds */
    setTimerOffset(_poll, 0);
    activate();
}
```

3.3.6.1 Processing Timer Event

RTRTimerCmd is an abstract base class for components that will receive timer events. In its descendant class description, **processTimerEvent()** must be defined to perform a specific timer related task.

The function **setTimerOffset()** is used to set the polling interval of the timer. Then the timer is activated using the function activate(). Once the function **processTimerEvent()** is executed, the timer will not be automatically reactivated, to implement a continuous timer, activate() must be invoked in the implementation of **processTimerEvent()**.

An example of **processTimerEvent** that is implemented as a period timer is shown as follows:

```
void TimedObject::processTimerEvent()
{
    /* update _num and activate the timer */
    update();
    activate();
}

void TimedObject::update()
{
    if (_lnum.value() == 123456789101112)
        _lnum = 30;
    else
        _lnum = 123456789101112;

    if (_num.value() == 65)
        _num = 20;
    else
        _num = 65;
}
```

The function **update()** here is used to switch the value of the public variables **_num** and **_lnum** between 20 and 65, and between 30 and 123456789101112, respectively, at the moment a timer event is triggered.

3.3.7 Other Public Objects

There are more example classes, which extract information about the system environment variables and the current directory and publish them to the shared memory. These classes are **`SysInfo`** and **`EnvVariable`**.

```
#ifdef _WIN32
#include <windows.h>
#include <direct.h>
#else
#include <unistd.h>
#endif

#include "sysinfo.h"

char* getCurrentDirectory (void);

SysInfo::SysInfo(RTRPublicObject& parent,const char* cname)
:RTRPublicObject(parent, cname, "System Information", "System",RTRManagedObject::Normal),
_cwd(*this, "Current Directory", "directory", getCurrentDirectory(), RTRFALSE)
{
}

SysInfo::~SysInfo()
{
}

#ifdef _WIN32
char* getCurrentDirectory(void)
{
    char* buffer = new char[100];

    /* Get the current working directory: */
    if( _getcwd( buffer, 100 ) == NULL )
        perror( "_getcwd error" );
    return buffer;

}
#else
char* getCurrentDirectory(void)
{
    char *cwd;
    if ((cwd = getcwd(NULL, 200)) == NULL)
        return "NULL";
    return cwd;
}
#endif
```

The following figure illustrates the **SysInfo** object.

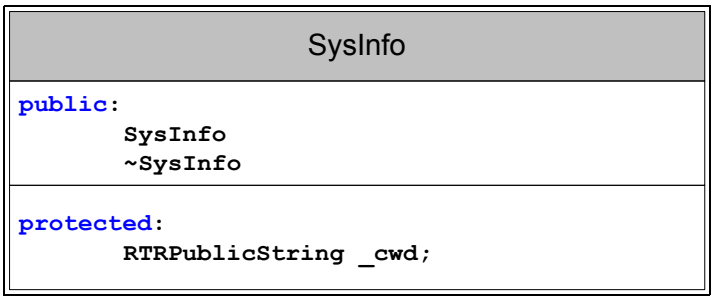

Figure 7. The SysInfo Object

The **SysInfo** class contains one public variable, which is of **RTRPublicString** type. The object uses the **getCurrentDirectory()** function to extract the information about the current working directory and then parse it to the public variable to be published in the shared memory.

The other example class is **EnvVariable**.

```
#ifdef _WIN32
#include <windows.h>
#else
#include <stdlib.h>
#endif

#include "envvar.h"

char* getEnvironment(const char* envname);

EnvVariable::EnvVariable(RTRPublicObject& parent,const char* cname)
:RTRPublicObject(parent, cname, "Environment Variables",
   "EnvironmentVariable",RTRManagedObject::Init),
_home(*this, "home", "environment variable", getEnvironment("HOME") , RTRFALSE),
_lib(*this, "lib", "environment variable", getEnvironment("LIB"), RTRFALSE),
_path(*this, "path", "environment variable", getEnvironment("PATH"), RTRFALSE),
_classpath(*this, "classpath", "environment variable", getEnvironment("CLASSPATH"), RTRFALSE)
{
}


EnvVariable::~EnvVariable()
{
}

char* getEnvironment(const char* envname)
{
    char* value;
    /* get environment variable */
    value = getenv(envname);

    if (!value)
        return "NULL";

    return value;
}
```

The following figure illustrates the **EnvVariable** class.

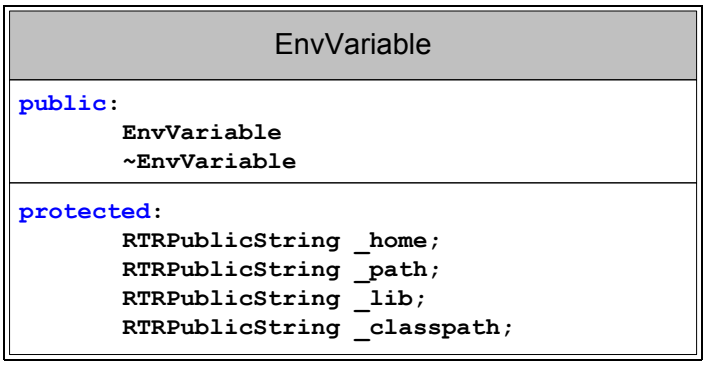

Figure 8. The EnvVariable Object

Similar to the **SysInfo** class, the **EnvVariable** class extracts the information about the environment variables and publishes them in the shared memory using the function **getEnvironment()**. It has four **RTRPublicString** variables, which represent four environment variables.

3.4 Support Utilities

Refinitiv Management Classes also provides other utility classes. Developers can utilize these classes as appropriate.

3.4.1 Command Line Processing

RTRCmdLineString is a class that provides functions to process command line arguments.

```
//-------------------------------------------------------------
// Process Command line arguments
//-------------------------------------------------------------
RTRCmdLineString inst("i", "instance", "instance #", "1", RTRCmdLineArg::False);
RTRCmdLineString cfg_file("c", "filename", "config filename", "", RTRCmdLineArg::False);

RTRCmdLine::cmdLine.resolve(argc, argv);
if ( RTRCmdLine::cmdLine.error() )
{
    RTRCmdLine::cmdLine.printUsage(cerr, argv[0]);
    return -1;
}
```

This sets the instance number of the application with the default value of one. The configuration file is also read. If the command line input from the user is invalid or the number of arguments does not match, the usage of the application is displayed.

```
RTRObjectId class_id("RMDS.PUBLISHER");
RTRApplicationId appId((const RTRString&)inst, "Publish");
```

Then the class identification and the application identification are set. The application identification is the name of the application. The class identification is used to categorize objects with the same class together in the shared memory.

3.4.2 Event logging

Refinitiv Management Classes also provides classes for event logging purposes; these are **`RTRDefaultLogger`** and **`RTRMgmtEvent`**. These classes provide the ability for components to generate events without being concerned with application context.

To log an event, these attributes must be set.

1. **`setIdentifier`**: The identifier of the component generating the event.
2. **`setText`**: A descriptive information about the event.
3. **`setSeverity`**: A severity which is a value between **`RTRMgmtEvent::Emergency`** and **`RTRMgmtEvent::Debug`**.

By default, a timestamp will be generated automatically if none is set. All events will be logged into a text file.

```
//RTRDefaultLogger logger(..);
//RTRMgmtEvent event;
event.setIdentifier(instanceId);
event.setText("a msg to be logged");
event.setSeverity("Info");
event.log();// log this message into a text file.
```

3.4.3 Configuration File Processing

In general for most applications, configuration variables are usually stored in a file. Refinitiv Management Classes provides class components to read and retrieve variables from a given configuration file. There are three steps to process a configuration file.

1. Get access to a given file from a command line or an environment variable or a default path.
2. Assign the file to a configuration database.
3. Retrieve any desire variables.

```
//--------------------------------------------------
// Configuration file
//--------------------------------------------------
// Find which config file to use:
// 1. Use file specified on command line.
// 2. Use file specified by env variable
// 3. Use default file

char ConfigDBFilename[MAX_FILENAME_LENGTH];
RTRString configFileName;
if ( cfg_file.valid() )
    configFileName = cfg_file.stringValue();
else
{
    if ( RTRGetEnv( “RMDS_CONFIG”,
                ConfigDBFilename,
                MAX_FILENAME_LENGTH ) == RTR_ENV_SUCCESS )
        configFileName = ConfigDBFilename;
    else
        configFileName = "/var/triarch/config/triarch.cnf";
}

RTRExecutablePathName cfgPath(configFileName);
```

```
ifstream configFile( configFileName, ios::in );
if ( !configFile )
{
    cerr << "Error opening config file \"" << cfgPath.path() << "\"" << endl;
    RTRCmdLine::cmdLine.printUsage(cerr, "Publisher");
    return -1;
}
configFile.close();
RTRXFileDb configDb(configFileName);
RTRConfig::setConfigDb(configDb);
cerr << "Using config file: \"" << cfgPath.path() << "\"" << endl;
```

The class **RTRConfigDb** is the abstract base class for a database of configuration variables. Variables are retrieved from the database on the basis of three pieces of information: a class identifier, an instance identifier, and a variable name. **RTRConfigDb** allows the requesting component to specify a default value, which will be used when no explicitly configured value is available. The use of default values is optional. It is a design issue. Variables for which there is no value and no default will be in an error state. For example:

```
RTRString var = RTRConfig::configDb().variable( classID, appId, "Varible", "10")
```

For more information, please refer to the *Refinitiv Management Classes Classes Reference Manual*.

4 Managing Application

4.1 Overview

This section explains an example of a managing application.

The managing application gets access to managed objects in a managed application by cloning interested objects within its process. After interested objects are cloned, the managing application registers itself as a client in order to receive notification in the form of callbacks as the information changes.

NOTE: Hereafter in this document, proxy objects and proxy variables are used to represent the cloned managed objects and managed variables on the consumer side. These proxies are created within the managing application, and mirror the objects and variables.

A managing application (which in this case is the **monitorRoots** program) attaches to a segment of shared memory by cloning a Proxy Managed Object Server Pool. In order to establish a connection with a managed application, a Proxy Managed Object Server of each shared memory segment is cloned. The managing application can monitor all root managed objects and their managed variables using the Proxy Managed Object Server. In addition, all information is displayed on the standard output device.

All the monitoring - of the Proxy Managed Object Server Pool, the Proxy Managed Object Servers, the root Proxy Managed Objects, and the Proxy Managed Variables - is done in the Monitor class.

Legend for Figure 9 and Figure 10:

- PMO: Proxy Managed Object
- PMOS: Proxy Managed Object Server
- PMOSP: Proxy Managed Object Server Pool
- PMV: Proxy Managed Variable

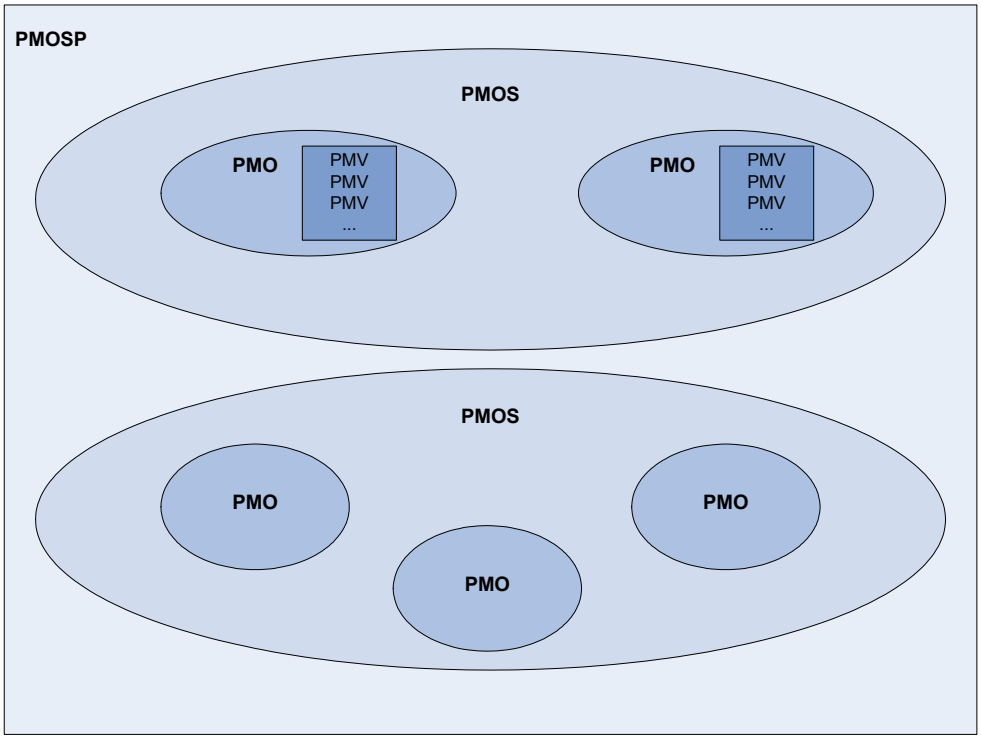

Figure 9. The Structure of Shared Memory

Figure 9 illustrates the structure of shared memory. Basically, the managing application (consumer) must become:

- a client of the Proxy Managed Object Server Pool in order to receive a notification when a server is added to or removed from the pool,
- a client of the Proxy Managed Object Server to retrieve the desired Proxy Managed Object from the Proxy Managed Object Server,
- a client of the Proxy Managed Object in order to get access to the Proxy Managed Variable of the Proxy Managed Object, and lastly
- a client of the corresponding Proxy Managed Variable to receive a notification whenever the variable's value changes.

For more information, refer to the *Refinitiv Management Classes Developer's Guide*.

This process is illustrated in Figure 10.

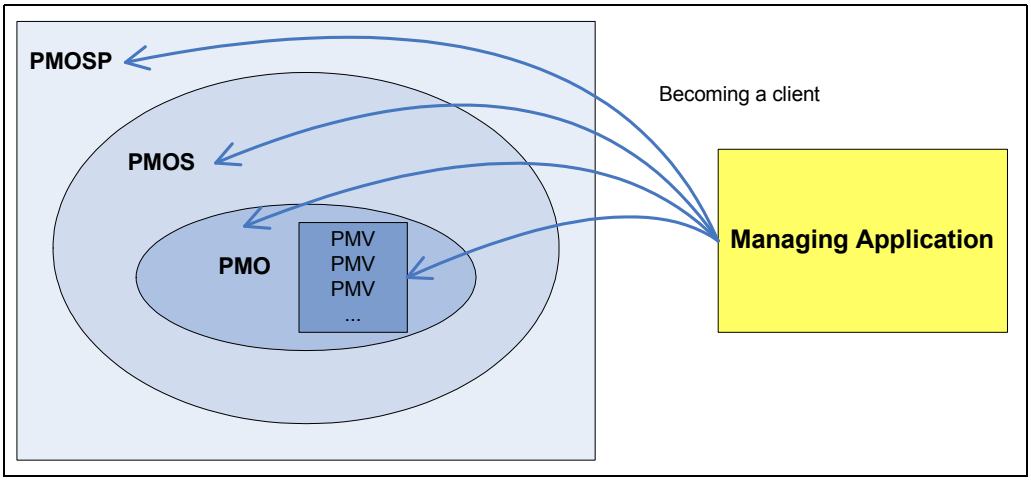

Figure 10. Process of Consumer Application

4.2 mainRoots.C

This file provides the main section of the application. Here we use the Monitor class, which is implemented in the file monitorRoots.C (see the following section).

It creates a server pool containing a number of shared memory segments with respect to the specified keys. Then it parses the server pool to the Monitor class.

```
#include <iostream>
using namespace std;

#include "rtr/selectni.h"// Event loop implementation
#include "rtr/shmpmosp.h"// RTRShmProxyManagedObjectServerPool

#include "monitorRoots.h"

int main(int argc, char **argv)
{
    // Create a base instance identifier

    // You could just pass a hard-coded literal string into the constructors below.
    // The appId approach is more elegant.

    RTRObjectId instanceId("shmApp");
    RTRShmProxyManagedObjectServerPool *pool =
            new RTRShmProxyManagedObjectServerPool(instanceId, "pool");
```

```
    char *keyPtr[] = {(char *)"81", (char *)"456", (char *)"999"};
    int len = 3;
    for (int i = 0; i < len; i++)
            pool->addServer(keyPtr[i]);
    // Monitor the server pool
    Monitor mon(*pool);

    // This application uses the select() based event loop.
    // You may choose a different implementation (e.g. Windows main loop
    // XWindows Event Notifier, you're own implementation etc.).
    RTRSelectNotifier::run();

    delete pool;

    return 0;
}
```

4.3 monitorRoots.h

This file provides the declaration for the Monitor class (in monitorRoots.C), which monitors all of the root managed objects and their managed variables.

```
#ifndef _monitorRoots_h
#define _monitorRoots_h

#include "rtr/pmosp.h"// RTRProxyManagedObjectServerPool
#include "rtr/pmospc.h"// RTRProxyManagedObjectServerPoolClient
#include "rtr/prxymos.h"// RTRProxyManagedObjectServer
#include "rtr/pmosc.h"// RTRProxyManagedObjectServerClient
#include "rtr/proxymo.h"// RTRProxyManagedObject and all Variables

/* This is a simple class which monitors (by becoming a client)
 * all of the root Managed Objects and their Managed Variables.
 * More complex examples may prefer to have separate classes
 * for each type of client and separate instances of those
 * classes for each Managed Object/Managed Variable.
 */
class Monitor :
    public RTRProxyManagedObjectServerPoolClient,
    public RTRProxyManagedObjectServerClient,
    public RTRProxyManagedObjectClient,
    public RTRProxyManagedVariableClient
{
public:
// Constructor
    Monitor(RTRProxyManagedObjectServerPool& p);

// Destructor
    ~Monitor();
```

```
// Event processing -- RTRProxyManagedObjectServerPoolClient
    // The events generated by a RTRProxyManagedObjectServerPool
    void processProxyManagedObjectServerAdded(
        RTRProxyManagedObjectServerPool& pmosp,
        RTRProxyManagedObjectServer& pmos);
    void processProxyManagedObjectServerRemoved(
        RTRProxyManagedObjectServerPool& pmosp,
        RTRProxyManagedObjectServer& pmos);

// Event processing -- RTRProxyManagedObjectServerClient
    // The events generated by a RTRProxyManagedObjectServer
    void processObjectServerError(
        RTRProxyManagedObjectServer& pmos);
    void processObjectServerSync(
        RTRProxyManagedObjectServer& pmos);
    void processObjectServerRootAdded(
        RTRProxyManagedObjectServer& pmos,
                const RTRProxyManagedObjectHandle& pmoh);
    void processObjectServerRootRemoved(
        RTRProxyManagedObjectServer& pmos,
                const RTRProxyManagedObjectHandle& pmoh);

// Event processing -- RTRProxyManagedObjectClient
    // The events generated by a RTRProxyManagedObject
    void processProxyManagedObjectError(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectSync(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectDeleted(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectInfo(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectInService(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectRecovering(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectWaiting(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectDead(
        const RTRProxyManagedObject& pmo);
    void processProxyManagedObjectChildAdded(
        const RTRProxyManagedObject& pmo,
            const RTRProxyManagedObjectHandle& pmoh);
    void processProxyManagedObjectChildRemoved(
        const RTRProxyManagedObject& pmo,
            const RTRProxyManagedObjectHandle& pmoh);
    void processProxyManagedObjectVariableAdded(
        const RTRProxyManagedObject& pmo,
            const RTRProxyManagedVariableHandle& pmoh);
    void processProxyManagedObjectVariableRemoved(
```

```
        const RTRProxyManagedObject& pmo,
            const RTRProxyManagedVariableHandle& pmoh);

// Event processing -- RTRProxyManagedVariableClient
    // The events generated by a RTRProxyManagedVariable
    void processProxyManagedVariableError(RTRProxyManagedVariable& pmv);
    void processProxyManagedVariableSync(RTRProxyManagedVariable& pmv);
    void processProxyManagedVariableUpdate(RTRProxyManagedVariable& pmv);
    void processProxyManagedVariableDeleted(RTRProxyManagedVariable& pmv);

private:
// Utilities
    void showPMVdata(RTRProxyManagedVariable& pmv);

    void showBoolean_data(RTRProxyManagedVariable& pmv);
    void showBooleanConfig_data(RTRProxyManagedVariable& pmv);
    void showCounter_data(RTRProxyManagedVariable& pmv);
    void showNumeric_data(RTRProxyManagedVariable& pmv);
    void showLargeNumeric_data(RTRProxyManagedVariable& pmv);
    void showNumericConfig_data(RTRProxyManagedVariable& pmv);
    void showNumericRange_data(RTRProxyManagedVariable& pmv);
    void showGauge_data(RTRProxyManagedVariable& pmv);
    void showGaugeConfig_data(RTRProxyManagedVariable& pmv);
    void showString_data(RTRProxyManagedVariable& pmv);
    void showStringConfig_data(RTRProxyManagedVariable& pmv);

    void addToList( const RTRProxyManagedObjectPtr obj );
    void addToList( const RTRProxyManagedVariablePtr var );

// Data
    RTRProxyManagedObjectServerPool& _pmosp;
    // Maintain a reference (Smart pointer) to all of the
    // root managed objects and managed variables so that
    // they will not be deleted (garbage collected).
    RTRLinkedList<RTRProxyManagedObjectPtr> _pmoList;
    RTRLinkedList<RTRProxyManagedVariablePtr> _pmvList;
};

#endif
```

4.4 monitorRoots.C

This file provides the implementation (Monitor class) for monitoring all of the root managed objects and their managed variables.

```
#include <iostream>
using namespace std;

#include "monitorRoots.h"


Monitor::Monitor( RTRProxyManagedObjectServerPool& p )
        : _pmosp(p)
{
// Register with the Server Pool to receive events.
```

```
_pmosp.addClient(*this);
    RTRLinkedListCursor<RTRProxyManagedObjectServer> iter = _pmosp.servers();
    for (iter.start(); !iter.off(); iter.forth())
        processProxyManagedObjectServerAdded(_pmosp, (RTRProxyManagedObjectServer&)*iter);
    //_pmosp.addClient(*this);
}

Monitor::~Monitor()
{
    // Be sure to drop client
    if ( _pmosp.hasClient(*this) )
        _pmosp.dropClient(*this);
}

// Event processing for RTRProxyManagedObjectServerPoolClient
void Monitor::processProxyManagedObjectServerAdded(
            RTRProxyManagedObjectServerPool& pmosp,
            RTRProxyManagedObjectServer& pmos
            )
{
    cout << "pmosp event: Added a pmos  @" << pmos.text() << endl;
    // Register with each Server to receive events
    if(pmos.inSync() == RTRTRUE)
    {
        cout << "pmos is inSync" << endl;
        processObjectServerSync(pmos);
    }
    else
    {
        cout << "pmos is not inSync" << endl;
    }
    pmos.addClient(*this);
}

void Monitor::processProxyManagedObjectServerRemoved(
            RTRProxyManagedObjectServerPool& pmosp,
            RTRProxyManagedObjectServer& pmos
            )
{
    cout << "pmosp event: Removed a pmos  @" << pmos.text() << endl;
    // I know that I'm a client (since I received this event)
    // so be sure to drop client when the Server is no longer valid.
    pmos.dropClient(*this);
}

//event processing for RTRProxyManagedObjectServerClient
void Monitor::processObjectServerError(
            RTRProxyManagedObjectServer& pmos
            )
{
    cout << "pmos event: Error  @" << pmos.text() << endl;
    // Drop client since this Server is no longer valid.
    pmos.dropClient(*this);
}

void Monitor::processObjectServerSync(
            RTRProxyManagedObjectServer& pmos
```

```
            )
{
    cout << "pmos event: in Sync  @" << pmos.text() << endl;

    // Now that the Server is in Sync I can iterate through all Root Proxy Managed Objects in pmos
    // for each root Proxy Managed Object I will become its client.
    RTRProxyManagedObjectHandleIterator pmosIterator = pmos.roots();
    for ( pmosIterator.start(); !pmosIterator.off(); pmosIterator.forth() )
    {
        // pmosIterator.item() is the handle to the current mo
        cout << "found handle to an mo: " << endl
             << "     instance identifier is " << pmosIterator.item().instanceId() << endl
             << "     name is " << pmosIterator.item().name() << endl
             << "     class identifier is " << pmosIterator.item().classId() << endl;

        //now clone the current Proxy Managed Object
        RTRProxyManagedObjectPtr pmoPtr = pmos.object(pmosIterator.item());

        // Maintain a smart pointer reference to the Proxy Managed Object
        // or else the object will be garbage collected.
        addToList(pmoPtr);

        pmoPtr->addClient( (RTRProxyManagedObjectClient &) *this );
        /* If the Object is already inSync() then I will not receive
        * the 'Sync' event, so I need to check it here.
        */
        if ( pmoPtr->inSync() == RTRTRUE )
        {
            cout << "pmoPtr is inSync" << endl;
            processProxyManagedObjectSync(*pmoPtr);
        }
        else
        {
            cout << "pmoPtr is not inSync" << endl;
        }
    }
}

void Monitor::processObjectServerRootAdded(
            RTRProxyManagedObjectServer& pmos,
            const RTRProxyManagedObjectHandle& pmoH
            )
{
    cout <<"pmos event: Added a root on pmos  @" << pmos.text() << endl;

    //now clone the current Proxy Managed Object
        RTRProxyManagedObjectPtr pmoPtr = pmos.object(pmoH);

        // Maintain a smart pointer reference to the Proxy Managed Object
        // or else the object will be garbage collected.
        addToList(pmoPtr);

        pmoPtr->addClient( (RTRProxyManagedObjectClient &) *this );
        /* If the Object is already inSync() then I will not receive
        * the 'Sync' event, so I need to check it here.
```

```
        */
        if ( pmoPtr->inSync() == RTRTRUE )
        {
            cout << "pmoPtr is inSync" << endl;
            processProxyManagedObjectSync(*pmoPtr);
        }
        else
        {
            cout << "pmoPtr is not inSync" << endl;
        }

}

void Monitor::processObjectServerRootRemoved(
            RTRProxyManagedObjectServer& pmos,
            const RTRProxyManagedObjectHandle& pmoh
            )
{
    cout <<"pmos event: Removed a root from pmos  @" << pmos.text() << endl;
}

//
// Event processing for RTRProxyManagedObjectClient
//
void Monitor::processProxyManagedObjectError(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: Error  @" << pmo.text() << endl;
}

void Monitor::processProxyManagedObjectSync(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: Sync" << endl;

    //iterate through all Proxy Managed Variables
    RTRProxyManagedVarHandleIterator pmvIterator = pmo.variableHandles();
    for ( pmvIterator.start(); !pmvIterator.off(); pmvIterator.forth() )
    {
        cout << " found a variable of type   " << pmvIterator.item().typeString() << endl
        << " with name " << pmvIterator.item().name() << endl;

        //clone this Proxy Manged Variable
        RTRProxyManagedVariablePtr pmvPtr = pmo.variableByName(pmvIterator.item().name());

        if (pmvPtr->error())
            return;

        //add to pmvList (so there will be no garabage collection with pmvPtr)
        addToList(pmvPtr);

        pmvPtr->addClient(*this);
```

```
        /* If the Variable is already inSync() then I will not receive
        * the 'Sync' event, so I need to check it here.
        */
        if ( pmvPtr->inSync() == RTRTRUE )
        {
            cout << "pmvPtr is inSync" << endl;
            processProxyManagedVariableSync(*pmvPtr);
        }
        else
        {
            cout << "pmvPtr is not inSync" << endl;
        }
    }
}

void Monitor::processProxyManagedObjectDeleted(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: Deleted" << endl;
}

void Monitor::processProxyManagedObjectInfo(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: Info   @" << pmo.text() << endl;
}

void Monitor::processProxyManagedObjectInService(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: InService" << endl;
}

void Monitor::processProxyManagedObjectRecovering(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: Recovering" << endl;
}

void Monitor::processProxyManagedObjectWaiting(
            const RTRProxyManagedObject& pmo
            )
{
    cout << "pmo event: Waiting" << endl;
}

void Monitor::processProxyManagedObjectDead(
            const RTRProxyManagedObject& pmo
            )
{
```

```
    cout << "pmo event: Dead" << endl;
}

void Monitor::processProxyManagedObjectChildAdded(
            const RTRProxyManagedObject& pmo,
            const RTRProxyManagedObjectHandle& pmoh
            )
{
    cout << "pmo event: ChildAdded" << endl;
}

void Monitor::processProxyManagedObjectChildRemoved(
            const RTRProxyManagedObject& pmo,
            const RTRProxyManagedObjectHandle& pmoh
            )
{
    cout << "pmo event: ChildRemoved" << endl;
}

void Monitor::processProxyManagedObjectVariableAdded(
            const RTRProxyManagedObject& pmo,
            const RTRProxyManagedVariableHandle& pmoh
            )
{
    cout << "pmo event: VariableAdded" << endl;
}

void Monitor::processProxyManagedObjectVariableRemoved(
            const RTRProxyManagedObject& pmo,
            const RTRProxyManagedVariableHandle& pmoh
            )
{
    cout << "pmo event: VariableRemoved" << endl;
}

//Event processing for RTRProxyManagedVariableClient
void Monitor::processProxyManagedVariableError(
            RTRProxyManagedVariable& pmv
            )
{
    cout << "pmv event: Error  @" << pmv.text() << endl;
}

void Monitor::processProxyManagedVariableSync(
            RTRProxyManagedVariable& pmv
            )
{
    cout << "pmv event: Sync" << endl;
    cout << "cloned mv information:   name        = " << pmv.name() << endl;
    cout << "                         type        = " << pmv.typeString() << endl;
    cout << "                         description = " << pmv.description() << endl;

    //show variable specific values
    showPMVdata(pmv);
}

void Monitor::processProxyManagedVariableUpdate(
```

```
            RTRProxyManagedVariable& pmv
            )
{
    cout << "pmv event: Update" << endl;

    cout << "new value of " << pmv.name()
         << " (with instance Id " << pmv.context().instanceId() << ") is "
         << pmv.toString() << endl;
}

void Monitor::processProxyManagedVariableDeleted(
            RTRProxyManagedVariable& pmv
            )
{
    cout << "pmv event: Deleted" << endl;
    pmv.dropClient(*this);
}

// showPMVdata calls the specific variable member function, which shows some specific variable data
void Monitor::showPMVdata(
            RTRProxyManagedVariable& pmv
            )
{
    switch ( pmv.type() )
    {
        case RTRProxyManagedVariableHandle::Boolean:
            showBoolean_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::BooleanConfig:
            showBooleanConfig_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::Counter:
            showCounter_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::Numeric:
            showNumeric_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::LargeNumeric:
            showLargeNumeric_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::NumericConfig:
            showNumericConfig_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::NumericRange:
            showNumericRange_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::Gauge:
            showGauge_data(pmv);
            break;
```

```
        case RTRProxyManagedVariableHandle::GaugeConfig:
            showGaugeConfig_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::String:
            showString_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::StringConfig:
            showStringConfig_data(pmv);
            break;

        case RTRProxyManagedVariableHandle::Invalid:
            cout << "Invalid Proxy Managed Variable" << endl;
            break;

        default:
            cout << "Unknown Proxy Managed Variable" << endl;
    }
}


void Monitor::showBoolean_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedBoolean& pmb = (RTRProxyManagedBoolean&)pmv;
    if ( pmb.value() == RTRTRUE )
        cout << " value = RTRTRUE" << endl;
    else
        cout << " value = RTRFALSE" << endl;
}

void Monitor::showBooleanConfig_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedBooleanConfig& pmbc = (RTRProxyManagedBooleanConfig&)pmv;
    if ( pmbc.activeValue() == RTRTRUE )
        cout << " active value = RTRTRUE" << endl;
    else
        cout << " active value = RTRFALSE" << endl;
    if ( pmbc.storeValue() == RTRTRUE )
        cout << " store value  = RTRTRUE" << endl;
    else
        cout << " store value  = RTRFALSE" << endl;
    if ( pmbc.factoryDefault() == RTRTRUE )
        cout << " factory default = RTRTRUE" << endl;
    else
        cout << " factory default = RTRFALSE" << endl;
}

void Monitor::showCounter_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedCounter& pmc = (RTRProxyManagedCounter&)pmv;
    unsigned long cnt = pmc.value();
    cout << " value       = " << cnt << endl;
}

void Monitor::showNumeric_data(RTRProxyManagedVariable& pmv)
{
```

```
    RTRProxyManagedNumeric& pmn = (RTRProxyManagedNumeric&)pmv;
    cout << " value = " << pmn.value() << endl;
}

void Monitor::showLargeNumeric_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedLargeNumeric& pmln = (RTRProxyManagedLargeNumeric&)pmv;

    #if defined (_WIN32) || defined (_WIN64)
        char buf[30];//There is no operator << for __int64 type defined for the ostream class.
        sprintf(buf, "%I64d", pmln.value());
        cout << " value = " << buf << endl;
    #else
        cout << " value = " << pmln.value() << endl;
    #endif

}

void Monitor::showNumericConfig_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedNumericConfig& pmnc = (RTRProxyManagedNumericConfig&)pmv;
    cout << " active value = " << pmnc.activeValue() << endl
         << " min value    = " << pmnc.minValue() << endl
         << " max value    = " << pmnc.maxValue() << endl
         << " store value  = " << pmnc.storeValue() << endl
         << " factory default = " << pmnc.factoryDefault() << endl;
}

void Monitor::showNumericRange_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedNumericRange& pmnr = (RTRProxyManagedNumericRange&)pmv;
    cout << " min value   = " << pmnr.minValue() << endl
         << " max value   = " << pmnr.maxValue() << endl;
}

void Monitor::showGauge_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedGauge& pmg = (RTRProxyManagedGauge&)pmv;
    cout << " min value   = " << pmg.minValue() << endl
         << " max value   = " << pmg.maxValue() << endl
         << " lowWaterMark  = " << pmg.lowWaterMark() << endl
         << " highWaterMark = " << pmg.highWaterMark() << endl;
}

void Monitor::showGaugeConfig_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedGaugeConfig& pmgc = (RTRProxyManagedGaugeConfig&)pmv;
    cout << " min store value     = " << pmgc.minStoreValue() << endl
         << " min factory default = " << pmgc.minFactoryDefault() << endl
         << " max store value     = " << pmgc.maxStoreValue() << endl
         << " max factory default = " << pmgc.maxFactoryDefault() << endl;
}

void Monitor::showString_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedString& pms = (RTRProxyManagedString&)pmv;
    cout << " value = " << pms.value() << endl;
```

```
}

void Monitor::showStringConfig_data(RTRProxyManagedVariable& pmv)
{
    RTRProxyManagedStringConfig& pmsc = (RTRProxyManagedStringConfig&)pmv;
    RTRString active = pmsc.activeValue();
    cout << " active value = " << pmsc.activeValue() << endl;
    RTRString store = pmsc.storeValue();
    cout << " store value  = " << pmsc.storeValue() << endl;
    RTRString fDflt = pmsc.factoryDefault();
    cout << " factory default = " << pmsc.factoryDefault() << endl;
}

//create a linked list that contains
//all the proxy managed objects and their children
void Monitor::addToList( const RTRProxyManagedObjectPtr obj )
{
    RTRProxyManagedObjectPtr objPtr;
    RTRBOOL done = RTRFALSE;  //used to maintain correct managed object tree structure

    RTRProxyManagedObjectPtr *nPtr =  new RTRProxyManagedObjectPtr;
    *nPtr = obj;
    if ( _pmoList.empty() )
    {
        _pmoList.addRight( nPtr );
    }
    else
    {
        for ( _pmoList.start(); !_pmoList.off(); _pmoList.forth() )
        {
            objPtr = *(_pmoList.item());

            if ( obj->instanceId().parent() == objPtr->instanceId() )
            {
                _pmoList.addRight( nPtr );
                done = RTRTRUE;
                break;
            }
        }
        if ( !done )
        {
            _pmoList.start();
            _pmoList.addRight( nPtr );
        }
    }
}

//create a linked list that contains
//all the proxy managed variables
void Monitor::addToList( const RTRProxyManagedVariablePtr var )
{
    RTRProxyManagedVariablePtr   ptr;
    RTRBOOL variable_found = RTRFALSE;  //used to ensure that the variables
        //added in the same order as we see them

    RTRProxyManagedVariablePtr *nPtr =  new RTRProxyManagedVariablePtr;
    *nPtr = var;
```

```
    if ( _pmvList.empty() )
    {
        _pmvList.addRight( nPtr );
    }
    else
    {
        for ( _pmvList.start(); !_pmvList.off(); _pmvList.forth() )
        {
            ptr = *(_pmvList.item());
        if (ptr->error()) {
            if (ptr->hasClient(*this))
                ptr->dropClient(*this);
            _pmvList.remove();
            return;
            }

        if ( var->context().instanceId() == ptr->context().instanceId() )
            variable_found = RTRTRUE;

        if ( ( variable_found ) &&
            (var->context().instanceId() != ptr->context().instanceId()) )
            {
            _pmvList.addLeft( nPtr );
            return;
            }
        }

        // New variable added to list
        _pmvList.extend( nPtr );
    }
}
```

Document ID: RMC220UMTUT.200
Date of issue: December 2020