



# Design of an MROI System

Allen Farris

September 13, 2009

Version: 0.9

---

Document: **INT-409-ENG-0020 rev 0.9**  
Work package: **WP 4.09.03**  
System: **MROI Supervisory System**

Approved by: **Allen Farris**  
**MROI**

**Team Lead, Software and Control Systems**

*Magdalena Ridge Observatory  
New Mexico Tech  
801 Leroy Place  
Socorro, NM 87801 USA  
<http://www.mro.nmt.edu>*

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>System Structure</b>	<b>6</b>
2.1	A Basic System . . . . .	9
2.2	An Asynchronous System . . . . .	12
2.3	A Monitoring System . . . . .	13
2.4	The State Model . . . . .	14
<b>3</b>	<b>Communications Protocol</b>	<b>17</b>
3.1	Encoding and decoding types of data . . . . .	21
3.1.1	Methods for encoding data . . . . .	22
3.1.2	Methods for decoding data . . . . .	23
<b>4</b>	<b>Defining a High-level System Interface</b>	<b>24</b>
4.1	System Worksheet . . . . .	24
4.1.1	Columns in The System worksheet . . . . .	24
4.2	Monitor Worksheet . . . . .	26
4.2.1	Columns in the Monitor worksheet . . . . .	26
4.3	Fault Worksheet . . . . .	29
4.3.1	Columns in the Fault worksheet . . . . .	29
4.4	Control Worksheet . . . . .	29
4.4.1	Columns in the Control worksheet . . . . .	29
4.5	Parameters Worksheet . . . . .	31
4.5.1	Columns in the Parameters worksheet . . . . .	31
4.6	The Code Generation Framework . . . . .	33
<b>5</b>	<b>Change History</b>	<b>35</b>
5.1	Version 0.9 . . . . .	35
<b>6</b>	<b>Additional information</b>	<b>36</b>

# 1 Introduction

The overview document on the Supervisory System (see Reference [1]) focused on the nature and structure of the MROI Supervisory System. This document focuses on an application system that is managed by the Supervisory System. In subsequent sections we will discuss:

- How such a system is structured,
- Its high-level definition,
- Its nature as a server and how it interacts with remote clients,
- How it interacts with key elements of the Supervisory System, and
- The communications protocol it uses.

Before tackling these issues, we need to have a better grasp of exactly what a system is.

The MROI Interferometer consists of a collection of systems that are managed by the Supervisory System. Each instance of a system interacts with the MROI Interferometer as a whole in specific ways that are defined by that system.

The Supervisory System itself is a collection of modules: one Executive, Database Manager, and Operator Interface; one or more Data Collectors; and, one or more Supervisors, each of which creates its own Fault Manager.

An MROI application system managed by the Supervisory System lives in a complex environment, which is depicted in Figure 1. The system is of a specific type and is uniquely identified, within the MROI network, by a specific name assigned to that system instance. The system has a state at all times; it implements a well-defined state model. The system has a log file, on which it publishes messages related to internal events that occur within its lifetime. The system publishes monitor data which is produced during its execution. It also is able to communicate with other remote servers: the Database Manager, Fault Manager, and Telescope Operator. Using these connections, a System is able to ask the Database Manager for initialization data, for example, publish faults and alerts, and, if necessary, send messages directly to the telescope operator. The system, in general, functions as a server, i.e., it responds to commands sent by one or more remote clients, but, more specifically, it functions under the control of a Supervisor. All of this functionality is implemented by a suite of classes that are used in defining an MROI application system.

A system in this context is any collection of hardware and software modules that function as a unified whole, i.e. has a well-defined set of monitor points and control commands. Such a system may be a collection of other systems; so, a system can have a hierarchical structure. The Supervisory System, Figure 2, is a collection of systems. The Environmental Monitoring System, Figure 3, is a collection of weather stations and other hardware, each of which is itself a system. Note also that both the Supervisory System and Environmental Monitoring System are extensions of a ControlSystem, which is part of the suite of classes mentioned above. From the point of view of the Supervisory System, a system is any software module that is an extension of the classes that implement the complex MROI environment described above.

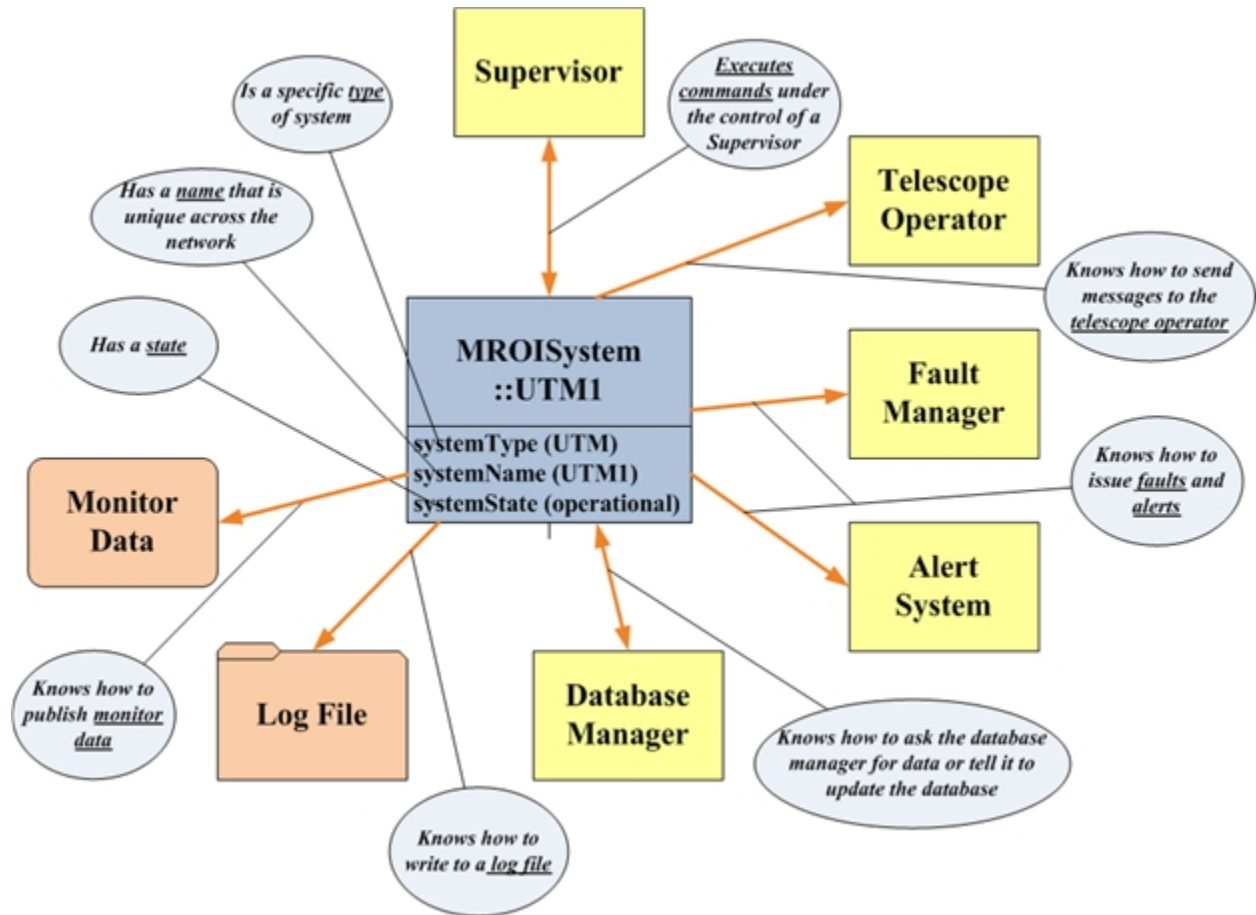


Figure 1: An MROI System

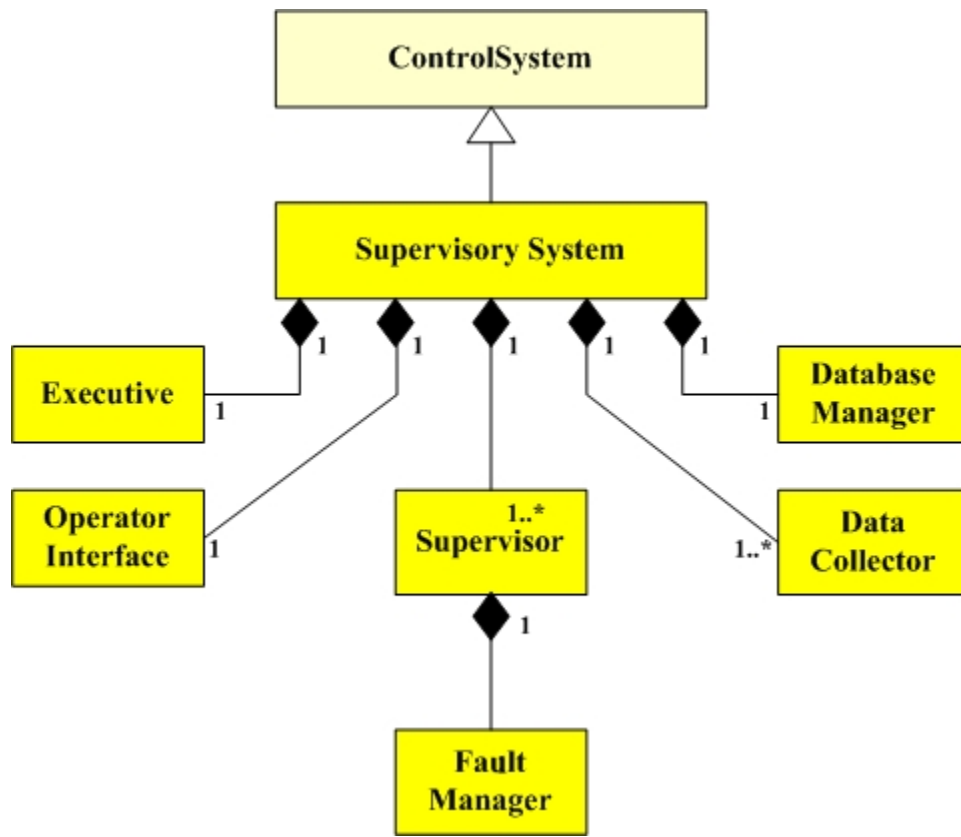


Figure 2: [The Supervisory System](#)

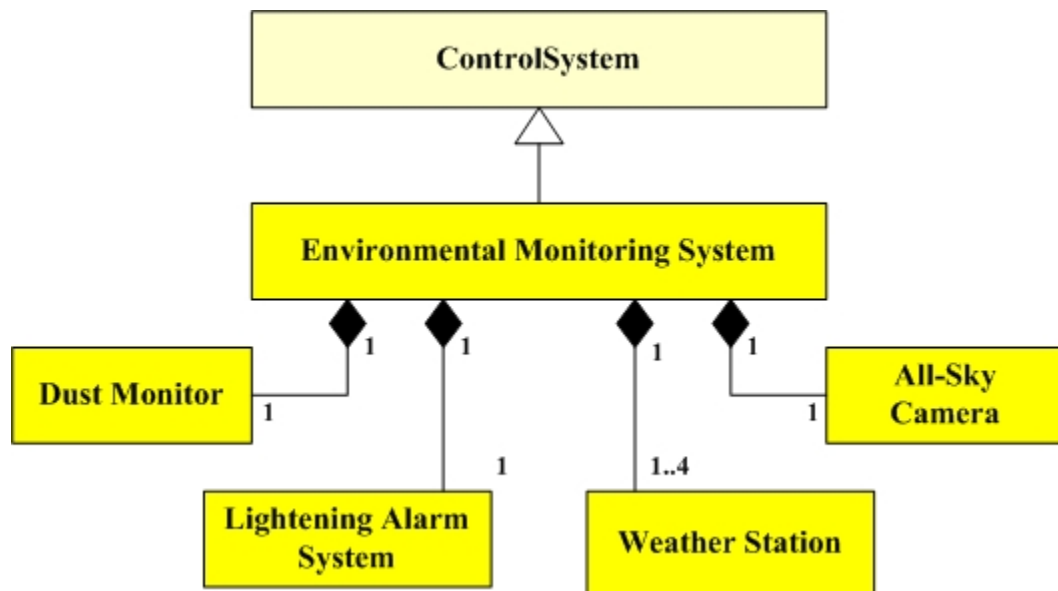


Figure 3: [The Environmental Monitoring System](#)

## 2 System Structure

MROI application systems are diverse. Frequently they have demanding real-time requirements and are capable of generating data at high rates. In addition, because some of them are supplied by external vendors, their implementation techniques vary considerably. Nevertheless, they do have common characteristics. The software described in this section is designed to exploit those common features. Its purpose is to aid in building systems that are being designed and developed in-house and to streamline the process of integrating all application systems into a coherent, distributed network managed by the Supervisory System.

The classes described here and shown in Figure 4 are designed to implement what is common to MROI application systems. There are three types of systems in increasing levels of complexity. We will refer to these three types as basic, asynchronous, and monitoring systems. Actual MROI systems are extensions of one of these three types.

The basic system classes (shown in yellow in Figure 4) implement the environment described in Figure 1. All commands submitted by a client to this type of system are synchronous commands; if the command is expected to return some item of data or a complex object, the client is blocked until the request is processed and that item of data or object is returned. An asynchronous system (shown in light blue in Figure 4) is itself an extension of a basic system that adds the ability to process client requests asynchronously. An asynchronous command returns a response immediately but returns the item of data or object of interest at some later time; the client is not blocked while waiting for the returned data. Asynchronous methods create a new thread of execution to process the command and return the object of interest whenever that processing is complete. The reason these are broken into two sets of classes is that asynchronous commands introduce a great deal of complexity, and systems that do not need such functionality need not be burdened with this additional complexity. The third type of system, a monitoring system, is an extension of an asynchronous system and adds the ability to publish data by periodically monitoring the status of internal states, measurements by various sensors, images from detectors, etc. The classes to support these activities are shown in Figure 4 as tan colored.

All of the classes to support basic, asynchronous, and monitoring systems, described above (the yellow, light blue, and tan classes), are used by application systems that are servers to facilitate communications with remote clients. Those remote clients use “proxy” objects within their domain of execution to communicate with the remote systems. Corresponding to the three types of systems, there are three types of proxies: basic, asynchronous, and monitoring proxies (shown in green in Figure 4). Again, a specific system proxy is an extension of one of these proxies. Figure 5 shows a weather station. The weather station system is a monitoring system, so it is extended from the MonitoringSystem class; likewise its proxy, WeatherStationProxy, is extended from the MonitoringSystemProxy class. The additional classes in this diagram will be explained later.

Systems, regardless of whether they are basic, asynchronous, or monitoring, are not required to return an item of data or an object of any kind in response to client commands. Those commands may merely initiate some required action. They all may, however, return an exception, if there is an error of some kind. Synchronous commands, including those that do

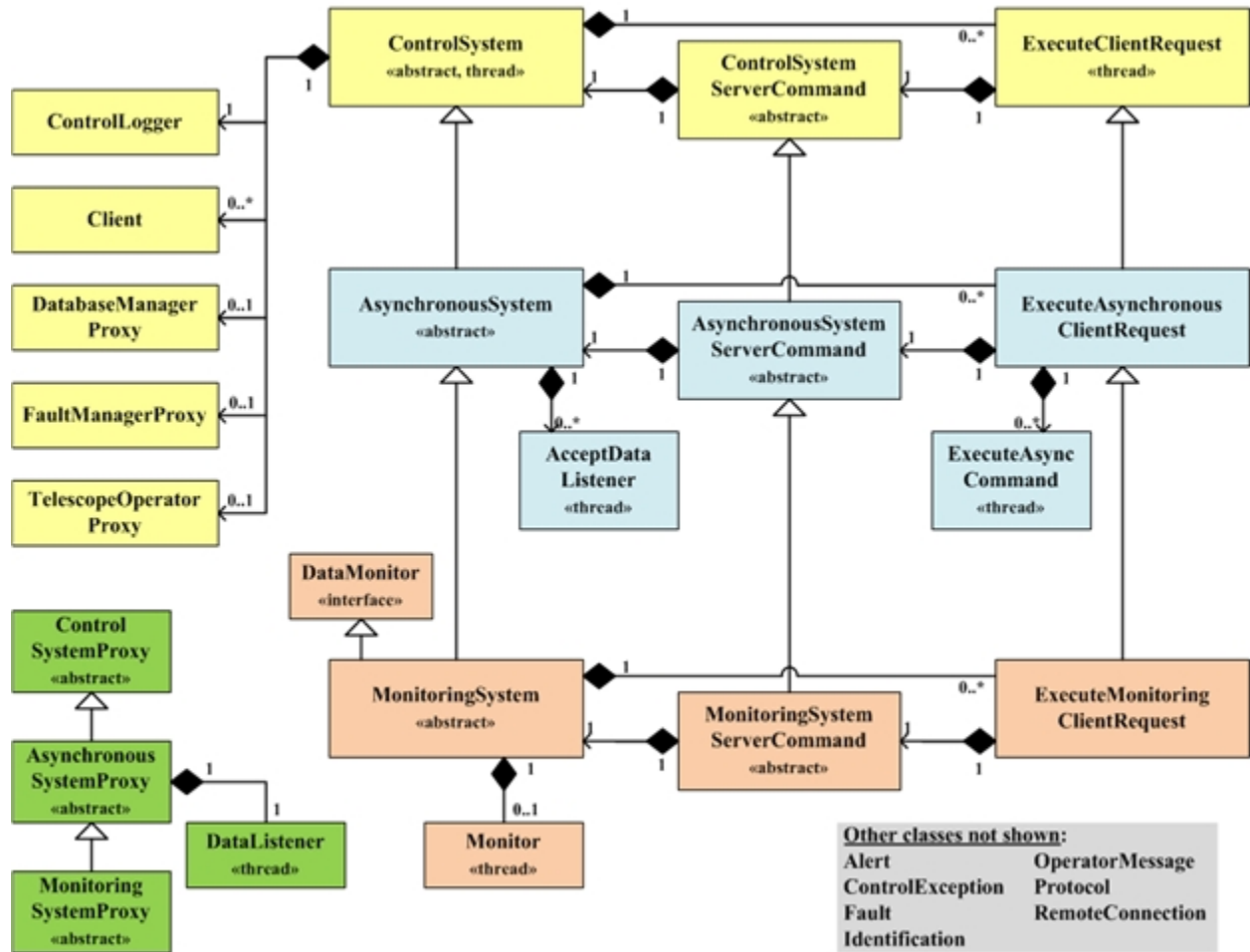


Figure 4: Classes for Basic, Asynchronous, and Monitoring Systems

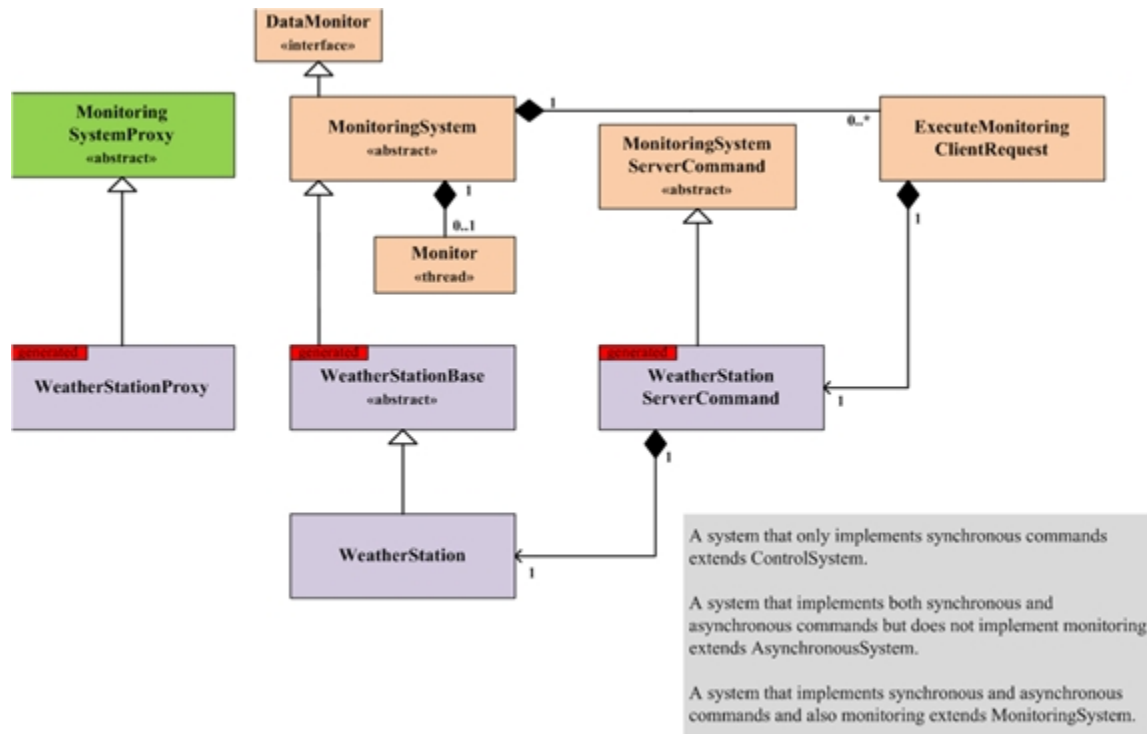


Figure 5: **A Weather Station as a System**

not return any data, will return an exception to the requesting client, if the command was not accepted (if some parameter has an incorrect value, for example) or if the processing ended in an error state. Asynchronous commands always return something immediately: a signal that indicates the command was valid and accepted, or an exception that indicates the command was not accepted because of some error in the command. Later, if the asynchronous command processing ends in an error state, an exception is returned to the client instead of the expected data. If an asynchronous command returns nothing (return type is 'void'), when the processing is complete, the client is sent a message informing that the command has been completed.

The main goal of these classes may be viewed by an application system in the following manner. Their purpose, and the method of defining a high-level application interface, is to allow an application system to merely implement its functional methods, without having to worry about all of the technical details of how a remote client interacts with the system to execute commands and receive their results. As Figure 5 shows, the key classes implementing the communications protocol are automatically generated from the definition of the high-level interface. This leaves an application free to do what it knows how to do best, viz. implement the complex methods that make it function successfully.

Finally, an additional point needs to be made about an application system. The method of defining a high-level application interface outlined here supports the idea of a system that is hierarchically structured. The parts of such a system are themselves systems, i.e. they are extensions of basic, asynchronous, or monitoring classes. However, a complex application system such as the Unit Telescope Mount or the Delay Line System is not required to use



this approach. It is entirely free to implement its internal structure as it sees fit. All that is required is to implement a high-level interface, conforming to the protocols defined here, that allows that system to be managed by the Supervisory System. With vendor supplied software systems such as the Unit Telescope Mount, whose contract was negotiated before this protocol was defined, we will implement an adapter layer that will translate the vendor supplied protocol to the protocol defined here.

## 2.1 A Basic System

There are three significant classes (see Figure 4) that implement the functionality needed by a basic system: the `ControlSystem` class, the `ExecuteClientRequest` class, and `ControlSystemServerCommand` class. The `ControlSystem` class is the most important of these. It implements the following items:

- Defines a system type, instance name, and package name,
- Creates a log file and logger associated with this system,
- Using an assigned port on the host on which it is being executed it creates a server socket on which it listens for connections by remote clients,
- Has the ability to connect to the Telescope Operator, Database Manager, and Fault Manager,
- Maintains a list of current connections to any other remote clients,
- Maintains a list of clients currently connected to this server,
- Maintains a list of currently active threads in this system,
- Keeps a complete list of monitor points and commands associated with this system,
- Implements the state model for this system with controlled functionality to change states.

The `ControlSystem` class implements the following methods that are accessible by remote clients. (All method signatures are presented using Java.) In these methods ‘`SystemType`’ and ‘`SystemState`’ are enumerations and ‘`RemoteConnection`’ is a class that contains the system name, IP address, and port number of the remote connection. As previously explained, if these methods fail in some manner, an exception is returned to the client.

- *Get the type of system – Synchronous*  
`SystemType getSystemType ()`
- *Get the name of this system – Synchronous*  
`String getSystemName ()`

- *Get the package name associated with this system – Synchronous*  
String getPackageName ()
- *Get the IP address of the computer on which this system is executing. – Synchronous*  
String getIPAddress ()
- *Get the main server port number on which this system listens for clients – Synchronous*  
int getMainPort ()
- *Get the current value of the backlog parameter – Synchronous*  
int getBacklog()
- *Get the current value of the SO\_timeout parameter – Synchronous*  
int getSOTimeout()
- *Get the name of the log file currently being used – Synchronous*  
String getLogFilename ()
- *Get the current state of the system – Synchronous*  
SystemState getSystemState ()
- *Get the parameters used in accessing the database manager – Synchronous*  
RemoteConnection getDatabaseManager ()
- *Get the parameters used in accessing the telescope operator – Synchronous*  
RemoteConnection getTelescopeOperator ()
- *Get the parameters used in accessing the fault manager – Synchronous*  
RemoteConnection getFaultManager ()
- *Break this connection – Synchronous*  
void breakConnection()
- *Terminate the execution of this system – Synchronous*  
void terminate()
- *Test the communications network – Synchronous*  
void test()
- *Set the parameters used in accessing the database manager – Synchronous*  
void setDatabaseManager (String systemName, String address, int port)
- *Set the parameters used in accessing the telescope operator – Synchronous*  
void setTelescopeOperator (String systemName, String address, int port)
- *Set the parameters used in accessing the fault manager – Synchronous*  
void setFaultManager (String systemName, String address, int port)

- *Set the SO\_timeout parameter – Synchronous*  
void setSOTimeout (int timeout)
- *Set the loglevel parameter used to select the logging filter level – Synchronous*  
void setLogLevel (String loglevel)
- *Initialize this system in synchronous mode – Synchronous*  
SystemState initializeSystem ()
- *Begin the initialization process, but do not respond as if this were an asynchronous command – Synchronous*  
void beginInitializeSystem ()
- *Place this system in operational mode – Synchronous*  
SystemState operateSystem ()
- *Place this system in diagnostic mode – Synchronous*  
SystemState diagnosticModeOn ()
- *Place this system back in operational mode – Synchronous*  
SystemState diagnosticModeOff ()
- *Shut down this system in synchronous mode – Synchronous*  
SystemState shutdownSystem ()
- *Begin the shut down process, but do not respond as if this were an asynchronous command – Synchronous*  
void beginShutdownSystem ()
- *Notify this system in synchronous mode that it is about to be aborted – Synchronous*  
SystemState aboutToAbortSystem ()
- *Begin the about to abort process, but do not respond as if this were an asynchronous command – Synchronous*  
void beginAboutToAbortSystem ()
- *Place this system in the stopped state – Synchronous*  
SystemState stopSystem ()

List 1: **Methods in a basic system accessible by remote clients**


The ExecuteClientRequest is a class that is in charge of receiving requests by a remote client and executing them. It is created as a separate thread when a connection is accepted from a remote client. This class uses the ControlSystemServerCommand, which it instantiates as an object to be used in executing client commands. The ControlSystemServerCommand object decodes any parameters associated with the client command using the protocol defined in Section 3, executes the command by calling methods in the ControlSystem object, encodes the response using the defined protocol, and sends the response back to the client. The

reason for creating the classes in this manner is to separate the implementation of the methods within the system (in `ControlSystem`) from the mechanics in handling remote clients (in `ExecuteClientRequest`) and also from the communications protocol used in formatting messages between servers and clients (in `ControlSystemServerCommand`).

This separation is maintained by all extensions from these classes. In fact, another reason for maintaining this separation is that we can automatically generate classes such as the `WeatherStationServerCommand` class (see Figure 5) from the spreadsheets that define that system's high-level interface.

## 2.2 An Asynchronous System

An asynchronous command, in general, takes anywhere from a few seconds to a few minutes to complete its execution. It usually does not take hours to execute. An asynchronous system adds the functionality needed to execute such commands by creating threads to handle their execution. The three classes (see Figure 4) `AsynchronousSystem`, `ExecuteAsynchronousClientRequest`, and `AsynchronousSystemServerCommand` extend the basic system classes `ControlSystem`, `ExecuteClientRequest`, and `ControlSystemServerCommand` respectively and play analogous roles. The class `ExecuteAsyncCommand` is the thread that is instantiated to execute an asynchronous command.

All systems have a main port to which a client connects in order to interact with the system. If a system implements asynchronous commands, there is a second port on which data and objects returned from asynchronous commands are published. In general these are not the same port; if they are the same, the system must insure that message packets from multiple threads are not scrambled. Using this second data port, the `AsynchronousSystem` class creates a data server socket on which it listens for connections by remote clients. A remote client must open a connection to this socket to receive the output from asynchronous commands. 

The additional methods that an asynchronous system implements that are accessible by remote clients are presented below.


- *Get the port number on which this system reports asynchronous data – Synchronous*  
`int getDataPort ()`
- *Initialize this system – Asynchronous*  
`void initializeSystemAsync (String methodName, String exceptionName)`
- *Shut down this system – Asynchronous*  
`void shutdownSystemAsync (String methodName, String exceptionName)`
- *This system should execute the about\_to\_abort process – Asynchronous*  
`void aboutToAbortSystemAsync (String methodName, String exceptionName)`

List 2: **Additional methods in an asynchronous system**

The asynchronous methods presented above are used by a client (they are provided by the ‘proxy’ object) and require two method names as parameters: the name of a method that is called to receive the data returned from the execution of the command and the name of a method that is called if an exception is returned. These parameters are general features of asynchronous commands. The DataListener thread is created within the proxy object and it calls these methods when the command has been completed.

In implementing asynchronous commands a system may choose to provide greater flexibility in how a client executes them. This technique was utilized in some of the commands to change state, e.g. the command to initialize a system. This can be a very complex process within a system; so there are three versions of this command. One that is executed synchronously, one that only starts the process of initialization and is also synchronous, and one that is completely asynchronous. A client that only starts initialization with the synchronous command must make additional requests at a later time to discover the state the system. This technique is overkill to use in all cases, but it is sometimes useful.

## 2.3 A Monitoring System

A monitoring system publishes internal status data and other types of data periodically. Each item of such data may be sampled at varying rates and published at varying intervals. In addition, this monitoring capability may be turned on and off. The three classes (see Figure 4) MonitoringSystem, ExecuteMonitoringClientRequest, and MonitoringSystemServerCommand extend the asynchronous system classes AsynchronousSystem, ExecuteAsynchronousClientRequest, and AsynchronousSystemServerCommand respectively and implement the functionality needed to support monitoring. The monitoring data are published on the same data port as asynchronous data. The Monitor class, that is part of the MonitoringSystem, is a thread that periodically wakes up, samples the necessary data, and publishes it on the data port. 

The additional methods that a monitoring system implements that are accessible by remote clients are presented below.

- *Turn data monitoring on – Synchronous*  
void monitoringOn ()
- *Turn data monitoring off – Synchronous*  
void monitoringOff ()
- *Is data monitoring currently on? – Synchronous*  
boolean isMonitoring ()

List 3: [Additional methods in a monitoring system](#)

A monitoring system is usually created by the Executive, which starts and initializes the system. Monitoring is turned on after the system has been initialized and is in the operational state. Before this happens the Executive assigns a Data Collector object to the system. The Data Collector listens for monitor data on the data port assigned to this system.

## 2.4 The State Model

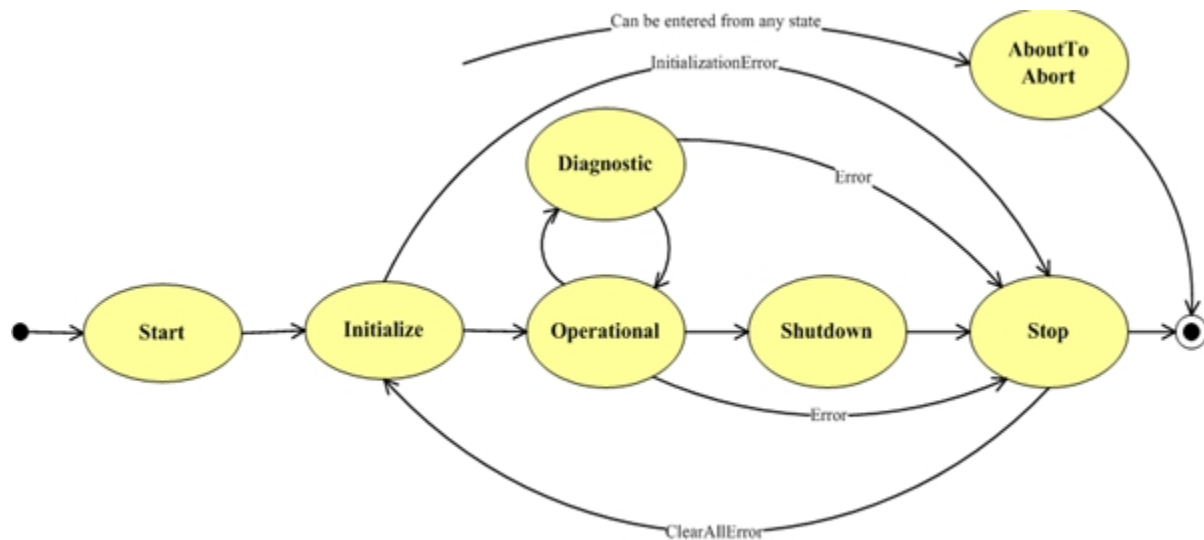


Figure 6: **The State Model of a System**

An important aspect of every system is the state model. This section will describe that state model. It is intended to be simple but rich enough to capture the essential features of MROI systems. The implementation of the state model is such that extended systems cannot change the semantics of the model. They can insert actions, such as initialization actions, that are unique to a particular type of system into the model and they can manipulate their own internal state by calling the appropriate state change methods but they cannot redefine the model. In addition, all state changes within the lifetime of a system are automatically recorded in the system's log.

SystemState is an enumeration of the values of the state model as described in the Supervisory System. All systems are in one of these states at any time.

- **UNDEFINED** *The state after which a system has merely been created as a software object.*
- **STARTED** *The system's main thread, in which the server listens for remote clients, has been started but the system has not been initialized.*
- **INITIALIZING** *The system is in the process of being initialized.*
- **INITIALIZED** *The system has been initialized.*
- **OPERATIONAL** *The system is operational.*
- **DIAGNOSTIC** *The system is in the diagnostic mode.*
- **SHUTTINGDOWN** *The system is in the process of shutting down.*

- **SHUTDOWN** *The system has been shut down.*
- **STOPPED** *The system has been stopped.*
- **ABORTING** *The system is in the process of aborting.*
- **ABORTED** *The system has aborted.*

List 4: **SystemState: An Enumeration of the States of a System**

When a system is created the constructor initializes all system variables, sets the system state to ‘UNDEFINED’, and creates the log file and the logger. If any errors are encountered during this construction process, a Control Exception is thrown; so it is up to the program that creates the object to deal with this error. No threads in the created object are started.

To start the system, its major thread is started. This action places the system in the START state. At this point the main server socket is created and it can accept clients on the main server socket. If this system is an asynchronous system, the data socket is also created at this time. Up to this point these actions are performed by the shell script that is executed remotely to start the system. This shell script is executed under the control of the Executive. The Executive then accesses the remote system via the main server socket and data needed to access the telescope operator, database manager, and fault manager are set. Any system specific actions associated with the start state are then executed. This concludes the start phase.

When told to initialize itself, the system attempts to access the telescope operator and fault manager to verify these links are working. These links are then closed. It then contacts the database manager and gets its set of monitor points and commands and any other initialization data it might need, and closes that link. The system then performs any specific actions needed to initialize this system, which may take some time. These actions are intended to make sure all subcomponents are started and slave computers are booted and initialized; it does not include complex system alignment procedures, for example. Basically, this initialization includes all the actions needed to bring the system to the state where useful activity can begin. During this time, monitor data are not collected; however, at the end of this phase, monitor data is ready to be collected. This concludes the initialization phase.

At this point the system is ready to go the operational state. The first action that the Executive does at this point is to execute the actions necessary to collect monitor data and start the process of collecting that data. The system is then told to go the operational state. At this point the system is ready to perform any command. It is at this point that activities such as the process of aligning the mirrors can begin.

The normal shutdown process begins when the system receives the “shutdownSystem()” command. There is a precisely defined set of actions that take place within the system at this time.

1. All threads processing asynchronous commands within the system are told to terminate gracefully.

2. Next, all remotely connected clients, except the client issuing the shutdown command, are told their connections are about to be broken and their connections are then terminated. This action terminates all external connections except the one issuing the shutdown command and it also terminates the threads processing those requests.
3. At this point any activity collecting monitor data is turned off.
4. The system then takes any system specific actions necessary to stop its internal sub-components, which probably means placing them in a standby state.
5. This concludes the shutdown actions and the system is in the SHUTDOWN state.

Once the system is in the SHUTDOWN state it can be told either to re-initialize or to stop. These commands can only be given by the remote client, usually the Executive, that issued the shutdown command. No other external client is accepted by the system after the shutdown process begins. If some pathological condition occurs, it is assumed that if the system is in the SHUTDOWN or STOPPED state, which could be determined by examining the system's log file for example, the operating system could kill the main system process without damage to the system.

One can also stop the system by issuing the “aboutToAbortSystem()” command. However, this is not a graceful shutdown. Its purpose is to give the system an advance warning in order to save any critical data before being terminated. The system may be aborted at any point after this command is given, so only minimal activities should be undertaken. Even if these actions are successfully carried out, the system can only be terminated; it cannot be re-initialized.

The “normal” method of terminating the system is to issue a change of state command as outlined above, either a “shutdownSystem” or an “aboutToAbortSystem” command. However, it is possible for a client, which should be the Executive, to issue a “terminate” command.

The “terminate” method is the last thing that is executed in the system's “run” method. Its action is dependent on the state the system is in. For example, if it was called in an operational state, a graceful shutdown is attempted. Basically, what one wishes to accomplish is to terminate all connections and threads in an orderly manner and take whatever actions are necessary to achieve an orderly shutdown of all sub-systems within this system.

In addition to stopping the system by issuing commands, there is a timeout associated with the system. After the initial client is accepted, if the server has no clients within a certain window of time, the timeout will be initiated and the system will shutdown of its own accord, by executing the “terminate” command. Under normal circumstances, the Executive prevents this from happening by issuing a “getSystemState” command periodically. If all external communication with the system is lost, this timeout mechanism will stop the system.



### 3 Communications Protocol

This section will discuss the protocol by which an arbitrary remote client interacts with such a system. A client formats a message and sends it to the server system. The server then processes the message, formats a response, and sends the response to the client.

This protocol is based on TCP/IP. The specifics of how various data items are encoded and decoded within the message format will be discussed shortly.

The first item within any message exchanged between servers and clients is a single byte that indicates the type of message. Its valid values are listed below.

- **UNKNOWN** 0. *Not intended to be used.*
- **SYSTEM\_IDENTIFICATION** 1. *Used by both clients and servers to identify their system in the connection procedure.*

#### Messages sent from a server to a client

- **EXECUTED** 2. *Server to client: a command has been executed successfully.*
- **EXECUTED\_NULL** 3. *Server to client: a command has been executed successfully but the result was null.*
- **EXCEPTION** 4. *Server to client: an executing command threw an exception.*
- **ACCEPTED** 5. *Server to client: an asynchronous command has been accepted for execution.*
- **MONITOR\_DATA** 6. *Indicates monitor data sent from server to client.*

#### Messages sent from a client to a server implemented in a basic system

- **GET\_SYSTEM\_TYPE** 7. *Client to server: get the type of system.*
- **GET\_PACKAGE\_NAME** 8. *Client to server: get the package name associated with the system.*
- **GET\_SYSTEM\_NAME** 9. *Client to server: get the name of this system.*
- **GET\_IP\_ADDRESS** 10. *Client to server: get the IP address of the computer on which this system is executing.*
- **GET\_MAIN\_PORT** 11. *Client to server: get the main server port number.*
- **GET\_BACKLOG** 12. *Client to server: get the current value of the backlog parameter.*

- **GET\_SO\_TIMEOUT** 13. *Client to server: get the current value of the so\_timeout parameter.*
- **GET\_LOG\_FILENAME** 14. *Client to server: get the name of the log file currently being used.*
- **GET\_SYSTEM\_STATE** 15. *Client to server: get the current state of the system.*
- **GET\_DATABASE\_MANAGER\_CONNECTION** 16. *Client to server: get the parameters used in accessing the database manager.*
- **GET\_TELESCOPE\_OPERATOR\_CONNECTION** 17. *Client to server: get the parameters used in accessing the telescope operator.*
- **GET\_FAULT\_MANAGER\_CONNECTION** 18. *Client to server: get the parameters used in accessing the fault manager.*
- **BREAK\_CONNECTION** 19. *Client to server: break this connection.*
- **TERMINATE** 20. *Client to server: terminate the execution of this system.*
- **TEST** 21. *Client to server: Used only to test the communications network.*
- **SET\_DATABASE\_MANAGER** 22. *Client to server: set the parameters used in accessing the database manager.*
- **SET\_TELESCOPE\_OPERATOR** 23. *Client to server: set the parameters used in accessing the telescope operator.*
- **SET\_FAULT\_MANAGER** 24. *Client to server: set the parameters used in accessing the fault manager.*
- **SET\_SOTIMEOUT** 25. *Client to server: set the so\_timeout parameter.*
- **SET\_LOGLEVEL** 26. *Client to server: set the loglevel parameter used to select the logging filter level.*
- **INITIALIZE\_SYSTEM** 27. *Client to server: initialize this system in synchronous mode.*
- **BEGIN\_INITIALIZE\_SYSTEM** 28. *Client to server: begin the initialization process, but do not respond as if this were an asynchronous command.*
- **OPERATE\_SYSTEM** 29. *Client to server: place this system in operational mode.*
- **DIAGNOSTIC\_MODE\_ON** 30. *Client to server: place this system in diagnostic mode.*
- **DIAGNOSTIC\_MODE\_OFF** 31. *Client to server: place this system back in operational mode.*

- **SHUTDOWN\_SYSTEM** 32. *Client to server: shut down this system in synchronous mode.*
- **BEGIN\_SHUTDOWN\_SYSTEM** 33. *Client to server: begin the shut down process, but do not respond as if this were an asynchronous command.*
- **ABOUT\_TO\_ABORT\_SYSTEM** 34. *Client to server: notify this system in synchronous mode that it is about to be aborted.*
- **BEGIN\_ABOUT\_TO\_ABORT\_SYSTEM** 35. *Client to server: begin the about to abort process, but do not respond as if this were an asynchronous command.*
- **STOP\_SYSTEM** 36. *Client to server: place this system in the stopped state.*

### Messages sent from a client to a server implemented in an asynchronous system

- **GET\_DATAPORT** 37. *Client to server: get the port number on which this system reports asynchronous data.*
- **INITIALIZE\_SYSTEM\_ASYNC** 38. *Client to server: initialize this system in asynchronous mode.*
- **SHUTDOWN\_SYSTEM\_ASYNC** 39. *Client to server: shut down this system in asynchronous mode.*
- **ABOUT\_TO\_ABORT\_SYSTEM\_ASYNC** 40. *Client to server: this system should execute the about\_to\_abort process in asynchronous mode.*

### Messages sent from a client to a server implemented in a monitoring system

- **MONITOR\_ON** 41. *Client to server: turn data monitoring on.*
- **MONITOR\_OFF** 42. *Client to server: turn data monitoring off.*
- **IS\_MONITORING** 43. *Client to server: is data monitoring currently on?*

### Messages sent from a client to a server implemented in an extended system

- **SYNCHRONOUS\_COMMAND** 44. *Indicates a general synchronous command sent by a client to a server.*
- **ASYNCHRONOUS\_COMMAND** 45. *Indicates a general asynchronous command sent by a client to a server.*

List 5: [MessageType: An Enumeration of Types of Messages](#)

For message types 1 – 43 if any parameters are required they immediately follow the byte indicating the message type. They are encoded as binary data as described in Section 3.1. These messages represent all the “built-in” commands implemented in the basic, asynchronous, and monitoring classes from which all other systems are extended.

A message of type `SYNCHRONOUS_COMMAND` is used by any extended system that defines a synchronous command. It is followed by:

- command name, encoded as a String
- argument 1 encoded
- argument 2 encoded
- argument 3 encoded
- . . .

A message of type `ASYNCHRONOUS_COMMAND` is used by any extended system that defines an asynchronous command. It is followed by:

- command name, encoded as a String
- command time, encoded as a long integer
- argument 1 encoded
- argument 2 encoded
- argument 3 encoded
- . . .

The command time is the time the command was sent to the server and is used as a tag to distinguish multiple commands of the same name sent in overlapping windows of time.

The first item of data returned by the server to the client is an enumeration indicating the type of response: `EXECUTED`, `EXECUTED_NULL`, `EXCEPTION`, `ACCEPTED`. The second item of data is the returned data of interest, if anything was supposed to be returned. If `EXECUTED` is returned, it is followed by the returned data, unless the return-type of the command was ‘void’. If `EXECUTED_NULL` is returned, the command was successfully executed but the returned object was null. If `EXCEPTION` is returned, the command was not successfully executed and it is followed by the exception itself. `ACCEPTED` is returned only by an asynchronous command and indicates that the command was accepted and is being executed. If the command was not accepted, an exception is returned.

If anything is returned by executing an asynchronous command it is returned on the data output socket. For asynchronous commands, the message that is returned on the data output socket is similar to the format for synchronous commands and is as follows.

- **message-type** either *EXECUTED*, *EXECUTED\_NULL*, or *EXCEPTION*
- **command name** the command name to which this returned data is in response
- **command time** the command time used to identify the command to which this returned data is in response
- **returned item of data** may be null or an exception, depending on message type

The format of monitor data requires special consideration. All monitor data are identified by the `MONITOR_DATA` message type. Its general format is as follows.

- **MONITOR\_DATA** the message type indicating monitor data
- **system instance identifier** a short integer assigned by the database that identifies this system instance
- **monitored property identifier** a short integer assigned by the database that identifies this monitored property

- **sampling time** the time at which the monitored property was sampled
- **monitor data** the actual data that was sampled, encoded as described in Section 3.1

The ‘system instance identifier’ and the ‘monitored property identifier’ are obtained from the Database Manager when the system is initialized and gets its list of monitored properties and commands.

### 3.1 Encoding and decoding types of data

The permitted data types are:

- **boolean** single byte: either 0 or 1
- **byte** 8-bit signed binary integer
- **short** 16-bit signed binary integer
- **int** 32-bit signed binary integer
- **long** 64-bit signed binary integer
- **float** IEEE 32-bit single precision floating point number
- **double** IEEE 64-bit double precision floating point number
- **char** 16-bit Unicode character

- **String** *an array of bytes that is the UTF-8 encoding of the character string, preceded by the length of the array as a short*
- **enumerations** *the ordinal value of the enumerated item as a byte*
- **any DataStreamable class** *'DataStreamable' is an interface that requires methods to 'read' and 'write' data associated with input and output streams*
- **1-D arrays of the above** *the array of items is preceded by the number of items in the array as an int*

There is a collection of static methods in the Protocol class that encodes and decodes all of these data types. All of these methods throw “java.io.IOException”. All of the binary data is in “network” order, i.e. big-endian bit order.

### 3.1.1 Methods for encoding data

```
import java.io.DataInputStream
import java.io.DataOutputStream

public static void encode (boolean x, DataOutputStream out)
public static void encode (byte x, DataOutputStream out)
public static void encode (short x, DataOutputStream out)
public static void encode (int x, DataOutputStream out)
public static void encode (long x, DataOutputStream out)
public static void encode (float x, DataOutputStream out)
public static void encode (double x, DataOutputStream out)
public static void encode (char x, DataOutputStream out)
public static void encode (String x, DataOutputStream out)
public static void encode (Enum<?> x, DataOutputStream out)
public static void encode (DataStreamable x, DataOutputStream out)
public static void encode (boolean[] x, DataOutputStream out)
public static void encode (byte[] x, DataOutputStream out)
public static void encode (short[] x, DataOutputStream out)
public static void encode (int[] x, DataOutputStream out)
public static void encode (long[] x, DataOutputStream out)
public static void encode (float[] x, DataOutputStream out)
public static void encode (double[] x, DataOutputStream out)
public static void encode (char[] x, DataOutputStream out)
public static void encode (String[] x, DataOutputStream out)
public static void encode (Enum<?>[] x, DataOutputStream out)
public static void encode (DataStreamable[] x, DataOutputStream out)
```

### 3.1.2 Methods for decoding data

```
public static boolean decodeBoolean (DataInputStream in)
public static byte decodeByte (DataInputStream in)
public static short decodeShort (DataInputStream in)
public static int decodeInt (DataInputStream in)
public static long decodeLong (DataInputStream in)
public static float decodeFloat (DataInputStream in)
public static double decodeDouble (DataInputStream in)
public static char decodeChar (DataInputStream in)
public static String decodeString (DataInputStream in)
public static Enum<?> decodeEnum (Enum<?> s, DataInputStream in)
public static DataStreamable decode (DataStreamable x, DataInputStream in)
public static boolean[] decodeBooleanArray (DataInputStream in)
public static byte[] decodeByteArray (DataInputStream in)
public static short[] decodeShortArray (DataInputStream in)
public static int[] decodeIntArray (DataInputStream in)
public static long[] decodeLongArray (DataInputStream in)
public static float[] decodeFloatArray (DataInputStream in)
public static double[] decodeDoubleArray (DataInputStream in)
public static char[] decodeCharArray (DataInputStream in)
public static String[] decodeStringArray (DataInputStream in)
public static Enum<?>[] decodeEnumArray (Enum<?> s, DataInputStream in)
public static Object decodeArray ( DataStreamable x, DataInputStream in)
```



## 4 Defining a High-level System Interface

In this section we will discuss how the high-level interface to a system is defined.

Each type of system in MROI is described using a set of Excel spreadsheets that provide a complete description of the high-level interface to that system. These spreadsheets provide complete information about all monitor points and all control commands, together with their fault conditions and parameters, that are used to interact with the system. There are five worksheets in a spreadsheet that describe a system: the system worksheet, used to describe the system as a whole; the monitor worksheet, which is used to define all monitor points in detail; the fault worksheet, used to define faults associated with monitor points; the control worksheet, which defines all commands that control the system; and, the parameters worksheet, which defines parameters associated with control commands or with the system as whole.

The spreadsheet, in the form of these five worksheets, is stored in XML format. A program then parses the XML file to extract the data in the worksheets, which is used to create a model of the system. This model is then used as input to a Java-based code generation framework that generates whatever code is necessary to integrate the system into the MROI framework.

We will now define the contents of these worksheets in greater detail. Each column in the worksheet is described together with an indication of the type of entry is expected in that column.

### 4.1 System Worksheet

The System worksheet is shown in Figure 7. This worksheet describes the system as a whole. One can describe more than one system, as in the example. The first entry is the system as a whole (the Environmental Monitoring System in this case) followed by its component systems (a Weather Station in this case). This is only an example, and the real worksheet would include entries for the all sky camera, dust monitor, etc. Only the first entry includes the work package and defining document, since it is assumed that all components are described by the same document.

#### 4.1.1 Columns in The System worksheet

- **Name** (*name*) *The official name of this type of system, which is usually short.*
- **Description** (*text*) *A brief description of the system.*
- **Package** (*text*) *The name of the Java package associated with this system.*
- **Full Name** (*text*) *The full name of this type of system.*
- **Extends** (*text*) *The name of the class that this system extends, or 'none' if it does not extend anything.*



System Interface Definition		
Name	Description	Package
EnvironmentalMonitoringSystem	tbd	mroi.simulation.ems
WeatherStation	tbd	mroi.simulation.ems

Full Name	Extends	Parent System
Environmental Monitoring System	ControlSystem	none
Weather Station	MonitoringSystem	EnvironmentalMonitoringSystem

Implement	Is Asynchronous	Is A Monitor	Work Package
yes	no	no	4.15.00
yes	yes	yes	

Document Title	Document Number	Document Issue	Document Date
Environmental Monitoring System Requirements	415-INT-REQ-0010	1	12/1/2008


Figure 7: System Interface Definition – The System Worksheet

- **Parent System** (*text*) The name of the parent system, or ‘none’ if it is not part of a larger system.
- **Implement** (*yes/no*) Should this system be implemented? (It may merely be a package repository for a collection of independent systems.)
- **Is Asynchronous** (*yes/no*) Does this system implement asynchronous commands?
- **Is A Monitor** (*yes/no*) Does this system publish monitor data?
- **Work Package** (*text*) The name of the work package associated with this system.
- **Document Title** (*text*) The title of the document that describes this system.
- **Document Number** (*text*) The number of the document that describes this system.
- **Document Issue** (*text*) The particular version of the document that describes this system.
- **Document Date** (*date*) The date of the document that describes this system.

## 4.2 Monitor Worksheet

The Monitor worksheet is shown in Figure 8. This worksheet defines all monitored properties in the system, including all status data, image data, and other data that might be calculated in the course of fulfilling its functions.

### 4.2.1 Columns in the Monitor worksheet

- **Name** (*name*) The name of this monitor point, which must be unique within the system.
- **System** (*name*) The name of the type of system to which this monitor point belongs.
- **Description** (*text*) A brief description of this monitor point.
- **Returns** (*text*) The name of the type of object, basic or extended data type that is returned by this monitor point.
- **Can Be Null** (*yes/no*) Can this monitor point return a null value?
- **Throws Exception** (*yes/no*) Can this monitor point throw an exception?
- **Asynchronous** (*yes/no*) Is this monitor point implemented using an asynchronous command? 
- **Data Unit** (*text*) The units used to store this monitor point in the archive. (or ‘none’ if units do not apply.)

### Monitor Points

Name	System	Description	Returns	Can Be Null	Throws Exception
Temperature	WeatherStation	tbd	Temperature	no	yes
WindSpeed	WeatherStation	tbd	Speed	no	yes
WindDirection	WeatherStation	tbd	Angle	no	yes
TemperatureInterval	WeatherStation	tbd	Duration	no	no
WindSpeedInterval	WeatherStation	tbd	Duration	no	no
WindDirectionInterval	WeatherStation	tbd	Duration	no	no

Asynchronous	Data Unit	Minimum Value	Maximum Value	Default Value	System Unit	Raw Data Type
no	°C	-30	60	30	degC	Temperature
no	m/sec	0	100	4	m/sec	Speed
no	°	0	360	45	degC	Angle
no	sec	1	300	5	sec	Duration
no	sec	1	300	5	sec	Duration
no	sec	1	300	5	sec	Duration



Scale	Offset	Mode	Implement	Archive Interval (secs)	Archive Only On Change
none	none	any	yes	5	no
none	none	any	yes	5	no
none	none	any	yes	5	no
none	none	any	yes	none	no
none	none	any	yes	none	no
none	none	any	yes	none	no

Display Unit	Graph Minimum	Graph Maximum	Graph Title
degC	-10	40	Ambiant Air Temperature (degC)
m/sec	0	20	Wind Speed (m/sec)
degC	0	360	Wind Direction (deg)
sec	none	none	none
sec	none	none	none
sec	none	none	none

Figure 8: System Interface Definition – The Monitor Worksheet

- **Minimum Value** (*number*) *The minimum value of this monitor point. (or ‘none’ if this is not applicable.)*
- **Maximum Value** (*number*) *The maximum value of this monitor point. (or ‘none’ if this is not applicable.)*
- **Default Value** (*number*) *The default value of this monitor point. (or ‘none’ if this is not applicable.)*
- **System Unit** (*text*) *The units used by the Supervisory System in conjunction with this monitor point. (or ‘none’ if units do not apply.)*
- **Raw Data Type** (*name*) *The raw data type associated with this monitor point that is used internally in the system, i.e. the type used internally in measuring its value. (or ‘none’ if this notion isn’t applicable or it is the same as the system value.)*
- **Scale** (*number*) *The scale factor used to convert the raw value to its system value, or ‘none’ if no conversion is necessary. The formula is:  $system\_value = scale\_factor * raw\_value + offset$ .*
- **Offset** (*number*) *The offset factor used to convert the raw value to its system value, or ‘none’ if no conversion is necessary. The formula is:  $system\_value = scale\_factor * raw\_value + offset$ .*
- **Mode** (*name*) *The operating mode (startup, initialization, operational, diagnostic, shutdown, or any) in which this monitor point may be sampled.*
- **Implement** (*yes/no*) *Should the method to get the value of this monitor point be automatically generated? (If not, a signature will be generated that requires this method to be hand crafted.)*
- **Archive Interval (secs)** (*number, in seconds*) *What is the interval of time that this property should be stored in the archive?*
- **Archive Only On Change** (*yes/no*) *Should the value of this property be archived only when it changes?*
- **Display Unit** (*text*) *The unit, associated with this property, that is used in graphical displays.*
- **Graph Minimum** (*number*) *The minimum value used in graphical displays.*
- **Graph Maximum** (*number*) *The maximum value used in graphical displays.*
- **Graph Title** (*text*) *The title used in graphical displays.*

## 4.3 Fault Worksheet

The Fault worksheet is shown in Figure 9. The Fault worksheet defines all faults associated with the system. These fault definitions may either be associated with a particular monitored property or with the system as a whole.

### 4.3.1 Columns in the Fault worksheet

- **Fault Name** (*name*) *The name of the fault, which must be unique within this system.*
- **System** (*name*) *The name of the type of system associated with this fault.*
- **Monitor Point** (*name*) *The name of the monitored property associated with this fault, if any. If this fault is associated with the system as a whole, ‘none’ should be entered.*
- **Description** (*text*) *A description of this fault.*
- **Fault Condition** (*text*) *A boolean condition that defines this fault.*
- **Fault Severity** (*name*) *The severity associated with this fault.*
- **Fault Action** (*list*) *A list of actions to be taken if this fault occurs. (Not defined at this time.)*

## 4.4 Control Worksheet

The Control worksheet is shown in Figure 10. This Control worksheet defines all control commands by which the system may be managed. In general, monitor points do not change the internal state of a system; they only retrieve data. Control commands are intended to change the internal state of the system.


### 4.4.1 Columns in the Control worksheet

- **Name** (*name*) *The name of the control command, which must be unique within this system.*
- **System** (*name*) *The name of the type of system associated with this control command.*
- **Description** (*text*) *A description of this control command.*
- **Returns** (*text*) *The name of the object, basic or extended data type that is returned by the method executing this command.*
- **Can Be Null** (*yes/no*) *Can this command return a null value?*

Fault Definitions			
Fault Name	System	Monitor Point	Description
TooCold	WeatherStation	Temperature	tbd
TooHot	WeatherStation	Temperature	tbd
HighWind	WeatherStation	WindSpeed	tbd
Wind	WeatherStation	WindSpeed	tbd

Fault Condition	Fault Severity	Fault Action
value < -10.0	Severe	AllStop
value > 40.0	Severe	AllStop
value > 20.0	Severe	AllStop
value > 10.0	Warning	Continue

Figure 9: System Interface Definition – The Fault Worksheet

Control Commands				
Name 	System	Description	Returns	Can Be Null
setTemperatureInterval	WeatherStation	tbd	void	no
setWindSpeedInterval	WeatherStation	tbd	void	no
setWindDirectionInterval	WeatherStation	tbd	void	no
getAverageWindSpeed	WeatherStation	tbd	Speed	no

Throws Exception	Asynchronous	Mode	Implement
no	no	any	yes
no	no	any	yes
no	no	any	yes
no	yes	any	yes

Figure 10: System Interface Definition – The Control Worksheet

- **Throws Exception** *(yes/no) Can this command throw an exception?*
- **Asynchronous** *(yes/no) Is the method executing this command an asynchronous method?*
- **Mode name** *The operating mode (startup, initialization, operational, diagnostic, shutdown, or any) in which this control command may be executed.*
- **Implement** *(yes/no) Should the method to execute this control command be automatically generated? (If not, a signature will be generated that requires this method to be hand crafted.)*

## 4.5 Parameters Worksheet

The Parameters worksheet is shown in Figure 11. This worksheet defines any parameters that are associated with the control commands. Parameters may also be defined that apply to the system as a whole.

### 4.5.1 Columns in the Parameters worksheet

- **Parameter Name** *(name) The name of this parameter, which must be unique within the command with which it is associated.*
- **System** *(name) The name of the type of system associated with this parameter.*
- **Command** *(name) The name of the type of control command associated with this parameter. If this parameter is associated with the system as a whole, rather than a specific command, then 'none' should be entered.*
- **Description** *(text) A description of this control command.*
- **Required** *(yes/no) Is this parameter required?*
- **Data Type** *(name) The name of the data type of this parameter that is used by the Supervisory System.*
- **Data Unit** *(text) The unit, associated with this parameter, that is used by the Supervisory System.*
- **Minimum Value** *(number) The minimum value allowed for this parameter in units used by the Supervisory System.*
- **Maximum Value** *(number) The maximum value allowed for this parameter in units used by the Supervisory System.*
- **Default Value** *(number) A default value for this parameter in units used by the Supervisory System.*

## Parameters

Parameter Name	System	Command	Description	Required
temperatureInterval	WeatherStation	setTemperatureInterval	tbd	yes
windSpeedInterval	WeatherStation	setWindSpeedInterval	tbd	yes
windDirectionInterval	WeatherStation	setWindDirectionInterval	tbd	yes
minutes	WeatherStation	getAverageWindSpeed	tbd	yes
methodName	WeatherStation	getAverageWindSpeed	tbd	yes
exceptionName	WeatherStation	getAverageWindSpeed	tbd	yes

Data Type	Data Unit	Minimum Value	Maximum Value	Default Value
Duration	sec	1	300	5
Duration	sec	1	300	5
Duration	sec	1	300	5
Duration	min	1	60	5
String	none	none	none	none
String	none	none	none	none

System Unit	Raw Data Type	Scale	Offset
sec	integer	none	none
sec	integer	none	none
sec	integer	none	none
min	float	none	none
none	none	none	none
none	none	none	none

Figure 11: System Interface Definition – The Parameters Worksheet



- **System Unit** (*name*) *The unit, associated with this parameter, that is required by this system.*
- **Raw Data Type** (*name*) *The internal data type associated with the command that requires this parameter.*
- **Scale** (*number*) *The scale factor used to convert the system value to its internal value, or 'none' if no conversion is necessary. The formula is:  $internal\_value = scale\_factor * system\_value + offset$ .*
- **Offset** (*number*) *The offset factor used to convert the system value to its internal value, or 'none' if no conversion is necessary. The formula is:  $internal\_value = scale\_factor * system\_value + offset$ .*

## 4.6 The Code Generation Framework

Using the spreadsheets (Figures 7 – 11) three files are generated containing the Java classes: WeatherStationBase, WeatherStationProxy, and WeatherStationServerCommand. These files are completely implemented and handle all the encoding and decoding of message formats passed between the server and clients. The actual WeatherStation class, which is the internal implementation of the weather station system, only has to extend WeatherStationBase, which means that it is required to implement the abstract methods in the base class. The signatures of the generated methods in these three classes are given below.

### Class and methods defined in file WeatherStationBase.java:

```
abstract public class WeatherStationBase extends MonitoringSystem
    public WeatherStationBase (String systemName, String hostAddress,
        int portNumber, int dataPort, int backlog) throws ControlException
    abstract public void monitorOn();
    abstract public void monitorOff();
    abstract public boolean isMonitoring();
    abstract public Temperature getTemperature () throws ControlException;
    abstract public Speed getWindSpeed () throws ControlException;
    abstract public Angle getWindDirection () throws ControlException;
    abstract public Duration getTemperatureInterval ();
    abstract public Duration getWindSpeedInterval ();
    abstract public Duration getWindDirectionInterval ();
    abstract public void setTemperatureInterval
        (Duration temperatureInterval);
    abstract public void setWindSpeedInterval
        (Duration windSpeedInterval);
    abstract public void setWindDirectionInterval
        (Duration windDirectionInterval);
    abstract public Speed getAverageWindSpeed
```

(Duration minutes);

### Class and methods defined in file WeatherStationProxy.java:

```
public class WeatherStationProxy extends MonitoringSystemProxy
    public WeatherStationProxy (String systemName, String hostAddress,
        int portNumber, Identification clientId, ControlLogger logger)
    public Temperature getTemperature () throws ControlException
    public Speed getWindSpeed () throws ControlException
    public Angle getWindDirection () throws ControlException
    public Duration getTemperatureInterval () throws ControlException
    public Duration getWindSpeedInterval () throws ControlException
    public Duration getWindDirectionInterval () throws ControlException
    public void setTemperatureInterval
        (Duration temperatureInterval) throws ControlException
    public void setWindSpeedInterval
        (Duration windSpeedInterval) throws ControlException
    public void setWindDirectionInterval
        (Duration windDirectionInterval) throws ControlException
    public void getAverageWindSpeed (Duration minutes, String methodName,
        String exceptionName) throws ControlException
```

### Class and methods defined in file WeatherStationServerCommand.java:

```
public class WeatherStationServerCommand extends
    MonitoringSystemServerCommand
    public WeatherStationServerCommand (WeatherStation server,
        DataOutputStream out, DataInputStream in)
    public void getTemperature () throws IOException
    public void getWindSpeed () throws IOException
    public void getWindDirection () throws IOException
    public void getTemperatureInterval () throws IOException
    public void getWindSpeedInterval () throws IOException
    public void getWindDirectionInterval () throws IOException
    public void setTemperatureInterval () throws IOException
    public void setWindSpeedInterval () throws IOException
    public void setWindDirectionInterval () throws IOException
    public void getAverageWindSpeed () throws IOException
```

This document has had as its intention to illustrate the structure of a system from the Supervisory System's perspective. In doing so we have emphasized Java as an implementation language. Discussions are underway to enlarge this perspective to accommodate systems implemented in other languages, such as C or C++.

## 5 Change History

### 5.1 Version 0.9

Version 0.9 was an initial draft that was distributed internally for review. There were no versions prior to Version 0.9.

## 6 Additional information

### References

- [1] A. Farris, *MROI Supervisory System: A Conceptual Design Overview*, internal document (WP 4.09.03 Version 1.0), February 20, 2009.
- [2] A. Farris, *The MROI Monitor and Configuration Database*, internal document number INT-409-ENG-0030 rev 1.2, September 3, 2009.
- [3] A. Farris, *RDM: A software system based on the relational data model for supporting the definition and collection of data for scientific applications*, internal document (WP 4.09.03 Version 1.1), August 11, 2009. (Not yet published.)

Reference [1] gives an overview of the structure of the Supervisory System. References [2] and [3] provide more information about the MROI Monitor and Configuration Database. All of the above documents may be found in the MROI document repository (which is forthcoming).