An Interface to a System in C

Allen Farris

May 17, 2010

Version: **0.2**

Document: **INT-409-ENG-0070 rev 0.2**
Work package: **WP 4.09.01**
System: **MROI Software Engineering**

Approved by: **Allen Farris**
MROI
Lead, Software and Control Systems Group

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

Contents

1 Overview **10**

1.1 Structure of an MROI system 10

1.2 Object-oriented programming in C 12

1.3 Basic system classes 14

2 High-level Interface Definition **16**

2.1 System worksheet 16

2.2 Monitor Worksheet 18

2.3 Fault Worksheet 19

2.4 Control Worksheet 19

2.5 Parameters Worksheet 20

2.6 Requirements on the system 21

3 An Example **24**

4 Communications and Monitoring **30**

5 Enumerations **31**

5.1 Enumeration: **MROIAlertLevel** 31

5.2 Enumeration: **MROIExceptionType** 31

5.3 Enumeration: **MROILogLevel** 32

5.4 Enumeration: **MROILogType** 32

5.5 Enumeration: **MROIMessageType** 33

5.6 Enumeration: **MROISystemType** 36

5.7 Enumeration: **MROISystemState** 36

5.8 Enumeration: **MROIHardwareType** 37

6 Extended Data Types **38**

7 MCDB Structures **39**

7.1 Float Sample 39

7.2 Name Value Pair 39

8 Additional General Methods **40**

9 The ControlSystem Class **41**

9.1 **Fields** . . . 41

9.2 Methods for constructing and destroying a ControlSystem . . . 42

9.2.1 **_globalError** . . . 42

9.2.2 createControlSystem . . . 43

9.2.3 **_initControlSystem** . . . 44

9.2.4 destroyControlSystem . . . 45

9.3 Methods for handling exceptions . . . 45

9.3.1 isSystemException . . . 45

9.3.2 setSystemException . . . 45

9.3.3 clearSystemException . . . 46

9.3.4 getSystemException . . . 46

9.4 Methods for sending faults, alerts, and operator messages. . . . 47

9.4.1 sendMROIFault . . . 47

9.4.2 sendMROIAlert . . . 48

9.4.3 sendMROIOperatorMessage . . . 48

9.5 Methods for setting log characteristics and writing to the log. . . . 49

9.5.1 getLogFilename . . . 49

9.5.2 setLoggerBuffersize . . . 50

9.5.3 setLoggerThreadsOption . . . 50

9.5.4 logSevere . . . 50

9.5.5 logWarning . . . 51

9.5.6 logInfo . . . 51

9.5.7 logConfig . . . 52

9.5.8 logFine . . . 52

9.5.9 logFiner . . . 53

9.5.10 logFinest . . . 53

9.5.11 logCurrentException . . . 54

9.5.12 getLogger . . . 54

9.6 Methods for accessing the database manager . . . 54

9.6.1 connectToDatabaseManager . . . 54

9.6.2 disconnectFromDatabaseManager . . . 55

9.7 Methods for getting basic system characteristics . . . 55

9.7.1 getSystemType . . . 55
9.7.2 getPackageName . . . 55
9.7.3 getSystemName . . . 56
9.7.4 getHostAddress . . . 56
9.7.5 getMainPort . . . 56
9.7.6 getDataPort . . . 57
9.7.7 getBacklog . . . 57
9.7.8 getSOTimeout . . . 57
9.7.9 getSystemState . . . 58
9.7.10 getDatabaseManagerConnection . . . 58
9.7.11 getTelescopeOperatorConnection . . . 58
9.7.12 getFaultManagerConnection . . . 59
9.8 Methods for setting basic system characteristics . . . 59
9.8.1 setDatabaseManager . . . 59
9.8.2 setTelescopeOperator . . . 60
9.8.3 setFaultManager . . . 60
9.8.4 setSOTimeout . . . 61
9.8.5 setLogLevel . . . 61
9.9 Methods for implementing the system state model . . . 61
9.9.1 startSystem . . . 61
9.9.2 initializeSystem . . . 62
9.9.3 beginInitializeSystem . . . 62
9.9.4 operateSystem . . . 62
9.9.5 diagnosticModeOn . . . 63
9.9.6 diagnosticModeOff . . . 63
9.9.7 shutdownSystem . . . 63
9.9.8 beginShutdownSystem . . . 64
9.9.9 aboutToAbortSystem . . . 64
9.9.10 beginAboutToAbortSystem . . . 64
9.9.11 stopSystem . . . 65
9.9.12 initializeSystemAsync . . . 65
9.9.13 shutdownSystemAsync . . . 65
9.9.14 aboutToAbortSystemAsync . . . 66

9.10 Methods for implementing monitoring . . . 66
9.10.1 monitorOn . . . 66
9.10.2 monitorOff . . . 66
9.10.3 isMonitoring . . . 67
9.11 Methods related to communications . . . 67
9.11.1 breakConnection . . . 67
9.11.2 terminate . . . 67
9.11.3 test . . . 68
9.12 Other methods . . . 68
9.12.1 setControlMonitorPoints . . . 68
9.12.2 setControlCommands . . . 68

10 The ControlException Class **70**
10.1 **Fields** . . . 70
10.2 Methods . . . 70
10.2.1 createControlException . . . 70
10.2.2 destroyControlException . . . 71
10.2.3 setException . . . 71
10.2.4 readControlException . . . 72
10.2.5 writeControlException . . . 73
10.2.6 toStringControlException . . . 73
10.2.7 isStatusOK . . . 73
10.2.8 clearException . . . 74
10.2.9 getExceptionType . . . 74
10.2.10 getExceptionMessage . . . 75
10.2.11 getExceptionTime . . . 75
10.2.12 getExceptionFilename . . . 75
10.2.13 getExceptionLine . . . 76

11 The MROISocket Class **77**
11.1 **Fields** . . . 77
11.2 Methods . . . 77
11.2.1 createMROISocket . . . 77
11.2.2 destroyMROISocket . . . 78
11.2.3 getSocketInputStream . . . 78
11.2.4 getSocketOutputStream . . . 78

12 The MROIServerSocket Class **79**
12.1 Fields . . . 79
12.2 Methods . . . 79
12.2.1 createMROIServerSocket . . . 79
12.2.2 destroyMROIServerSocket . . . 80
12.2.3 acceptMROIServerSocket . . . 80

13 The SocketInputStream Class **81**
13.1 Fields . . . 81
13.2 Methods . . . 81
13.2.1 createSocketInputStream . . . 81
13.2.2 destroySocketInputStream . . . 81
13.2.3 receiveSocketInputStream . . . 82
13.2.4 readBoolean . . . 82
13.2.5 readByte . . . 83
13.2.6 readShort . . . 83
13.2.7 readInt . . . 83
13.2.8 readLong . . . 84
13.2.9 readFloat . . . 84
13.2.10 readDouble . . . 84
13.2.11 readString . . . 85
13.2.12 readEnum . . . 85
13.2.13 readBooleanArray . . . 85
13.2.14 readByteArray . . . 86
13.2.15 readShortArray . . . 86
13.2.16 readIntArray . . . 86
13.2.17 readLongArray . . . 87
13.2.18 readFloatArray . . . 87
13.2.19 readDoubleArray . . . 88
13.2.20 readStringArray . . . 88
13.2.21 readEnumArray . . . 88

14 The SocketOutputStream Class **90**
14.1 Fields . . . 90

14.2 Methods . . . 90
14.2.1 createSocketOutputStream . . . 90
14.2.2 destroySocketOutputStream . . . 90
14.2.3 sendSocketOutputStream . . . 91
14.2.4 writeBoolean . . . 91
14.2.5 writeByte . . . 91
14.2.6 writeShort . . . 92
14.2.7 writeInt . . . 92
14.2.8 writeLong . . . 92
14.2.9 writeFloat . . . 93
14.2.10 writeDouble . . . 93
14.2.11 writeString . . . 93
14.2.12 writeEnum . . . 94
14.2.13 writeBooleanArray . . . 94
14.2.14 writeByteArray . . . 94
14.2.15 writeShortArray . . . 95
14.2.16 writeIntArray . . . 95
14.2.17 writeLongArray . . . 96
14.2.18 writeFloatArray . . . 96
14.2.19 writeDoubleArray . . . 96
14.2.20 writeStringArray . . . 97
14.2.21 writeEnumArray . . . 97

15 The Fault Class **98**
15.1 **Fields** . . . 98
15.2 Methods . . . 98
15.2.1 createFault . . . 98
15.2.2 destroyFault . . . 99
15.2.3 writeFault . . . 99
15.2.4 toStringFault . . . 100

16 The Alert Class **101**
16.1 **Fields** . . . 101
16.2 Methods . . . 101

16.2.1 createAlert . . . 101
16.2.2 destroyAlert . . . 102
16.2.3 writeAlert . . . 102
16.2.4 toStringAlert . . . 103

17 The Identification Class **104**
17.1 **Fields** . . . 104
17.2 Methods . . . 104
17.2.1 createIdentification . . . 104
17.2.2 destroyIdentification . . . 104
17.2.3 readIdentification . . . 105
17.2.4 writeIdentification . . . 105
17.2.5 toStringIdentification . . . 105

18 The OperatorMessage Class **106**
18.1 **Fields** . . . 106
18.2 Methods . . . 106
18.2.1 createOperatorMessage . . . 106
18.2.2 destroyOperatorMessage . . . 106
18.2.3 writeOperatorMessage . . . 107
18.2.4 toStringOperatorMessage . . . 107

19 The RemoteConnection Class **108**
19.1 **Fields** . . . 108
19.2 Methods . . . 108
19.2.1 createRemoteConnection . . . 108
19.2.2 destroyRemoteConnection . . . 108
19.2.3 readRemoteConnection . . . 109
19.2.4 writeRemoteConnection . . . 109
19.2.5 toStringRemoteConnection . . . 109

20 The Client Class **110**
20.1 **Fields** . . . 110
20.2 Methods . . . 110
20.2.1 createClient . . . 110

20.2.2 destroyClient . . . 110
20.2.3 readClient . . . 111
20.2.4 writeClient . . . 111
20.2.5 toStringClient . . . 111

21 The ControlLogger Class **112**
21.1 **Fields** . . . 112
21.2 Methods . . . 112
21.2.1 createControlLogger . . . 112
21.2.2 destroyControlLogger . . . 113
21.2.3 setBuffersize . . . 113
21.2.4 setThreads . . . 114
21.2.5 **_writeToLog** . . . 114

22 Change History **116**
22.1 Version 0.1 . . . 116

23 Additional information **117**

24 Appendix **118**
24.1 ControlSystem.h . . . 118
24.2 EMSS spreadsheet . . . 147
24.3 Generated code for file: WeatherStation.h . . . 151
24.4 Generated code for file: WeatherStationInterface.c . . . 154
24.5 Implementation file: WeatherStation.c . . . 156
24.6 Test program: file CTestWeatherStation.c . . . 157

1 Overview

1.1 Structure of an MROI system

This document describes a suite of software that provides an interface to a sub-system within the MROI software system. To understand the structure, components and features of this interface and to use it effectively, one must understand the context in which it operates and the problem it is intended to solve.

The software that manages the MROI is organized as a collection of functionally independent systems managed by a centralized Supervisory System. These systems reflect the physical structure of the interferometer and its functionality: unit telescopes, beam relay system, delay lines, beam combining system, automated alignment system, etc. To operate the telescope the Supervisory System must be able to monitor and command all of these systems. Therefore, there must be some mechanism within each system that allows the Supervisory System to accomplish the actions necessary to direct its activities and monitor its results. Within the overall MROI software system the Supervisory System is implemented in Java, but the systems it manages may be implemented in Java or C[1]. This document describes a suite of software that an MROI system implemented in C must use to allow it to properly communicate with the Supervisory System.

The layer of software described in this document does not dictate how a system should be structured or implemented. It merely provides the high-level functionality that all systems must have in order to be managed by the Supervisory System. This high-level functionality provided by a system is understood to be complete; it must include all functions that are required to operate the system. The system may implement private functions that are not intended to be accessible by any outside agent. The system itself may be very complex and may be structured as a collection of sub-systems and, internally, employ these same interface techniques to manage its own sub-systems. However, this is not required; a system is free to use whatever techniques best accomplish its goals.

Figure 1 depicts a system that operates within the MROI system environment as viewed by the Supervisory System. It is identified by a unique name and is of a specified type. It also has a "state"; it is required to implement a very simple state model, which will be described later. It is commanded by a Supervisor, part of the Supervisory System, using a protocol that allows the Supervisor to execute functions within the system that have been identified as being public and externally executable. It also has the functionality to issue faults and alerts, as well as send messages directly to the telescope operator. It can, if necessary, communicate with the Database Manager, also part of the Supervisory System, to get, store or update data from the archive. The system also writes messages to a local log file in a standardized format. Finally, the system has the tools necessary to publish monitor data, which includes engineering data routinely produced to describe its internal conditions and science data, such as images, that result from executing specific commands. The interface layer of software described in this document provides such functionality.

[1]These techniques could be extended to other languages such as C++ or Python, but, so far, this has not been necessary.

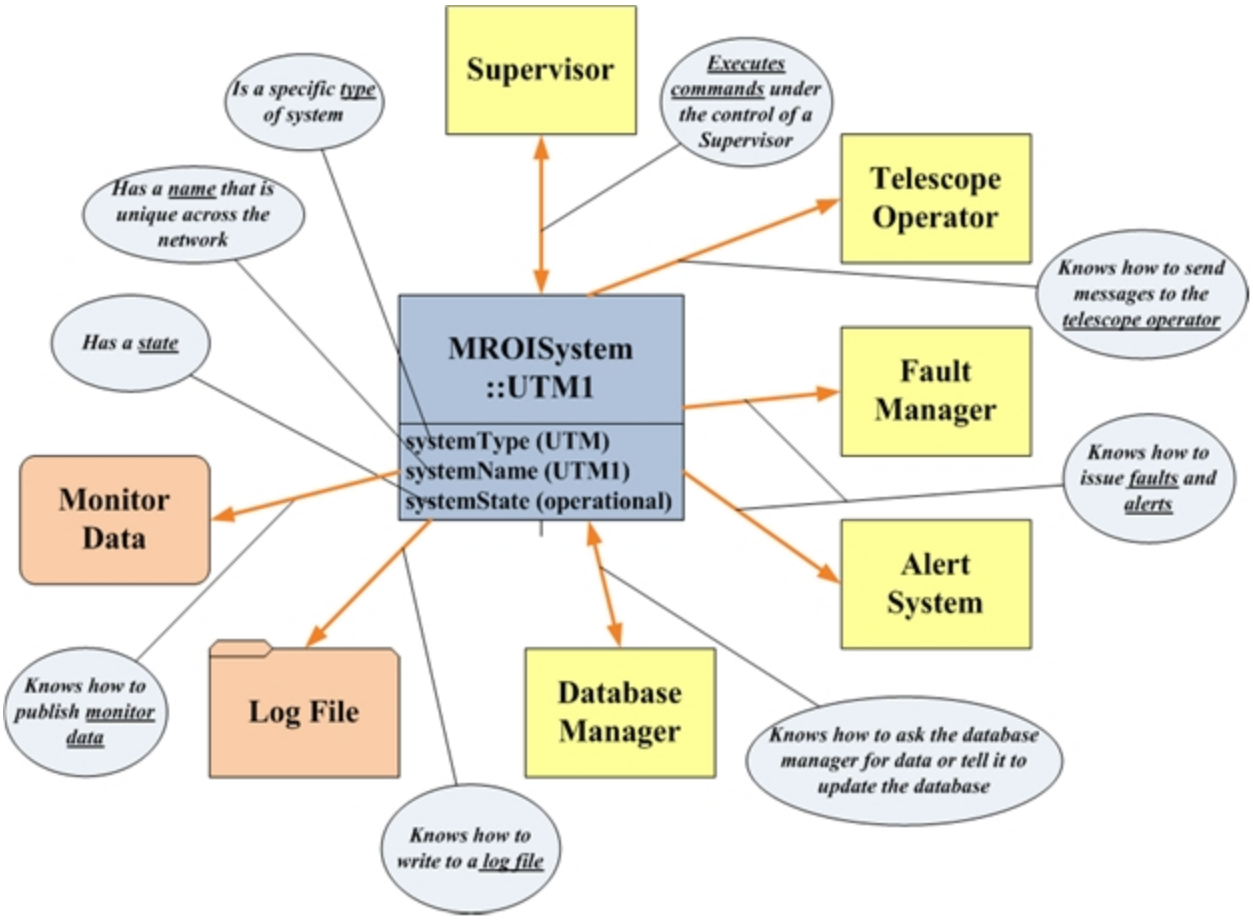

Figure 1: **Structure of an MROI system**

An important point to make about the nature of this interface software is that a system can be operated in a standalone mode without any network interaction. A system that uses this interface can be operated and tested without the Supervisory System or any of its components being present. This aspect of the design facilitates independent system development and testing. In standalone mode the only functionality a system is required to implement are necessary external interactions, such as getting configuration data.

How does a system make its public, externally executable functions available? These functions are not generic in nature, but, rather, are specific to a type of system. Further, what is the required protocol that allows the Supervisor, which may reside on a completely different computer, to execute these functions in the context of a distributed network?

All public, externally executable functions in a C system are of the usual form:

ReturnType **functionName** (Type1 parm1, Type2 parm2, Type3 parm3, . . .);

For each function, there is a unique function name, a series of parameters having the indicated data type, and something that is returned. The returned data may be a single item of data of one of the familiar data types or a structure of data. In this context of a distributed network, it may also return a structure indicating an error occurred while executing the

function. The protocol mentioned above allows a process, called the client, on a remote host to pass a formatted message to the system, called the server, that contains the name of the function and its list of parameters. The server then executes this function, using the supplied parameters. The interface protocol then formats the results and sends this message to the requesting client. The interface software described here handles the protocol, formatting messages, and all of the communications that occur between the client and the server.

The public, externally executable functions in a system are described in a simple spreadsheet. This spreadsheet is then input to a code generation framework that automatically generates the interface software necessary to format the messages that flow between the client and server and execute the proper functions within the system. The functions described in the spreadsheet are divided into two groups: (1) monitor functions that simply return data, take no parameters, and do not alter the state of the system; and, (2) command functions that cause actions in the system to be performed. The command functions take parameters and alter the state of the system. The spreadsheets also describe fault conditions that might be associated with monitored data, as well as the frequency and format of monitor data to be stored in the archive.

The various functions that make up this interface are implemented using an object-oriented approach and uses the language of classes, fields, and methods that belong to those classes. The techniques used for implementing this approach in C are described in the following section. The most important class is called "ControlSystem", which contains the generic functionality described above that is common to all MROI systems. The rest of the classes are used in various ways within the ControlSystem class. A specific type of system within MROI may be thought of as an extension of the ControlSystem class.

1.2 Object-oriented programming in C

This section is not an introduction to object-oriented programming concepts. However, it is an introduction to how to do object-oriented programming in the C language.

Generally speaking, a class[2] is represented as[3]:

```
class ControlLogger {
    // This section declares fields that belong to the class.
    ControlSystem* system;    // The control system to which this logger belongs.
    char* logFilename;        // The log file name associated with this system.
    long int logFile;         // The file handle of the log file.
    ControlException* status; // The current status of the logger.
    bool isThreads;           // If 'isThreads' is true, each write is protected by a lock.
    long int buffersize;      // The size of the buffer that holds the log records.
    char * buffer;            // The allocated buffer, if any, to hold the log records.

    // This section declares methods that belong to the class.
    createControlLogger (char* filename, ControlSystem* system);
    void destroyControlLogger ();
```

[2]This C-like example is not intended to display a correct syntax.

[3]This example is taken from the interface definition of the ControlLogger class.

```
    void setBuffersize (long int buffersize);
    void setThreads ();
    void writeToLog (MROILogLevel logLevel, MROILogType logType, char* message);
};
```

Fields within a class are internal items of data that are used to implement the functionality of the class. These are usually private and are not intended to be directly accessed. Methods are functions that are executed within the context of the class. They may be either private, i.e. only executed internally, or public, i.e. executed by some agent outside the context of the class. There are two special functions associated with a class: the constructor and the destructor. The constructor method, in this case 'createControlLogger', constructs the object by allocating space for all its fields, initializing them, and constructing any objects that are used internally. The destructor, 'destroyControlLogger', destroys the object by destroying all objects used internally and freeing all space currently allocated to the object.

We translate this approach into C in the following manner. The fields within the class are embedded in a C structure. The constructor method is implemented as a C function that allocates and initializes this structure and returns a pointer to the newly allocated structure. This pointer is said to point to an 'object' of the class type. All other methods that belong to the class are implemented as ordinary C functions, but each function takes an additional argument, viz. a pointer to the newly created object, the structure containing the current value of the fields. The functions all operate on this object. Additionally, the concept of 'private' fields or methods is difficult to implement in C, so we use no specific techniques to implement the object-oriented public/private distinction in C.

Using the example above, the C version of this class is:

```
/*
 * Class: ControlLogger
 */
typedef struct _ControlLogger ControlLogger;

struct _ControlLogger {
    // The control system to which this logger belongs.
    ControlSystem* system;
    // The log file name associated with this system.
    char* logFilename;
    // The file handle of the log file.
    long int logFile;
    // The current status of the logger.
    ControlException* status;
    // If 'isThreads' is true, each write is protected by a lock.
    bool isThreads;
    // The size of the buffer that holds the log records.
    long int buffersize;
    // The allocated buffer, if any, to hold the log records.
    char * buffer;
};

ControlLogger* createControlLogger (char* filename, ControlSystem* system);
void destroyControlLogger (ControlLogger* this);
```

```
void setBuffersize (ControlLogger* this, long int buffersize);
void setThreads (ControlLogger* this);
void _writeToLog (ControlLogger* this, MROILogLevel logLevel, MROILogType logType, char* message);
```

The 'createControlLogger' function returns a pointer to the newly created ControlLogger object. The other functions all take a pointer to that object as its first parameter: ControlLogger* this[4]. Since names of functions in C must be unique across the application, special care must be taken to craft function names. The underscore on the 'writeToLog' function is intended to convey that this function is intended to be private and not directly accessible. Likewise, the various fields within the ControlLogger structure should not be accessed directly.

The code above defining the ControlLogger class is usually placed in a '.h' file, along with the definitions of other classes in the application. The code that implements the methods of the class are placed in a file called 'ControlLogger.c'.

All the classes implemented in this interface follow this general pattern.

1.3 Basic system classes

The remaining sections of this document explain in detail the various features and functionality of this interface. The text of the interface definition, ControlSystem.h, is reproduced in the Appendix. If you want to know the precise syntax in C, you may reference this definition as you read the sections.

Section 2 describes the layout of the spreadsheets used to define the high-level interface of a system and the requirements it places on functions in a system. Section 5 contains a list of enumerations used in the interface definition. This list of enumerations may be thought of as providing a basic system-wide vocabulary and it will grow as applications are developed. Section 6 is a list of items that function as extended data types, most of which represent the values of physical quantities in specified units. Section 7 contains the definition of structures that are used in the MROI Monitor and Configuration database (see Reference [3] in section 23) and section 8 contains the definition of several general-purpose functions that are used in the implementation of this interface. In a manner similar to the enumerations, additions will be made to the extended data types and structures as they are needed in developing new applications.

The most important section is section 9, which is the definition of the ControlSystem class. It is complex and contains many methods. Its methods are divided into related groups:

- Constructor and destructor
- Methods for handling exceptions
- Methods for sending faults, alerts, and operator messages

[4]In various object-oriented programming languages it is common to designate such a pointer using the name 'this', meaning this object.

- Methods for setting log characteristics and writing to the log file
- Methods for accessing the Database Manager
- Methods for getting basic system characteristics
- Methods for setting basic system characteristics
- Methods for implementing the system state model
- Methods for implementing data monitoring
- Method related to communications
- Other methods used by the Supervisory System

The remaining sections define classes that are mainly used internally in the ControlSystem class. These include:

- ControlException
- MROISocket
- MROIServerSocket
- SocketInputStream
- SocketOutputStream
- Fault
- Alert
- Identification
- OperatorMessage
- RemoteConnection
- Client
- ControlLogger

2 High-level Interface Definition

The definition of the high-level interface to a system is expressed using a spreadsheet, actually a set of spreadsheets. This definition also imposes requirements on the system to implement the functionality expressed in the spreadsheets. In this section we will describe the contents of the spreadsheets and the constraints on the functions that a C system must implement.

Two spreadsheet applications have been used and tested to create the high-level interface definition of a system: Microsoft's Excel and OpenOffice's Calc. If Excel is used the spreadsheet must be saved using the "XML Spreadsheet 2003" format (".xml" file extension). If Calc is used, the spreadsheet should be saved as an ".ods" file, which is the OpenOffice default format. Either format may be used as input to the code generation process, but OpenOffice is preferred, simply because it is open source.

There are five worksheets contained in the spreadsheet, whose names are:

- **System** *Defines attributes of the system as a whole.*
- **Monitor** *Defines monitor points, which includes engineering data routinely produced to describe internal conditions and science data, such as images, that result from executing specific commands.*
- **Fault** *Defines faults associated with monitor points.*
- **Control** *Defines commands used to initiate various actions within the system.*
- **Parameters** *Defines parameters associated with specific commands or associated with the system as a whole.*

The example in section 3 refers to the spreadsheet in the appendix 24.2, which may be used a template.

In filling out the columns of a worksheet, there is an important point that must be made. The value of a column must not be blank; if a particular column does not apply or there is no value for it, then 'none' should be entered. The purpose of this convention is to simplify the process of parsing the spreadsheet file. Blank column entries are particularly difficult to deal with in the output formats.

In the descriptions of the worksheets that follow, the difference between a 'name' and 'text' is that a name cannot have embedded spaces while text can.

2.1 System worksheet

The System worksheet contains basic data about the system as a whole, including the formal document that serves as the primary description of the system. It is understood that the system conforms to and is an implementation of the requirements and concepts in this document.

There can be more than one system described in a given spreadsheet. For example, a given system, such as the Environmental Monitoring System, may consist of a collection of different types of systems: weather station, all-sky camera, seeing monitor, dust monitor, etc. These can all be described in a single spreadsheet with the subordinate systems being a part of the overall Environmental Monitoring System.

Row 1 must be the name of the worksheet: System Interface Definition.

Row 2 must contain the names of the columns.

The actual values of the columns begin in row 3. The column names and their meaning are:

- **Name** *(name) The short name by which this type of system is known.*
- **Description** *(text) A brief description of the system.*
- **Package** *(name) The package name of this system (only used for Java systems).*
- **Import** *(text) Additional files to be imported for the system (only used for Java systems). This entry can be a series of names separated by spaces.*
- **Full Name** *(text) The full name of this system.*
- **Extends** *(name) The name of a system of which this one is an extension (only used for Java systems).*
- **Parent System** *(name) The name of the system to which this system belongs. This entry allows the definition of multiple systems that are contained within a master system.*
- **Implement** *(no, Java, C, C-no-threads) Whether this system is to be implemented or not; and, if so, whether it is a Java or C system. If it is a C system, the 'C-no-threads' word indicates that the C system does not use threads.*
- **Is Asynchronous** *(yes, no) Does this system implement asynchronous methods?*
- **Is A Monitor** *(yes, no) Does this system produce monitor data?*
- **Work Package** *(text) The MROI work package to which this system belongs.*
- **Document Title** *(text) The title of the document that serves as the primary description of this system.*
- **Document Number** *(text) The MROI document number of the primary document.*
- **Document Issue** *(text) The revision number of the primary document to which this system conforms.*
- **Document Date** *(date) The date of the primary document to which this system conforms.*

2.2 Monitor Worksheet

The Monitor worksheet describes each monitor point within the system. Names of monitor points must be unique within the system to which they belong.

Row 1 must be the name of the worksheet: Monitor Points.

Row 2 must contain the names of the columns.

The actual values of the columns begin in row 3. The column names and their meaning are:

- **Name** *(name) The name of this monitor point.*
- **System** *(name) The name of the system (from the System worksheet) to which this monitor point belongs.*
- **Description** *(text) A brief description of this monitor point.*
- **Returns** *(name) The data type that this monitor point returns.*
- **Can Be Null** *(yes, no) Can this monitor point return a null value?*
- **Throws Exception** *(yes, no) Can this monitor point return an exception?*
- **Asynchronous** *(yes, no) Is this monitor points implemented by an asynchronous method?*
- **Data Unit** *(name) The physical unit that describes this monitor point (this is the unit that is used in archiving this data), called the canonical value.*
- **Minimum Value** *(number) The maximum value of this monitor point.*
- **Maximum Value** *(number) The minimum value of this monitor point.*
- **Default Value** *(number) A default for this monitor point (used in simulations).*
- **System Unit** *(name) The physical unit that is used internally in this system in describing a raw value of this monitor point.*
- **Raw Data Type** *(name) The data type associated with the raw value of this monitor point.*
- **Scale** *(number) The scale used to convert a raw value to a canonical value. The formula is canonical-value = raw-value * scale + offset.*
- **Offset** *(number) The offset used to convert a raw value to a canonical value.*
- **Mode** *(any, diagnostic, operational) The system state in which this monitor point can be executed, usually 'any'. For example, if labeled 'diagnostic' then this monitor point can only be executed in diagnostic mode.*
- **Implement** *(yes, no) Should a method be generated to execute this monitor point?*

- **Archive Interval (secs)** *(name) The interval, in seconds, at which this monitor point should be stored in the archive.*
- **Archive Only On Change** *(yes, no) Should this monitor point be archived only when it changes in value?*
- **Display Unit** *(name) The units in which to display, in a GUI, values retrieved from the archive for this monitor point.*
- **Graph Minimum** *(number) The graph minimum to be used in a GUI display.*
- **Graph Maximum** *(number) The graph maximum to be used in a GUI display.*
- **Graph Title** *(text) The title to be used in a GUI display.*

2.3 Fault Worksheet

The Fault worksheet describes each fault that might be generated by the system. Names of faults must be unique within the system to which they belong.

Row 1 must be the name of the worksheet: Fault Definitions.

Row 2 must contain the names of the columns.

The actual values of the columns begin in row 3. The column names and their meaning are:

- **Fault Name** *(name) The name of this fault condition.*
- **System** *(name) The name of the system (from the System worksheet) to which this fault belongs.*
- **Monitor Point** *(name) The name of the monitor point, if any, (from the Monitor worksheet) to which this fault belongs. A fault condition may be associated with the system as whole, in which case 'none' should be entered.*
- **Description** *(text) A brief description of the fault condition.*
- **Fault Condition** *(text) This item has not been defined.*
- **Fault Severity** *(text) This item has not been defined.*
- **Fault Action** *(text) This item has not been defined.*

2.4 Control Worksheet

The Control worksheet describes each command in the system. Parameters that are associated with these commands are in the next worksheet. Names of commands must be unique within the system to which they belong.

Row 1 must be the name of the worksheet: Control Commands.

Row 2 must contain the names of the columns.

The actual values of the columns begin in row 3. The column names and their meaning are:

- **Name** *(name) The name of this command.*
- **System** *(name) The name of the system (from the System worksheet) to which this command belongs.*
- **Description** *(text) A brief description of this command.*
- **Returns** *(name) The data type that this command returns.*
- **Can Be Null** *(yes, no) Can this command return a null value?*
- **Throws Exception** *(yes, no) Can this command return an exception?*
- **Asynchronous** *(yes, no) Is this command implemented by an asynchronous method?*
- **Mode** *(any, diagnostic, operational) The system state in which this command can be executed, usually 'any'. For example, if labeled 'diagnostic' then this command can only be executed in diagnostic mode.*
- **Implement** *(yes, no) Should a method be generated to execute this command? This item usually 'no'; it might be 'yes' if there are necessary data conversions to be made.*

2.5 Parameters Worksheet

The Parameters worksheet define the parameters that are associated with specific commands or with the system as a whole. Parameter names associated with commands only have to be unique within the context of the command to which they belong. Names of parameters that belong to the system as a whole are required to be unique within that entire context.

Row 1 must be the name of the worksheet: Parameters.

Row 2 must contain the names of the columns.

The actual values of the columns begin in row 3. The column names and their meaning are:

- **Parameter Name** *(name) The name of this parameter.*
- **System** *(name) The name of the system (from the System worksheet) to which this parameter belongs.*
- **Command** *(name) The name of the control command (from the Control worksheet) to which this parameter belongs, if any. If this parameter belongs to the system as a whole, then 'none' should be entered.*
- **Description** *(text) A brief description of this command.*
- **Required** *(yes, no) Is this parameter required?*

- **Data Type** *(name) The data type of this parameter.*
- **Data Unit** *(name) The physical unit that describes this parameter. This is the canonical value, the unit used by an external agent in sending data to this system.*
- **Minimum Value** *(number) The maximum value of this parameter, in canonical units.*
- **Maximum Value** *(number) The minimum value of this parameter, in canonical units.*
- **Default Value** *(number) A default for this parameter (used in simulations), in canonical units.*
- **System Unit** *(name) The physical unit that is used internally in this system in describing a raw value of this parameter.*
- **Raw Data Type** *(name) The data type associated with the raw value of this parameter.*
- **Scale** *(number) The scale used to convert a raw value to a canonical value. The formula is canonical-value = raw-value * scale + offset.*
- **Offset** *(number) The offset used to convert a raw value to a canonical value. The formula is canonical-value = raw-value * scale + offset.*

2.6 Requirements on the system

The previous discussion provides a view of the basic process of defining the high-level interface. The question we will turn to now is: What is required of the system? How does this definition get translated into the actual internal workings of the system?

A detailed example is given in the following section. For now, we will merely make a few introductory remarks that will serve as an overview to this topic.

For each monitor point in the Monitor worksheet or control command in the Control worksheet of the spreadsheet, the system must implement a corresponding function. The first issue is what such a function returns. In general, this function can return: (1) a primitive data type, (2) one of the extended data types listed in section 6, or (3) one of the structures in section 7. The items in these sections will be expanded as necessary. The supported primitive data types in C are:

- **void** *indicates that nothing is returned*
- **bool** *true or false*
- **char** *a signed 8-bit integer*
- **short int** *a signed16-bit integer*

- **long int** *a signed 32-bit integer*
- **long long int** *a signed 64-bit integer*
- **float** *an IEEE 32-bit floating point number*
- **double** *an IEEE 64-bit double precision number*
- **string** *a sequence of characters followed by a null (char* in C)*
- **enumeration** *the numerical value of the enumeration, which is an int in C*

If the spreadsheet entry defining the monitor point or the control command can return an exception (the 'Throws Exception' column is 'yes'), which is usually the case, then the function must contain an additional parameter, viz. a pointer to a ControlException object: "ControlException* err". (The ControlException class is discussed in section 10.) This exception object should initially be cleared and, if an error is encountered in executing the function, the exception should be set using the 'setException' method.

Therefore, if we have a monitor point called 'Temperature42' that returns a temperature, its signature in the system would be:

```
Temperature getTemperature42(System* this, ControlException* err);
```

The parameter 'this' is a pointer to the system object that was created during the system initialization process and is an extension[5] of the 'ControlSystem' class. The monitor point name is the name that in the database; the name of the function that reads the monitor point is is formed by placing 'get' before the monitor point name.

If we have a control command that sets an exposure time in seconds as an integer, its signature in the system would be:

```
void setExposure(System* this, long int seconds, ControlException* err);
```

The above holds for all synchronous functions and for all asynchronous functions in C systems implemented using threads. However, for C systems that do not use threads, asynchronous functions must be implemented differently. We must now explain this issue.

Synchronous and asynchronous functions are really defined from the client's perspective. A synchronous command blocks the client until the function is executed by the server. In other words, it executes the function immediately and returns; the client waits until the server has completed the execution. An asynchronous command is one that usually takes a much longer time to execute; it does not block the client. An asynchronous command is accepted by the server and an acknowledgement is immediately sent to the client. The function is then scheduled to be executed by the server at some later time; so the results of the execution are not sent to the client until a much later time.

[5]The process of "extending" a class will be explained in the example in section 3.

Asynchronous commands are most easily implemented using threads. A thread, whose purpose is to execute the asynchronous function, is created and executed, returning the results whenever it is completed. However, not all C programs, especially those with hard real-time constraints, are implemented using threads. Typically, such programs place the command on a queue to be executed whenever the system has enough time to do it. The handling of such cases requires a slightly different mechanism. For C systems that are implemented without using threads, asynchronous commands require an additional parameter: a pointer to a function that handles whatever is returned by the execution of the command. This mechanism is called a "callback".

Suppose there is a command to activate a small motor that moves a mirror. The command takes the required angular displacement in radians and returns the new position of the mirror when the operation is completed. The function is asynchronous. The signature of such a function within a C system that does not use threads is:

```
void moveMirror(System* this, Angle offset, ControlException* err,
    void (*callback)(System*, Angle, ControlException*));
```

The 'callback' parameter is a pointer to a function of the form:

```
void moveMirrorReturn (System* this, Angle newPosition, ControlException* err);
```

The actual call to the 'moveMirror' function is:

```
moveMirror(this, offset, err, moveMirrorReturn);
```

The 'moveMirror' function must be implemented in the following manner. If this command is not a valid command or if there is any error in any parameter, then the ControlException is set and the function returns immediately; nothing else happens. Otherwise, the system pointer, callback pointer and exception are saved, whatever action is required to move the mirror is initiated, and the function returns, indicating that the function has been accepted. When the action is completed, the callback function is called, in this case 'moveMirrorReturn'. The code generation framework automatically generates the callback function, in this case 'moveMirrorReturn', for the C-no-threads case. However, if the system is being tested in the standalone mode, this function must be supplied.

3 An Example

As an example we will use a weather station within the Environmental Monitoring System. This is not the real weather station; it is a simple example to illustrate the features of the interface. The spreadsheets that define the high-level interface are shown in the appendix in section 24.2.

The System spreadsheet indicates that WeatherStation is a system implemented in C using threads. In the following subsection we will indicate the differences in a system that does not use threads. Three are three monitor points: 'Temperature', 'WindSpeed', and 'WindDirection', which are all synchronous commands. There is a single asynchronous command: 'getAverageWindSpeed', which takes an interval of time in minutes as a parameter. The Parameters worksheet also indicates that there are two parameters that belong to the WeatherStation system and are not part of any command: 'weatherFilename' and 'weatherFile'.

We must explain certain features of the naming convention. The name of the monitor point is the name used to identify this data item in the archive. The names of functions that implement retrieving the associated data are formed by placing the word 'get' before the name of the monitor point. Monitor points do not alter the internal state of the system; they merely retrieve data. Control commands, on the other hand, do alter the state of the system. Their results are not inserted into the archive and their name indicates the function that performs the action. So, their name is not altered in any way; it is the name of the function to accomplish the action. The command 'getAverageWindSpeed' is placed in the Control worksheet because an asynchronous command alters the state of the system, in the sense that resources must be allocated and actions scheduled over a long period of time to perform the function.

The spreadsheet is used as the input to the code generator. Two files are produced: 'WeatherStation.h' and 'WeatherStationInterface.c'. The text of these files are in the appendix in sections 24.3 and 24.4.

First, we will deal with the ".h" file. A structure is defined called 'WeatherStation'. The first portion of this structure is exactly the same as the 'ControlSystem' structure. Fields that are unique to the WeatherStation system are appended to the end of this structure. This way of implementing the WeatherStation structure means that if we have created a WeatherStation object, i.e. a pointer to a WeatherStation structure, the cast:

```
ControlSystem* sys = (ControlSystem*)weatherStation;
```

is always valid. This fact enables us to use a pointer to a WeatherStation object as a parameter to all of the functions that belong to a ControlSystem described in section 9. This technique is the means by which a WeatherStation is said to be an 'extension' of a ControlSystem.

Following the definition of the WeatherSystem structure are the definitions of methods that are unique to the WeatherStation system. These methods, together with those associated with ControlSystem, constitute the high-level interface to the WeatherSystem system. There are two constructors: 'createStandAloneWeatherStation', which creates a WeatherStation in

standalone mode, and 'createWeatherStation', which creates a server that implements the client-server protocol. The first is intended to be used in development and testing. Next there are three methods that are required to implement actions associated with the state model. We will say more about those shortly. The remaining methods are all associated with the monitor points and control commands.

For each monitor point a function definition is generated:

```
Temperature getTemperature(WeatherStation* this, ControlException* err);
```

that retrieves the value of that monitor point. There are two additional functions that are also generated:

```
Duration getTemperatureInterval(WeatherStation* this);
void setTemperatureInterval(WeatherStation* this, Duration temperatureInterval);
```

that allow a user to get and set the interval at with that monitor point is sampled. The remaining function defines the asynchronous method 'getAverageWindSpeed':

```
void getAverageWindSpeed(WeatherStation* this, Duration minutes,
    ControlException* err, void (*callback)(WeatherStation*, Speed, ControlException*));
```

The second file that is generated is 'WeatherStationInterface.c'. It implements methods for all constructors and destructors, as well as the methods that get and set sampling intervals associated with the monitor points[6]. The constructors merely initialize all fields to their default values, as specified in the spreadsheet. They also do one other important thing; they create a log file that is associated with the system. This log file is a text file whose name is the system name to which is appended the current time (to the nearest millisecond), for example:

```
weather1_2010_05_14T17_22_15_968.txt
```

The remaining methods, 'getTemperature', 'getWindSpeed', 'getWindDirection', 'getAverageWindSpeed', and the three action methods required by the state model must be implemented by the WeatherStation application. This brings us to a discussion of the state model.

The state model that a system is required to implement is shown in Figure 2. This model and the constraints it implies is fully implemented within the ControlSystem methods. The only requirement that is imposed on a system is to implement actions required to initialize the system ('initializeWeatherStationAction'), shutdown the system ('shutdownWeatherStationAction'), and save crucial data in the event it is about to be aborted ('aboutToBeAbortedWeatherStationAction'). These methods are required; but, they do not need to actually do anything, if this makes sense within the application. The initialization method should contain whatever actions are required to bring the WeatherStation to an operational state. The ControlSystem portion of the WeatherStation structure contains function pointers to these three methods that are initialized by the constructors when the WeatherStation object

[6]This file also implements various functions that are used internally in implementing the client-server communications protocol, which are not shown in this example.

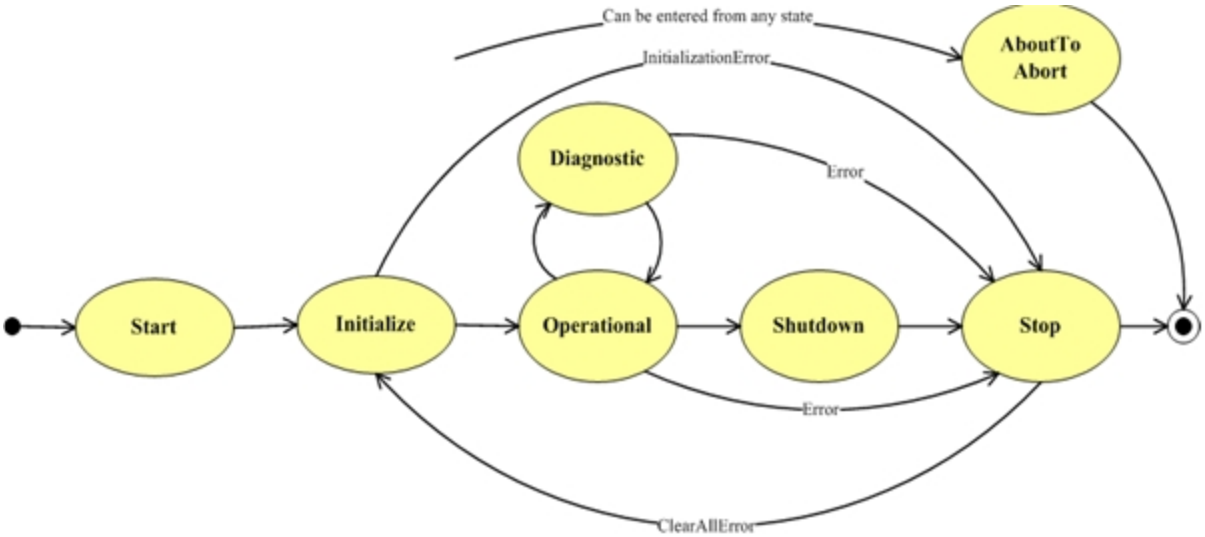

Figure 2: **The Control System state model**

is created. There are actually two sets of these function pointers, one for a C system that uses threads and one for C systems that do not use threads. The reason there are two sets is that the no-threads case requires an additional callback parameter.

When a WeatherStation object is created it is in the UNDEFINED state. In this state the only things the have been done are to initialize the fields and create the log file. The first thing that must be done is to start the system by calling the 'startSystem' method. In the standalone mode this method doesn't really do anything but in the client-server model it creates the server sockets and begins to listen for clients requesting to be connected to the server. At this point the system is in the START state. Next the system is initialized by calling the 'initializeSystem' method. The method performs all actions necessary to bring the system to an operational state. It is assumed that this method is asynchronous and will take some time to complete[7]. At the beginning of the method the system is in the INITIALIZING state; when it has completed the system is in the INITIALIZED state. Following these actions, the system is made operational by call the 'operateSystem' method. The system is then in the OPERATIONAL state. The normal course of action to terminate the system is to call the 'shutdownSystem' method, which is intended to shutdown the system gracefully. However, the system can also be stopped abruptly; before this is done the 'aboutToAbortSystem' method is called to allow systems to save any crucial data.

These methods are illustrated in two additional files in the appendix: 'WeatherStation.c' and 'CTestWeatherStation.c' in sections 24.5 and 24.6. This first corresponds to what would be implemented by the actual Weather Station application. The second corresponds to a standalone test program to test some of the basic features of the system.

The 'WeatherStation.c' file implements the three action methods, the three functions to return the values of the monitor points, and the asynchronous command. The test program

[7]In the client-server mode, as opposed to the standalone mode, a thread is spawned to execute the initialization actions.

performs the following actions. The weather station is created.

```
WeatherStation* weather1 = createStandaloneWeatherStation ("test");
```

Then the weather station is started, initialized, and placed into operation. After each operation the system status is checked for any errors and the current system state is reported.

```
startSystem(weather1);
initializeSystem(weather1);
operateSystem(weather1);
```

Then, a few monitor functions are exercised.

```
ControlException* err = createControlException(weather1);
Temperature t = getTemperature(weather1, err);
Speed s = getWindSpeed(weather1, err);
Angle a = getWindDirection(weather1, err);
```

The system is then shutdown, stopped and destroyed.

```
shutdownSystem(weather1);
stopSystem(weather1);
destroyWeatherStation(weather1);
```

The output from executing the test program is:

```
WeatherStation created in standalone mode.
WeatherStation state: UNDEFINED
WeatherStation state: STARTED
Executing initializeWeatherStationAction
WeatherStation state: INITIALIZED
WeatherStation state: OPERATIONAL
The temperature is  30.00
The wind speed is   4.00
The wind direction is   0.79
Executing shutdownWeatherStationAction
WeatherStation state: SHUTDOWN
WeatherStation state: STOPPED
WeatherStation destroyed.
```

The previous discussion implemented the case of a C system implemented using threads. We now turn to the differences introduced if the case is changed to not use threads[8]. The generated ".h" file is the same except for the signatures of the action methods; these all have callback function pointers.

```
// Actions associated with state changes
    void initializeWeatherStationAction(WeatherStation* , void (*callback)(WeatherStation*));
    void shutdownWeatherStationAction(WeatherStation* , void (*callback)(WeatherStation*));
    void aboutToAbortWeatherStationAction(WeatherStation* , void (*callback)(WeatherStation*));
```

[8]In this case the cell in the System worksheet of the EMSS spreadsheet labeled 'Implement' would be 'C-no-threads'.

In the generated file 'WeatherStationInterface.c' the only difference is the initialization of the function pointers to the action methods. The major difference is in how these action methods are implemented in the application. Each action method is broken into two parts: the first part saves the system and callback pointers and schedules the task to be done at a later time.

```
static void (*CallbackPointer)(WeatherStation*);
static WeatherStation* SystemObject;
void initializeWeatherStationAction(WeatherStation* this, void (*callback)(WeatherStation*)) {
    printf("%s\n", "Executing initializeWeatherStationAction");
    // Do something to save the system pointer and the callback pointer.
    CallbackPointer = callback;
    SystemObject = this;
}
void doInitializeWeatherStationAction(WeatherStation* this) {
    printf("%s\n", "Executing doInitializeWeatherStationAction");
    // Actions to initialize the WeatherStation go here.
    CallbackPointer(SystemObject);
}
```

In the no-threads case the sequence of actions in the test program are:

```
    WeatherStation* weather1 = createStandaloneWeatherStation ("test");
    startSystem(weather1);
    initializeSystem(weather1);
    doInitializeWeatherStationAction(weather1);
    operateSystem(weather1);

    ControlException* err = createControlException(weather1);
    Temperature t = getTemperature(weather1, err);
    Speed s = getWindSpeed(weather1, err);
    Angle a = getWindDirection(weather1, err);

    shutdownSystem(weather1);
    doShutdownWeatherStationAction(weather1);
    stopSystem(weather1);
    destroyWeatherStation(weather1);
```

The output from executing the no-threads version of the test program is:

```
    WeatherStation created in standalone mode.
    WeatherStation state: UNDEFINED
    WeatherStation state: STARTED
    Executing initializeWeatherStationAction
    WeatherStation state: INITIALIZING
    Executing doInitializeWeatherStationAction
    WeatherStation state: INITIALIZED
    WeatherStation state: OPERATIONAL
    The temperature is  30.00
    The wind speed is   4.00
    The wind direction is   0.79
    Executing shutdownWeatherStationAction
    WeatherStation state: SHUTTINGDOWN
```

```
Executing doShutdownWeatherStationAction
WeatherStation state: SHUTDOWN
WeatherStation state: STOPPED
WeatherStation destroyed.
```

4 Communications and Monitoring

This section has not been completed.

5 Enumerations

The enumerations listed below are C versions of the enumerations documented in the MROI Monitor and Configuration Database (see Reference [3] in section 23). The declaration of these enumerations is contained in the file ControlSystem.h listed in the appendix.

5.1 Enumeration: MROIAlertLevel

The alert level indicates the degree of importance attached to the alert. The enumerated values are ordered in decreasing order of significance.

AlertLevel_SEVERE *A severe condition has been detected that demands immediate attention.*

AlertLevel_ERROR *An error has occurred that will possibly affect observations.*

AlertLevel_WARNING *A condition has occurred that might affect adversely affect observations but does not warrant stopping them.*

AlertLevel_INFO *This level is only used to test the mechanism for sending an alert. It is not inserted into the database and no response is necessary.*

5.2 Enumeration: MROIExceptionType

An exception type identifies a type of exception that is thrown during the execution of a system. This list is intended to be easily extensible as systems are further developed.

ExceptionType_UNDEFINED *Only used temporarily in creating exceptions as object created by remote clients.*

ExceptionType_INVALID_REQUEST *The request submitted to a server is not valid.*

ExceptionType_INVALID_PARAMETER *The request submitted to a server contains an invalid parameter.*

ExceptionType_ACTION_FAILED *A request submitted to a server failed to execute.*

ExceptionType_REPLY_ERROR *An error occurred in sending a reply to a client.*

ExceptionType_OBJECT_CREATION_EXCEPTION *An error occurred in creating an object.*

ExceptionType_IO_ERROR *An I/O error, usually in communication between remote clients and servers, has occurred.*

ExceptionType_MEMORY_ALLOCATION_FAILED *Indicates an error allocating memory.*

ExceptionType_NO_ERROR *Used in C systems to indeicate the absence of an exception.*

ExceptionType_NULL_POINTER *Used in C systems to indicate a null pointer.*

ExceptionType_BAD_READ *An I/O error reading some item.*

ExceptionType_UNEXPECTED_EOF *An unexpected end-of-file when reading some item.*

ExceptionType_BUFFER_OVERFLOW *A buffer overflow occured when writing some item.*

5.3 Enumeration: MROILogLevel

The log level enumeration is copied from Java's logging utility. The values are ordered in decreasing significance, or, another way of looking at it is in increasing level of detail. In the Java logging utility one can set the log level at a given level and suppress all messages below that level.

LogLevel_SEVERE *The highest level, indicating a condition that usually stops execution.*

LogLevel_WARNING *An adverse condition that does not terminate execution.*

LogLevel_INFO *Any significant information relevant to the execution of the system.*

LogLevel_CONFIG *Information relevant to the current configuration of the system, usually associated with startup conditions.*

LogLevel_FINE *Debugging information at a high level of detail.*

LogLevel_FINER *Debugging information at a medium level of detail.*

LogLevel_FINEST *Debugging information at a low level of detail.*

5.4 Enumeration: MROILogType

The log type indicates the type of log message, as opposed to the log level, which is associated with the level of detail. It is intended to indicate the purpose of the log message. These enumerations are used by the process that ingests log messages into the database. For example, STATE_CHANGE, FAULT, and ALERT messages are selected for special treatment during the ingest process.

LogType_UNDEFINED *Only used temporarily in creating exceptions as object created by remote clients.*

LogType_STATE_CHANGE *Indicates a change in the state of the object. It may also be used for changes in the sub-states of a system.*

LogType_ERROR *Indicates an error message, which might be an error in a client message to the system*

LogType_LOG_FILE_CREATED *Indicates that a log file has been successfully created.*

LogType_SERVER_SOCKET_CREATED *A server socket has been created.*

LogType_DATA_SOCKET_CREATED *A data socket has been created.*

LogType_EXCEPTION *A exception has occurred and the message contains the exception. It may also be used to record a failed client request and the exception that is sent to the client.*

LogType_FAULT *A fault has occurred and the message contains the fault.*

LogType_ALERT *An alert has occurred and the message contains the alert.*

LogType_OPERATOR_MESSAGE *A message has been sent to the telescope operator.*

LogType_INFO *Indicates an information or debugging message.*

5.5 Enumeration: MROIMessageType

MessageType is an enumeration of types of messages exchanged between servers and clients. An enumerated value of MessageType is the first byte of any message format. Messages are not recorded in the database; this enumeration is only used internally in the communications software between servers and clients.

MessageType_UNKNOWN *Not intended to be used.*

MessageType_SYSTEM_IDENTIFICATION *Used by both clients and servers to identify the system.*

MessageType_SYNCHRONOUS_COMMAND *Indicates a general synchronous command sent by a client to a server.*

MessageType_ASYNCHRONOUS_COMMAND *Indicates a general asynchronous command sent by a client to a server.*

MessageType_EXECUTED *Server to client: a command has been executed successfully.*

MessageType_EXECUTED_NULL *Server to client: a command has been executed successfully but the result was null.*

MessageType_EXCEPTION *Server to client: an executing command threw an exception.*

MessageType_ACCEPTED *Server to client: an asynchronous command has been accepted for execution.*

MessageType_MONITOR_DATA *Indicates monitor data sent from server to client.*

MessageType_GET_SYSTEM_TYPE *Client to server: get the type of system.*

MessageType_GET_PACKAGE_NAME *Client to server: get the package name associated with the system.*

MessageType_GET_SYSTEM_NAME *Client to server: get the name of this system.*

MessageType_GET_HOST_ADDRESS *Client to server: get the address of the computer on which this system is executing.*

MessageType_GET_MAIN_PORT *Client to server: get the main server port number.*

MessageType_GET_BACKLOG *Client to server: get the current value of the backlog parameter.*

MessageType_GET_SO_TIMEOUT *Client to server: get the current value of the so_timeout parameter.*

MessageType_GET_LOG_FILENAME *Client to server: get the name of the log file currently being used.*

MessageType_GET_SYSTEM_STATE *Client to server: get the current state of the system.*

MessageType_GET_DATABASE_MANAGER_CONNECTION *Client to server: get the parameters used in accessing the database manager.*

MessageType_GET_TELESCOPE_OPERATOR_CONNECTION *Client to server: get the parameters used in accessing the telescope operator.*

MessageType_GET_FAULT_MANAGER_CONNECTION *Client to server: get the parameters used in accessing the fault manager.*

MessageType_BREAK_CONNECTION *Client to server: break this connection.*

MessageType_TERMINATE *Client to server: terminate the execution of this system.*

MessageType_TEST *Client to server: Used only to test the communications network.*

MessageType_SET_DATABASE_MANAGER *Client to server: set the parameters used in accessing the database manager.*

MessageType_SET_TELESCOPE_OPERATOR *Client to server: set the parameters used in accessing the telescope operator.*

MessageType_SET_FAULT_MANAGER *Client to server: set the parameters used in accessing the fault manager.*

MessageType_SET_SOTIMEOUT *Client to server: set the so_timeout parameter.*

MessageType_SET_LOGLEVEL *Client to server: set the loglevel parameter used to select the logging filter level.*

MessageType_INITIALIZE_SYSTEM *Client to server: initialize this system in synchronous mode.*

MessageType_BEGIN_INITIALIZE_SYSTEM *Client to server: begin the initialization process, but do not respond as if this were an asynchronous command.*

MessageType_OPERATE_SYSTEM *Client to server: place this system in operational mode.*

MessageType_DIAGNOSTIC_MODE_ON *Client to server: place this system in diagnostic mode.*

MessageType_DIAGNOSTIC_MODE_OFF *Client to server: place this system back in operational mode.*

MessageType_SHUTDOWN_SYSTEM *Client to server: shut down this system in synchronous mode.*

MessageType_BEGIN_SHUTDOWN_SYSTEM *Client to server: begin the shut down process, but do not respond as if this were an asynchronous command.*

MessageType_ABOUT_TO_ABORT_SYSTEM *Client to server: notify this system in synchronous mode that it is about to be aborted.*

MessageType_BEGIN_ABOUT_TO_ABORT_SYSTEM *Client to server: begin the about to abort process, but do not respond as if this were an asynchronous command.*

MessageType_STOP_SYSTEM *Client to server: place this system in the stopped state.*

MessageType_GET_DATAPORT *Client to server: get the port number on which this system reports monitor data.*

MessageType_INITIALIZE_SYSTEM_ASYNC *Client to server: initialize this system in asynchronous mode.*

MessageType_SHUTDOWN_SYSTEM_ASYNC *Client to server: shut down this system in asynchronous mode.*

MessageType_ABOUT_TO_ABORT_SYSTEM_ASYNC *Client to server: this system should execute the about_to_abort process in asynchronous mode.*

MessageType_MONITOR_ON *Client to server: turn data monitoring on.*

MessageType_MONITOR_OFF *Client to server: turn data monitoring off.*

MessageType_IS_MONITORING *Client to server: is data monitoring currently on?*

5.6 Enumeration: MROISystemType

The SystemType enumeration is a list of official names of types of systems within the MROI. All operational instances of systems are instantiations of these basic types. This list of system types will grow as new systems are added to the database.

SystemType_UNKNOWN *Not intended to be used.*

SystemType_Executive *The Executive system within the Supervisory System.*

SystemType_Supervisor *The Supervisor system within the Supervisory System.*

SystemType_FaultManager *The Fault Manager system within the Supervisory System.*

SystemType_DatabaseManager *The Database Manager system within the Supervisory System.*

SystemType_DataCollector *The Data Collector system within the Supervisory System.*

SystemType_TelescopeOperator *The Telescope Operator system within the Supervisory System.*

SystemType_OperatorInterface *The Operator Interface system within the Supervisory System.*

SystemType_UTM *The Unit Telescope Mount.*

SystemType_FTT *The Fast Tip-Tilt System.*

SystemType_SIC *The System Integration Camera system.*

SystemType_WAS *The Wide-Field Acquisition System.*

SystemType_UTE *The Unit Telescope Enclosure.*

SystemType_EnvironmentalMonitoringSystem *The Environmental Monitoring System as a whole.*

SystemType_WeatherStation *A weather station within the Environmental Monitoring System.*

5.7 Enumeration: MROISystemState

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.

SystemState_UNDEFINED *The state after which a system has merely been created as a software object.*

SystemState_STARTED *The system's main thread, in which the server listens for remote clients, has been started but the system has not been initialized.*

SystemState_INITIALIZING *The system is in the process of being initialized.*

SystemState_INITIALIZED *The system has been initialized.*

SystemState_OPERATIONAL *The system is operational.*

SystemState_DIAGNOSTIC *The system is in the diagnostic mode.*

SystemState_SHUTTINGDOWN *The system is in the process of shutting down.*

SystemState_SHUTDOWN *The system has been shut down.*

SystemState_STOPPED *The system has been stopped.*

SystemState_ABORTING *The system is in the process of aborting.*

SystemState_ABORTED *The system has aborted.*

5.8 Enumeration: MROIHardwareType

The HardwareType enumeration is a list of official names of types of hardware within the MROI. All operational instances of hardware are instances of these types. This list of hardware types will grow as new systems are added to the database.

HardwareType_UNKNOWN *Not intended to be used.*

HardwareType_Array *An array of unit telescopes.*

HardwareType_UT *A unit telescope.*

HardwareType_UTM *The Unit Telescope Mount hardware.*

HardwareType_WAS *The Wide-Field Acquisition camera.*

HardwareType_UTE *The Unit Telescope Enclosure hardware.*

HardwareType_STATION *The physical station on which a unit telescope resides.*

HardwareType_FTT *The Fast Tip-Tilt hardware.*

HardwareType_SIC *The system integration camera.*

HardwareType_WeatherStation *A weather station.*

HardwareType_AllSkyCamera *The all-sky camera.*

6 Extended Data Types

The following is a list of items that function as extended data types. These are the C versions of a list of such items in the MROI Monitor and Configuration Database (see Reference [3] in section 23). The declaration of these data types are contained in the file ControlSystem.h listed in the appendix.

All of these items have associated functions that enable them to be written to and read from data streams, as well as being converted to character strings.

Angle *(double) An angle in radians.*

AngularRate *(double) The rate at which an angle changes in radians per second.*

Complex *(double complex) A complex number represented as two double precision numbers.*

Duration *(double) A duration in time in units of nanoseconds.*

FComplex *(float complex) A complex number represented as two single precision numbers.*

Flux *(double) Flux in units of Jansky.*

Frequency *(double) A frequency in units of Hertz.*

Humidity *(double) Relative humidity.*

Length *(double) A length in meters.*

Pressure *(double) Atmospheric pressure.*

Speed *(double) Speed in meters per second.*

Temperature *(double) A temperature in units of degrees centigrade.*

MROITime *(long long int) A time in units of nanoseconds since 2000-01-01T00:00:0.000000.*

MROTime has additional methods to create a time given the year, month, day, hour, minute, and second. One can also create a time from a FITS-formatted string as well as from the current system time.

7 MCDB Structures

There are two data structures, again from the MROI Monitor and Configuration Database (see Reference [3] in section 23), that are included for completeness. They are rarely used. These have associated functions that are constructors, destructors, and that enable them to be written to and read from data streams, as well as being converted to character strings.

The declaration of these structures and associated functions are contained in the file ControlSystem.h listed in the appendix.

7.1 Float Sample

The FloatSample structure represents the measurement of a generic quantity that can be represented as a floating point number. The units associated with the measurement are defined in the monitored property associated with the quantity.

```
typedef struct {
    MROITime time;
    float value;
} FloatSample;
```

7.2 Name Value Pair

The NameValuePair structure represents a named value expressed as a string of characters.

```
typedef struct {
    char* name;
    char* value;
} NameValuePair;
```

8 Additional General Methods

There are a several general purpose data structures and functions that are used internally and included here.

The first of these implements a string buffer.

```
typedef struct {
    int size;
    char* buffer;
    int mark;
} StringBuffer;
```

It has the associated functions:

```
// Create a StringBuffer object of the specified size.
    StringBuffer* createStringBuffer(int size);
// Destroy the specified StringBuffer object.
    void destroyStringBuffer(StringBuffer* this);
// Append a character string to the specified StringBuffer.
    int appendStringBuffer(StringBuffer* this, const char* s);
// Get the current contents of the specified StringBuffer.
    char* getStringBuffer(StringBuffer* this);
// Get the current size of the contents of the StringBuffer.
    int getSizeStringBuffer(StringBuffer* this);
```

Other functions are:

```
// Copy the specified string into a newly allocated space and return that space.
    char* createString(char* s, ControlException* status);
// Convert the specified boolean value to a character string.
    char* toStringBoolean(bool x);
// Convert the specified byte value to a character string.
    char* toStringByte(char x);
// Convert the specified short int value to a character string.
    char* toStringShort(short int x);
// Convert the specified long int value to a character string.
    char* toStringInt(long int x);
// Convert the specified long long int value to a character string.
    char* toStringLong(long long int x);
// Convert the specified float value to a character string.
    char* toStringFloat(float x);
// Convert the specified double value to a character string.
    char* toStringDouble(double x);
```

9 The ControlSystem Class

There is no description.

9.1 Fields

Fields are private and should not be directly accessed.

- **systemType** *(MROISystemType) The name of this type of system*
- **packageName** *(const char*) The name associated with this package*
- **systemName** *(const char*) The name of that identifies this instance of the system*
- **logger** *(ControlLogger*) The logger associated with this system*
- **hostAddress** *(const char*) The address of the host that this system runs on*
- **mainPort** *(long int) The main port on this system's host on which the system listens for connections*
- **backlog** *(long int) The backlog on the ports associated with this system*
- **soTimeout** *(long int) The default timeout, in milliseconds, for the accept() function. This is used after the initial accept and may be reset by the Executive*
- **dataPort** *(long int) The data port on this system's host on which the system listens for connections to the data port*
- **serverSocket** *(MROIServerSocket*) The server socket on the main port*
- **serverDataSocket** *(MROIServerSocket*) The server socket on the data port*
- **telescopeOperatorName** *(char*) The name of the Telescope Operator system*
- **telescopeOperatorIPAddress** *(char*) The IP address of the location of the Telescope Operator system*
- **telescopeOperatorPort** *(long int) The port used to connect to the Telescope Operator system*
- **databaseManagerName** *(char*) The name of the Database Manager system*
- **databaseManagerIPAddress** *(char*) The IP address of the location of the Database Manager system*
- **databaseManagerPort** *(long int) The port used to connect to the Database Manager system*

- **faultManagerName** *(char*) The name of the Fault Manager system*
- **faultManagerIPAddress** *(char*) The IP address of the location of the Fault Manager system*
- **faultManagerPort** *(long int) The port used to connect to the Fault Manager system*
- **numberMonitorPoint** *(long int) The number of monitor points associated with this system*
- **monitorPoint** *(const char**) The names of the monitor points associated with this system*
- **numberCommand** *(long int) The number of commands associated with this system*
- **command** *(const char**) The names of commands associated with this system*
- **systemState** *(MROISystemState) The current state of the system*
- **monitoring** *(bool) Is the system currently producing monitor data?*
- **status** *(ControlException*) The exception status associated with this control system*
- **noThreads** *(bool) Whether this C system is implemented without using threads*
- *(void (*initializeAction)(void*)) A pointer to the system's initialization actions that uses threads*
- *(void (*initializeActionNoThread)(void*, void(*)(void*))) A pointer to the system's initialization actions that does not use threads*
- *(void (*shutdownAction)(void*)) A pointer to the system's shutdown actions that uses threads*
- *(void (*shutdownActionNoThread)(void*, void(*)(void*))) A pointer to the system's shutdown actions that does not use threads*
- *(void (*aboutToAbortAction)(void*)) A pointer to the system's about-to-abort actions that uses threads*
- *(void (*aboutToAbortActionNoThread)(void*, void(*)(void*))) A pointer to the system's about-to-abort actions that does not use threads*

9.2 Methods for constructing and destroying a ControlSystem

9.2.1 _globalError

This method is private and should not be accessed directly.

The globalError method is called if an unrecoverable error occurs that cannot be reported using the exception mechanism, such as encountering a null pointer. This method terminates the entire program abruptly.

Returns:

- **void** *This method returns nothing.*

Parameters:

- **message** *(const char*) A message explaining the global error.*
- **filename** *(const char*) The name of the source code file in which the global error occurred.*
- **lineNumber** *(long int) The line number of the source code file in which the global error occurred.*

Exceptions:

- *This method terminates the entire program abruptly.*

9.2.2 createControlSystem

Constructor: Create a control system.

Returns:

- **ControlSystem*** *A pointer to the newly created ControlSystem object.*

Parameters:

- **systemType** *(MROISystemType) The type of system from the enumeration of system types.*
- **packageName** *(const char*) The package name associated with this system.*
- **systemName** *(const char*) The name of this system instance.*
- **hostAddress** *(const char*) The address of the host on which this system executes.*
- **mainPort** *(long int) The main port on the host on which this system listens for clients.*
- **dataPort** *(long int) The data port on the host on which the system listens for connections to the data port.*
- **backlog** *(long int) The backlog associated with the number of supported clients.*

Exceptions:

- *If memory allocation for the ControlSystem object fails or*
- *the ControlSystem object's ControlException fails to be created, NULL is returned;*
- *otherwise, the ControlSystem object is created and returned.*
- *If any input parameter is not valid, the ControlSystem's exception status is set.*

9.2.3 _initControlSystem

This method is private and should not be accessed directly.

The _initControlSystem is an internal method used to initialize a ControlSystem structure. It is used by C systems that are extensions of the basic ControlSystem.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*) The ControlSystem structure to be initialized.*
- **systemType** *(MROISystemType) The type of system from the enumeration of system types.*
- **packageName** *(const char*) The package name associated with this system.*
- **systemName** *(const char*) The name of this system instance.*
- **hostAddress** *(const char*) The address of the host on which this system executes.*
- **mainPort** *(long int) The main port on the host on which this system listens for clients.*
- **dataPort** *(long int) The data port on the host on which the system listens for connections to the data port.*
- **backlog** *(long int) The backlog associated with the number of supported clients.*

Exceptions:

- *If any input parameter is not valid, the ControlSystem's exception status is set.*

9.2.4 destroyControlSystem

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*) The ControlSystem object that is to be destroyed.*

Exceptions:

- *No exceptions are set by this method.*

9.3 Methods for handling exceptions

9.3.1 isSystemException

There is no description.

Returns:

- **bool** *Returns true if and only if this ControlSystem's status indicates an exception has occurred.*

Parameters:

- **system** *(void*) The ControlSystem object*

Exceptions:

- *We will do exceptions later.*

9.3.2 setSystemException

An exception has occurred. Set this ControlSystem's status to indicate an exception with the sepcified values.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*) The ControlSystem object whose status is being set to indicate an exception.*
- **type** *(MROIExceptionType) The enumerated type of exception.*
- **message** *(const char*) A message explaining the exception.*
- **filename** *(const char*) The name of the source code file in which the exception occurred.*
- **lineNumber** *(long int) The number of the line within the source code file at which the exception occurred.*

Exceptions:

- *If 'this' is null, this method merely returns.*
- *If this ControlSystem's status currently indicates an exception, then the method merely returns.*
- *If internal space has to be reallocated to hold a long message or filename and memory allocation fails, the ControlSystem's status is set and the method merely returns.*

9.3.3 clearSystemException

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*) The ControlSystem object whose status is being cleared.*

Exceptions:

- *We will do exceptions later.*

9.3.4 getSystemException

There is no description.

Returns:

- **ControlException*** *The ControlSystem's exception object.*

Parameters:

- **system** *(void*) The ControlSystem object*

Exceptions:

- *We will do exceptions later.*

9.4 Methods for sending faults, alerts, and operator messages.

9.4.1 sendMROIFault

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*) The ControlSystem object*
- **faultName** *(char*)*
- **monitoredPropertyName** *(char*)*
- **message** *(char*)*
- **numberData** *(int)*
- **data** *(char**)*
- **numberFault** *(int)*
- **faultTree** *(Fault*)*

Exceptions:

- *We will do exceptions later.*

9.4.2 sendMROIAlert

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **alertName** *(char*)*
- **alertLevel** *(MROIAlertLevel)*
- **monitoredPropertyName** *(char*)*
- **message** *(char*)*
- **fault** *(Fault)*

Exceptions:

- *We will do exceptions later.*

9.4.3 sendMROIOperatorMessage

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **message** *(char*)*

Exceptions:

- *We will do exceptions later.*

9.5 Methods for setting log characteristics and writing to the log.

A typical entry into the log file is:

```
FINE: 2009-06-22T17:23:58.233999360 Executive.Executive1 (SHUTTINGDOWN) DEBUG: Main server socket closed.
```

The order of items within the logging entry is:

- *Logging level*
- *Time in FITS format*
- *System type*
- *System name*
- *State of the system at the time the log message was entered*
- *Type of log entry*
- *log message*

This entry is entered into the log file as a single line of text.

9.5.1 getLogFilename

There is no description.

Returns:

- **char*** *The full path name of the current log file.*

Parameters:

- **system** *(void*)*
- **message** *(char*)*

Exceptions:

- *We will do exceptions later.*

9.5.2 setLoggerBuffersize

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **buffersize** *(long int)*

Exceptions:

- *We will do exceptions later.*

9.5.3 setLoggerThreadsOption

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **option** *(bool)*

Exceptions:

- *We will do exceptions later.*

9.5.4 logSevere

Write a log record indicating a severe problem, a condition that usually stops execution.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.5 logWarning

Write a log record indicating an adverse condition that does not terminate execution.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.6 logInfo

Write a log record indicating any significant information relevant to the execution of the system.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.7 logConfig

Write a log record indicating information relevant to the current configuration of the system, usually associated with startup conditions.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.8 logFine

Write a log record indicating debugging information at a high level of detail.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.9 logFiner

Write a log record indicating debugging information at a medium level of detail.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.10 logFinest

Write a log record indicating debugging information at a low level of detail.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.11 logCurrentException

Write a ControlException object to the log. This is written as a 'severe' log entry.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **exception** *(ControlException*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*

9.5.12 getLogger

There is no description.

Returns:

- **ControlLogger*** *Return the current logger.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.6 Methods for accessing the database manager

9.6.1 connectToDatabaseManager

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.6.2 disconnectFromDatabaseManager

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7 Methods for getting basic system characteristics

9.7.1 getSystemType

There is no description.

Returns:

- **MROISystemType** *Return the type of this system.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.2 getPackageName

There is no description.

Returns:

- **char*** *Return the package name associated with this system.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.3 getSystemName

There is no description.

Returns:

- **char*** *Return the name of this system.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.4 getHostAddress

There is no description.

Returns:

- **char*** *Return the host address associated with this system.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.5 getMainPort

There is no description.

Returns:

- **long int** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.6 getDataPort

There is no description.

Returns:

- **long int** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.7 getBacklog

There is no description.

Returns:

- **long int** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.8 getSOTimeout

There is no description.

Returns:

- **long int** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.9 getSystemState

There is no description.

Returns:

- **MROISystemState** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.10 getDatabaseManagerConnection

There is no description.

Returns:

- **RemoteConnection*** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.11 getTelescopeOperatorConnection

There is no description.

Returns:

- **RemoteConnection*** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.7.12 getFaultManagerConnection

There is no description.

Returns:

- **RemoteConnection*** *Return something.*

Parameters:

- **system** *(void*)*

Exceptions:

- *We will do exceptions later.*

9.8 Methods for setting basic system characteristics

9.8.1 setDatabaseManager

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **systemName** *(char*)*
- **ipAddress** *(char*)*
- **port** *(long int)*

Exceptions:

- *We will do exceptions later.*

9.8.2 setTelescopeOperator

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **systemName** *(char*)*
- **ipAddress** *(char*)*
- **port** *(long int)*

Exceptions:

- *We will do exceptions later.*

9.8.3 setFaultManager

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **systemName** *(char*)*
- **ipAddress** *(char*)*
- **port** *(long int)*

Exceptions:

- *We will do exceptions later.*

9.8.4 setSOTimeout

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **timeout** *(long int)*

Exceptions:

- *We will do exceptions later.*

9.8.5 setLogLevel

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **level** *(MROILogLevel)*

Exceptions:

- *We will do exceptions later.*

9.9 Methods for implementing the system state model

9.9.1 startSystem

Start the system. The system must be in the UNDEFINED state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.2 initializeSystem

Initialize the system. The system must be in the STARTED or STOPPED state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed and the system is placed in the STOPPED state.*

9.9.3 beginInitializeSystem

Begin the system initialization process but return immediately. The system must be in the STARTED or STOPPED state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.4 operateSystem

Place the system in the operational state. The system must be in the INITIALIZED state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.5 diagnosticModeOn

Place the system in the diagnostic mode. The system must be in the OPERATIONAL state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.6 diagnosticModeOff

Place the system in the operational mode. The system must be in the DIAGNOSTIC state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.7 shutdownSystem

Shut down the system. The system must be in the OPERATIONAL state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.8 beginShutdownSystem

Begin the system shutdown process but return immediately. The system must be in the OPERATIONAL state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.9 aboutToAbortSystem

The system is about to be aborted; save any crucial data now. The system may be in any state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.10 beginAboutToAbortSystem

The system is about to be aborted. Begin to save any crucial data now; but return immediately. The system may be in any state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.11 stopSystem

Place the system in the stopped state. The system must be in the SHUTDOWN state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.12 initializeSystemAsync

Initialize the system asynchronously. The system must be in the STARTED or STOPPED state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.13 shutdownSystemAsync

Shut down the system asynchronously. The system must be in the OPERATIONAL state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.9.14 aboutToAbortSystemAsync

The system is about to be aborted; save any crucial data asynchronously. The system may be in any state.

Returns:

- **MROISystemState** *The system state upon completion of this action.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An ControlSystem's exception is set if this action failed.*

9.10 Methods for implementing monitoring

9.10.1 monitorOn

Turn on data monitoring.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An exception is set if this action failed.*

9.10.2 monitorOff

Turn off data monitoring.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An exception is set if this action failed.*

9.10.3 isMonitoring

Turn on data monitoring.

Returns:

- **bool** *Return true if this system is monitoring data; otherwise return false.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An exception is set if this action failed.*

9.11 Methods related to communications

9.11.1 breakConnection

There is no description

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An exception is set if this action failed.*

9.11.2 terminate

There is no description

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An exception is set if this action failed.*

9.11.3 test

There is no description

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*

Exceptions:

- *An exception is set if this action failed.*

9.12 Other methods

9.12.1 setControlMonitorPoints

Set the names of the monitor points for this system.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **numberMonitorPoint** *(long int)*
- **monitorPointName** *(const char**)*

Exceptions:

- *An exception is set if this action failed.*

9.12.2 setControlCommands

Set the names of the commmands for this system.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **system** *(void*)*
- **numberCommand** *(long int)*
- **commandName** *(const char**)*

Exceptions:

- *An exception is set if this action failed.*

10 The ControlException Class

The ControlException is an object containing basic data about some exception that has occurred within a system. It contains the system to which the exception belongs, its type and time of creation. In addition to a message explaining the exception, this object contains the name of the source code file and line number of the statement that created the exception.

This ControlException object is created as a "null" exception, indicating that there has been no exception. An exception is indicated by calling the 'setException' method. This ControlException object may be reused, in the sense that its values may be cleared, using the 'clearException' method, and reset, using the 'setException' method again.

A fixed amount of space is allocated for the message (1024 bytes) and filename (512 bytes). This space is reallocated only if the size of the message or filename exceeds its currently allocated space. This space is freed when the exception object is destroyed.

For convenience in checking whether an exception has occurred, the method 'isStatusOK' returns true is there is no current exception and false if an exception has occurred.

10.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) The system to which this exception belongs.*
- **type** *(MROIExceptionType) The enumerated type of exception.*
- **time** *(MROITime) The time at which the exception was created.*
- **filename** *(char*) The name of the source code file containing the module that created the exception.*
- **filenameBuffersize** *(long int) The size of the area reserved for the filename.*
- **lineNumber** *(long int) The line number in the source code file at which the exception was created.*
- **message** *(char*) A message explaining the exception.*
- **messageBuffersize** *(long int) The size of the area reserved for the message.*

10.2 Methods

10.2.1 createControlException

Constructor: Create a new ControlException object that indicates there is no exception.

Returns:

- **ControlException*** *A pointer to the newly created ControlException object.*

Parameters:

- **system** *(void*) The ControlSystem to which this exception belongs.*

Exceptions:

- *If 'system' is null, then NULL is returned.*
- *If there is an error allocating memory, the system's status is set to indicate an exception and NULL is returned.*

10.2.2 destroyControlException

Destructor: Destroy the specified ControlException object.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlException*) A pointer to the ControlException that is to be destroyed.*

Exceptions:

- *No exceptions are set by this method.*

10.2.3 setException

An exception has occurred. Set this ControlException's values to the specified parameters.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlException*) The ControlException object whose values are to be set.*
- **type** *(MROIExceptionType) The enumerated type of exception.*
- **message** *(const char*) A message explaining the exception.*
- **filename** *(const char*) The name of the source code file in which the exception occurred.*

- **lineNumber** *(long int) The number of the line within the source code file at which the exception occurred.*

Exceptions:

- *If 'this' is null, this method merely returns.*
- *If this ControlException currently indicates an error, then the system's status is set and the method merely returns.*
- *If internal space has to be reallocated to hold a long message or filename and memory allocation fails, the system's status is set and the method merely returns.*

10.2.4 readControlException

Read this exception's values from the specified input stream.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlException*)*
- **in** *(SocketInputStream*)*

Exceptions:

- *If 'in' is null, the method does nothing and merely returns.*
- *If 'this' is null, the input stream's status is set to indicate an exception.*
- *If 'in' currently indicates an exception, the input stream's socket status is set to indicate an exception has occurred.*
- *If an error occurrs reading any item of data from the input stream, the input stream's status is set.*
- *If memory has to be reallocated and there is an allocation failure, the ControlException's status is set.*

10.2.5 writeControlException

Write this ContorlException's values to the specified output stream.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlException*)*
- **out** *(SocketOutputStream*)*

Exceptions:

- *If 'out' is null, the method doe nothing and merely returns.*
- *If 'this' is null, the output stream's status is set to indicate an exception.*
- *If an error occurrs writing any item of data to the output stream, the output stream's status is set and the method returns.*

10.2.6 toStringControlException

Convert this exception to a character string

Returns:

- **char*** *A character string containing the values of this ControlException.*

Parameters:

- **this** *(ControlException*) The ControlException object that is to be converted.*

Exceptions:

- *No exceptions are set by this method.*

10.2.7 isStatusOK

Is this current exception's status clear, i.e. is there an exception?

Returns:

- **bool** *True if and only if no exception has occurred.*

Parameters:

- **this** *(ControlException*) The ControlException object whose status is being examined.*

Exceptions:

- *No exceptions are set by this method.*

10.2.8 clearException

Clear this exception's values

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlException*) The ControlException object whose values are being cleared.*

Exceptions:

- *No exceptions are set by this method.*

10.2.9 getExceptionType

Return this ControlException's type

Returns:

- **MROIExceptionType** *The enumerated exception type of this ControlException object.*

Parameters:

- **this** *(ControlException*) The ControlException object whose exception type is being returned.*

Exceptions:

- *No exceptions are set by this method.*

10.2.10 getExceptionMessage

Return this ControlException's message

Returns:

- **const char*** *The message of this ControlException object.*

Parameters:

- **this** *(ControlException*) The ControlException object whose message is being returned.*

Exceptions:

- *No exceptions are set by this method.*

10.2.11 getExceptionTime

Return this ControlException's time of creation

Returns:

- **MROITime** *The time this ControlException object was created.*

Parameters:

- **this** *(ControlException*) The ControlException object whose time of creation is being returned.*

Exceptions:

- *No exceptions are set by this method.*

10.2.12 getExceptionFilename

Return this ControlException's filename

Returns:

- **const char*** *The source code filename that created this ControlException object.*

Parameters:

- **this** *(ControlException*) The ControlException object whose filename is being returned.*

Exceptions:

- *No exceptions are set by this method.*

10.2.13 getExceptionLine

Return this ControlException's line number

Returns:

- **long int** *The line number of the source code filename that created this ControlException object.*

Parameters:

- **this** *(ControlException*) The ControlException object whose line number is being returned.*

Exceptions:

- *No exceptions are set by this method.*

11 The MROISocket Class

There is no description.

11.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) The control system to which this MROISocket belongs.*
- **ipAddress** *(const char*) This socket's IP address.*
- **port** *(long int) This socket's port number.*
- **receiveBuffersize** *(long int) The size, in bytes, of the buffer to receive data.*
- **sendBuffersize** *(long int) The size, in bytes, of the buffer to send data.*
- **soTimeout** *(long int) The soTimeout parameter associated with this socket.*
- **socketFD** *(long int) The socketFD parameter associated with this socket.*
- **in** *(SocketInputStream*) The input stream buffer.*
- **out** *(SocketOutputStream*) The output stream buffer.*
- **status** *(ControlException*) The exception that is association with this socket.*

11.2 Methods

11.2.1 createMROISocket

There is no description.

Returns:

- **MROISocket***

Parameters:

- **system** *(ControlSystem*)*
- **port** *(long int)*
- **receiveBuffersize** *(long int)*
- **sendBuffersize** *(long int)*
- **soTimeout** *(long int)*

Exceptions:

-

11.2.2 destroyMROISocket

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(MROISocket*)*

Exceptions:

-

11.2.3 getSocketInputStream

There is no description.

Returns:

- **SocketInputStream***

Parameters:

- **this** *(MROISocket*)*

Exceptions:

-

11.2.4 getSocketOutputStream

There is no description.

Returns:

- **SocketOutputStream***

Parameters:

- **this** *(MROISocket*)*

Exceptions:

-

12 The MROIServerSocket Class

There is no description.

12.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) The control system to which this MROISocket belongs.*
- **ipAddress** *(const char*) This socket's IP address.*
- **port** *(long int) This socket's port number.*
- **backlog** *(long int) The backlog parameter associated with this socket.*
- **soServerTimeout** *(long int) The soTimeout parameter associated with this socket.*
- **status** *(ControlException*) The exception that is association with this socket.*

12.2 Methods

12.2.1 createMROIServerSocket

There is no description.

Returns:

- **MROIServerSocket***

Parameters:

- **system** *(ControlSystem*)*
- **port** *(long int)*
- **backlog** *(long int)*
- **soServerTimeout** *(long int)*

Exceptions:

-

12.2.2 destroyMROIServerSocket

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(MROIServerSocket*)*

Exceptions:

-

12.2.3 acceptMROIServerSocket

There is no description.

Returns:

- **MROISocket***

Parameters:

- **this** *(MROIServerSocket*)*
- **receiveBuffersize** *(long int)*
- **sendBuffersize** *(long int)*
- **soTimeout** *(long int)*

Exceptions:

-

13 The SocketInputStream Class

There is no description.

13.1 Fields

Fields are private and should not be accessed directly.

- **socket** *(MROISocket*) The socket to which this SocketInputStream belongs.*
- **buffersize** *(long int) The size of the buffer (default is 64K).*
- **buffer** *(char*) The allocated buffer.*
- **size** *(long int) The current number of bytes in the buffer.*
- **mark** *(long int) The current position of the read pointer.*
- **status** *(ControlException*) The status associated with any operation.*

13.2 Methods

13.2.1 createSocketInputStream

There is no description.

Returns:

- **SocketInputStream***

Parameters:

- **socket** *(MROISocket*)*

Exceptions:

-

13.2.2 destroySocketInputStream

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(SocketInputStream*)*

Exceptions:

-

13.2.3 receiveSocketInputStream

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.4 readBoolean

There is no description.

Returns:

- **bool**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.5 readByte

There is no description.

Returns:

- **char**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.6 readShort

There is no description.

Returns:

- **short int**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.7 readInt

There is no description.

Returns:

- **long int**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.8 readLong

There is no description.

Returns:

- **long long int**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.9 readFloat

There is no description.

Returns:

- **float**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.10 readDouble

There is no description.

Returns:

- **double**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.11 readString

There is no description.

Returns:

- **char***

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.12 readEnum

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*

Exceptions:

-

13.2.13 readBooleanArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(bool*)*

Exceptions:

-

13.2.14 readByteArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(char*)*

Exceptions:

-

13.2.15 readShortArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(short int*)*

Exceptions:

-

13.2.16 readIntArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(long int*)*

Exceptions:

-

13.2.17 readLongArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(long long int*)*

Exceptions:

-

13.2.18 readFloatArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(float*)*

Exceptions:

-

13.2.19 readDoubleArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(double*)*

Exceptions:

-

13.2.20 readStringArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(char**)*

Exceptions:

-

13.2.21 readEnumArray

There is no description.

Returns:

- **int**

Parameters:

- **in** *(SocketInputStream*)*
- **maxNumber** *(int)*
- **data** *(int*)*

Exceptions:

-

14 The SocketOutputStream Class

There is no description.

14.1 Fields

Fields are private and should not be accessed directly.

- **socket** *(MROISocket*) The socket to which this SocketOutputStream belongs.*
- **buffersize** *(long int) The size of the buffer (default is 64K).*
- **buffer** *(char*) The allocated buffer.*
- **mark** *(long int) The current position of the write pointer.*
- **status** *(ControlException*) The status associated with any operation.*

14.2 Methods

14.2.1 createSocketOutputStream

There is no description.

Returns:

- **SocketOutputStream***

Parameters:

- **socket** *(MROISocket*)*

Exceptions:

-

14.2.2 destroySocketOutputStream

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(SocketOutputStream*)*

Exceptions:

-

14.2.3 sendSocketOutputStream

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*

Exceptions:

-

14.2.4 writeBoolean

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(bool)*

Exceptions:

-

14.2.5 writeByte

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(char)*

Exceptions:

-

14.2.6 writeShort

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(short int)*

Exceptions:

-

14.2.7 writeInt

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(long int)*

Exceptions:

-

14.2.8 writeLong

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(long long int)*

Exceptions:

-

14.2.9 writeFloat

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(float)*

Exceptions:

-

14.2.10 writeDouble

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(double)*

Exceptions:

-

14.2.11 writeString

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(const char*)*

Exceptions:

-

14.2.12 writeEnum

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(int)*

Exceptions:

-

14.2.13 writeBooleanArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(bool*)*

Exceptions:

-

14.2.14 writeByteArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(char*)*

Exceptions:

-

14.2.15 writeShortArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(short int*)*

Exceptions:

-

14.2.16 writeIntArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(long int*)*

Exceptions:

-

14.2.17 writeLongArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(long long int*)*

Exceptions:

-

14.2.18 writeFloatArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(float*)*

Exceptions:

-

14.2.19 writeDoubleArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(double*)*

Exceptions:

-

14.2.20 writeStringArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(char**)*

Exceptions:

-

14.2.21 writeEnumArray

There is no description.

Returns:

- **int**

Parameters:

- **out** *(SocketOutputStream*)*
- **number** *(int)*
- **data** *(int*)*

Exceptions:

-

15 The Fault Class

There is no description.

15.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) description*
- **faultName** *(char*) description*
- **faultTime** *(MROITime) description*
- **stateAtTimeOfFault** *(MROISystemState) description*
- **monitoredPropertyName** *(char*) description*
- **message** *(char*) description*
- **numberData** *(long int) description*
- **data** *(char**) description*
- **numberFault** *(long int) description*
- **faultTree** *(Fault**) description*

15.2 Methods

15.2.1 createFault

There is no description.

Returns:

- **Fault***

Parameters:

- **system** *(ControlSystem*)*
- **faultName** *(char*)*
- **stateAtTimeOfFault** *(MROISystemState)*
- **monitoredPropertyName** *(char*)*

- **message** *(char*)*
- **numberData** *(int)*
- **data** *(char**)*
- **numberFault** *(int)*
- **faultTree** *(Fault**)*

Exceptions:

-

15.2.2 destroyFault

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(Fault*)*

Exceptions:

-

15.2.3 writeFault

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(Fault*)*
- **out** *(SocketOutputStream*)*

Exceptions:

-

15.2.4 toStringFault

There is no description.

Returns:

- **char***

Parameters:

- **this** *(Fault*)*

Exceptions:

-

16 The Alert Class

There is no description.

16.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) description*
- **systemType** *(MROISystemType) description*
- **systemName** *(char*) description*
- **systemState** *(MROISystemState) description*
- **alertName** *(char*) description*
- **alertLevel** *(MROIAlertLevel) description*
- **alertTime** *(MROITime) description*
- **monitoredPropertyName** *(char*) description*
- **message** *(char*) description*
- **fault** *(Fault*) description*

16.2 Methods

16.2.1 createAlert

There is no description.

Returns:

- **Alert***

Parameters:

- **system** *(ControlSystem*) description*
- **alertName** *(char*) description*
- **alertLevel** *(MROIAlertLevel) description*
- **monitoredPropertyName** *(char*) description*

- **message** *(char*) description*
- **fault** *(Fault) description*

Exceptions:

-

16.2.2 destroyAlert

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(Alert*)*

Exceptions:

-

16.2.3 writeAlert

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(Alert*)*
- **out** *(SocketOutputStream*)*

Exceptions:

-

16.2.4 toStringAlert

There is no description.

Returns:

- **char***

Parameters:

- **this** *(Alert*)*

Exceptions:

-

17 The Identification Class

There is no description.

17.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) description*
- **toDo** *(int) TODO*

17.2 Methods

17.2.1 createIdentification

There is no description.

Returns:

- **Identification***

Parameters: There are no parameters for this method.

Exceptions:

-

17.2.2 destroyIdentification

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(Identification*)*

Exceptions:

-

17.2.3 readIdentification

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **in** *(SocketInputStream*)*
- **data** *(Identification*)*

Exceptions:

-

17.2.4 writeIdentification

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(Identification*)*

Exceptions:

-

17.2.5 toStringIdentification

There is no description.

Returns:

- **char***

Parameters:

- **this** *(Identification*)*

Exceptions:

-

18 The OperatorMessage Class

There is no description.

18.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) description*
- **systemId** *(Identification*) description*
- **time** *(MROITime) description*
- **message** *(char*) description*

18.2 Methods

18.2.1 createOperatorMessage

There is no description.

Returns:

- **OperatorMessage***

Parameters:

- **system** *(ControlSystem*)*
- **message** *(char*)*

Exceptions:

-

18.2.2 destroyOperatorMessage

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(OperatorMessage*)*

Exceptions:

-

18.2.3 writeOperatorMessage

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(OperatorMessage*)*
- **out** *(SocketOutputStream*)*

Exceptions:

-

18.2.4 toStringOperatorMessage

There is no description.

Returns:

- **char***

Parameters:

- **this** *(OperatorMessage*)*

Exceptions:

-

19 The RemoteConnection Class

There is no description.

19.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) description*
- **toDo** *(int) TODO*

19.2 Methods

19.2.1 createRemoteConnection

There is no description.

Returns:

- **RemoteConnection***

Parameters: There are no parameters for this method.

Exceptions:

-

19.2.2 destroyRemoteConnection

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(RemoteConnection*)*

Exceptions:

-

19.2.3 readRemoteConnection

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **in** *(SocketInputStream*)*
- **data** *(RemoteConnection*)*

Exceptions:

-

19.2.4 writeRemoteConnection

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(RemoteConnection*)*

Exceptions:

-

19.2.5 toStringRemoteConnection

There is no description.

Returns:

- **char***

Parameters:

- **this** *(RemoteConnection*)*

Exceptions:

-

20 The Client Class

There is no description.

20.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) description*
- **toDo** *(int) TODO*

20.2 Methods

20.2.1 createClient

There is no description.

Returns:

- **Client***

Parameters: There are no parameters for this method.

Exceptions:

-

20.2.2 destroyClient

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(Client*)*

Exceptions:

-

20.2.3 readClient

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **in** *(SocketInputStream*)*
- **data** *(Client*)*

Exceptions:

-

20.2.4 writeClient

There is no description.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **out** *(SocketOutputStream*)*
- **data** *(Client*)*

Exceptions:

-

20.2.5 toStringClient

There is no description.

Returns:

- **char***

Parameters:

- **this** *(Client*)*

Exceptions:

-

21 The ControlLogger Class

The ControlLogger is a class that creates an object that manages a log file. The model is Java's logger. See java.util.logging.Logger.

The public methods are: 'logSevere', 'logWarning', 'logInfo', 'logConfig', 'logFine', 'logFiner', 'logFinest', 'logException'.

The logger is created to write each log entry directly to the file and to assume that the application is not using threads. These two assumptions may be changed by setting a buffer size (method 'setBufferSize') and by setting the threads option (method 'setThreads').

21.1 Fields

Fields are private and should not be accessed directly.

- **system** *(ControlSystem*) The control system to which this logger belongs.*
- **logFilename** *(char*) The log file name associated with this system.*
- **logFile** *(long int) The file handle of the log file.*
- **status** *(ControlException*) The current status of the logger.*
- **isThreads** *(bool) If 'isThreads' is true, each write is protected by a lock.*
- **buffersize** *(long int) The size of the buffer that holds the log records.*
- **buffer** *(char *) The allocated buffer, if any, to hold the log records.*

21.2 Methods

21.2.1 createControlLogger

Constructor: The createControlLogger method creates a ControlLogger object, as well as creating the log file and opening it. It also creates a ControlException object; this status object is always set after any operation.

The ControlLogger object is created to write each log entry directly to the file and to assume that the application is not using threads. If these options are to be changed, then the 'setBuffersize' or 'setThreads' methods must be called prior to writing any log records.

Returns:

- **ControlLogger*** *A pointer to the ControlLogger object that has been newly created.*

Parameters:

- **filename** *(char*) The full path name of the log file to be created.*
- **system** *(ControlSystem*) The ControlSystem to which this log file belongs.*

Exceptions:

- *If the 'system' parameter is null, NULL is returned and nothing else happens.*
- *If there is any error in allocating memory or in creating and opening the log file, the system's status is set and NULL is returned.*

21.2.2 destroyControlLogger

Destructor: Free all memory allocated to this ControlLogger and set 'this' to NULL.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlLogger*) The ControlLogger object that is to be destroyed.*

Exceptions:

- *No exceptions are set by this method.*

21.2.3 setBuffersize

Allocate a buffer for the log records. If the buffer is allocated, writes to the log are written to the buffer. The buffer is written to the file only when there is an overflow condition. If the buffersize parameter is 0, each write to the log is written directly to the file.

The 'setBuffersize' method may be called at any time during the execution of the application. If there is an existing buffer, its contents are written to the log file and the old buffer is freed before allocating a new buffer.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlLogger*) The ControlLogger object whose buffer is being allocated.*
- **buffersize** *(long int) The size, in bytes, of the buffer being allocated to the ControlLogger.*

Exceptions:

- *If the buffersize parameter is less than 0, this ControlException's status is set.*
- *If there is an error allocating memory for the buffer, this ControlException's status is set.*

21.2.4 setThreads

Set the 'threads' option. This options indicates that there are threads in the application. The result is that there is a lock that must be acquired in performing a write to the log.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlLogger*) The ControlLogger object whose 'threads' option is being set to true.*

Exceptions:

- *No exceptions are set by this method.*

21.2.5 _writeToLog

This method is private and should not be accessed directly.

The '_writeToLog' method is an internal method designed to write log records to the log file. It should not be used by an application. It is used by all of the public methods that are used to write particular types of log records. If the ControlLogger's status indicates an error, this method sets the ControlSystem's status and returns; nothing is written to the file.

Returns:

- **void** *This method does not return anything.*

Parameters:

- **this** *(ControlLogger*)*
- **logLevel** *(MROILogLevel)*
- **logType** *(MROILogType)*
- **message** *(char*)*

Exceptions:

- *If there is an error writing to the log, the ControlLogger's status is set.*
- *If there is an attempt to write to the log when the ControlLogger's status indicates an exception, nothing is written to the file and the system's status is set.*

22 Change History

22.1 Version 0.1

Version 0.1 is the initial version of this document.

Version 0.2 added an example system, its interface definition via spreadsheets and generated code. It also included implementation and a test program.

23 Additional information

References

[1] A. Farris, *MROI Supervisory System A Conceptual Design Overview*, internal document (INT-409-ENG-0010 rev 1.1, WP 4.09.03, MROI Supervisory System), September 14, 2009.

[2] A. Farris, *The Design of an MROI System*, internal document (INT-409-ENG-0020 rev 1.0, WP 4.09.01, MROI Software Engineering), September 13, 2009.

[3] A. Farris, *The MROI Monitor and Configuration Database*, internal document (INT-409-ENG-0030 rev 1.3, WP 4.09.04, MROI Data Handling System), February 8, 2010.

[4] 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 (INT-409-ENG-0040 rev 1.0, WP 4.09.01, MROI Software Engineering), September 17, 2009.

[5] A. Farris, *Xpand: A Java-based code generation framework*, internal document (INT-409-ENG-0050 rev 1.0, WP 4.09.01, MROI Software Engineering), January 20, 2010.

[6] A. Farris, *MROI Data Collector*, internal document (INT-409-ENG-0060 rev 0.1, WP 4.09.03, MROI Supervisory System), February 5, 2010.

24 Appendix

24.1 ControlSystem.h

```
#ifndef MROI_CONTROL_SYSTEM_H
#define MROI_CONTROL_SYSTEM_H
/**
 * New Mexico Institute of Mining and Technology
 * 801 Leroy Place
 * Socorro, NM  87801 USA
 * (c) 2009, 2010 by New Mexico Institute of Mining and Technology.
 * (c) Copyright 2009, 2010 by New Mexico Institute of Mining and Technology.
 * All rights reserved.
 *
 * This library is free software; you can redistribute it and/or modify it under
 * the terms of the GNU Lesser General Public License as published by the Free
 * Software Foundation; either version 2.1 of the License, or (at your option)
 * any later version.
 *
 * This library is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
 * details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this library; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA.
 *
 * File MROIControlSystem.h
 *
 * ////////////////////////////////////////////////////////////////
 * // WARNING!  DO NOT MODIFY THIS FILE!
 * //
 * // This is generated code!  Do not modify this file.
 * // Any changes will be lost when the file is re-generated.
 * //
 * ////////////////////////////////////////////////////////////////
 *
 */

# include <stdbool.h>
# include <complex.h>

/**
 ************************************************************************
 * Forward declarations for structures used in Control System
 ************************************************************************
 */

typedef struct _ControlException ControlException;
typedef struct _MROISocket MROISocket;
typedef struct _MROIServerSocket MROIServerSocket;
typedef struct _SocketInputStream SocketInputStream;
```

```
typedef struct _SocketOutputStream SocketOutputStream;
typedef struct _Fault Fault;
typedef struct _Alert Alert;
typedef struct _Identification Identification;
typedef struct _OperatorMessage OperatorMessage;
typedef struct _RemoteConnection RemoteConnection;
typedef struct _Client Client;
typedef struct _ControlLogger ControlLogger;
typedef struct _ControlSystem ControlSystem;

/*
 ********************************************************************
 * Enumerations for Project MROI Data Model MCDB
 ********************************************************************
 */

/*
 * Enumeration AlertLevel
 */
typedef enum {
    AlertLevel_SEVERE = 0 ,
    AlertLevel_ERROR  ,
    AlertLevel_WARNING  ,
    AlertLevel_INFO
} MROIAlertLevel;
char* MROIAlertLevelName[4];
long int MROIAlertLevelNumber;
char* toStringMROIAlertLevel(MROIAlertLevel x);

/*
 * Enumeration ExceptionType
 */
typedef enum {
    ExceptionType_UNDEFINED = 0 ,
    ExceptionType_INVALID_REQUEST  ,
    ExceptionType_INVALID_PARAMETER  ,
    ExceptionType_ACTION_FAILED  ,
    ExceptionType_REPLY_ERROR  ,
    ExceptionType_OBJECT_CREATION_EXCEPTION  ,
    ExceptionType_IO_ERROR  ,
    ExceptionType_MEMORY_ALLOCATION_FAILED  ,
    ExceptionType_NO_ERROR  ,
    ExceptionType_NULL_POINTER  ,
    ExceptionType_BAD_READ  ,
    ExceptionType_UNEXPECTED_EOF  ,
    ExceptionType_BUFFER_OVERFLOW
} MROIExceptionType;
char* MROIExceptionTypeName[13];
long int MROIExceptionTypeNumber;
char* toStringMROIExceptionType(MROIExceptionType x);

/*
 * Enumeration LogLevel
 */
```

```
typedef enum {
    LogLevel_SEVERE = 0 ,
    LogLevel_WARNING  ,
    LogLevel_INFO  ,
    LogLevel_CONFIG  ,
    LogLevel_FINE  ,
    LogLevel_FINER  ,
    LogLevel_FINEST
} MROILogLevel;
char* MROILogLevelName[7];
long int MROILogLevelNumber;
char* toStringMROILogLevel(MROILogLevel x);

/*
 * Enumeration LogType
 */
typedef enum {
    LogType_UNDEFINED = 0 ,
    LogType_STATE_CHANGE  ,
    LogType_ERROR  ,
    LogType_LOG_FILE_CREATED  ,
    LogType_SERVER_SOCKET_CREATED  ,
    LogType_DATA_SOCKET_CREATED  ,
    LogType_EXCEPTION  ,
    LogType_FAULT  ,
    LogType_ALERT  ,
    LogType_OPERATOR_MESSAGE  ,
    LogType_INFO
} MROILogType;
char* MROILogTypeName[11];
long int MROILogTypeNumber;
char* toStringMROILogType(MROILogType x);

/*
 * Enumeration MessageType
 */
typedef enum {
    MessageType_UNKNOWN = 0 ,
    MessageType_SYSTEM_IDENTIFICATION  ,
    MessageType_SYNCHRONOUS_COMMAND  ,
    MessageType_ASYNCHRONOUS_COMMAND  ,
    MessageType_EXECUTED  ,
    MessageType_EXECUTED_NULL  ,
    MessageType_EXCEPTION  ,
    MessageType_ACCEPTED  ,
    MessageType_MONITOR_DATA  ,
    MessageType_GET_SYSTEM_TYPE  ,
    MessageType_GET_PACKAGE_NAME  ,
    MessageType_GET_SYSTEM_NAME  ,
    MessageType_GET_HOST_ADDRESS  ,
    MessageType_GET_MAIN_PORT  ,
    MessageType_GET_BACKLOG  ,
    MessageType_GET_SO_TIMEOUT  ,
    MessageType_GET_LOG_FILENAME  ,
```

```
    MessageType_GET_SYSTEM_STATE  ,
    MessageType_GET_DATABASE_MANAGER_CONNECTION  ,
    MessageType_GET_TELESCOPE_OPERATOR_CONNECTION  ,
    MessageType_GET_FAULT_MANAGER_CONNECTION  ,
    MessageType_BREAK_CONNECTION  ,
    MessageType_TERMINATE  ,
    MessageType_TEST  ,
    MessageType_SET_DATABASE_MANAGER  ,
    MessageType_SET_TELESCOPE_OPERATOR  ,
    MessageType_SET_FAULT_MANAGER  ,
    MessageType_SET_SOTIMEOUT  ,
    MessageType_SET_LOGLEVEL  ,
    MessageType_INITIALIZE_SYSTEM  ,
    MessageType_BEGIN_INITIALIZE_SYSTEM  ,
    MessageType_OPERATE_SYSTEM  ,
    MessageType_DIAGNOSTIC_MODE_ON  ,
    MessageType_DIAGNOSTIC_MODE_OFF  ,
    MessageType_SHUTDOWN_SYSTEM  ,
    MessageType_BEGIN_SHUTDOWN_SYSTEM  ,
    MessageType_ABOUT_TO_ABORT_SYSTEM  ,
    MessageType_BEGIN_ABOUT_TO_ABORT_SYSTEM  ,
    MessageType_STOP_SYSTEM  ,
    MessageType_GET_DATAPORT  ,
    MessageType_INITIALIZE_SYSTEM_ASYNC  ,
    MessageType_SHUTDOWN_SYSTEM_ASYNC  ,
    MessageType_ABOUT_TO_ABORT_SYSTEM_ASYNC  ,
    MessageType_MONITOR_ON  ,
    MessageType_MONITOR_OFF  ,
    MessageType_IS_MONITORING
} MROIMessageType;
char* MROIMessageTypeName[46];
long int MROIMessageTypeNumber;
char* toStringMROIMessageType(MROIMessageType x);

/*
 * Enumeration SystemType
 */
typedef enum {
    SystemType_UNKNOWN = 0 ,
    SystemType_Executive  ,
    SystemType_Supervisor  ,
    SystemType_FaultManager  ,
    SystemType_DatabaseManager  ,
    SystemType_DataCollector  ,
    SystemType_TelescopeOperator  ,
    SystemType_OperatorInterface  ,
    SystemType_UTM  ,
    SystemType_FTT  ,
    SystemType_SIC  ,
    SystemType_WAS  ,
    SystemType_UTE  ,
    SystemType_EnvironmentalMonitoringSystem  ,
    SystemType_WeatherStation
} MROISystemType;
```

```
char* MROISystemTypeName[15];
long int MROISystemTypeNumber;
char* toStringMROISystemType(MROISystemType x);

/*
 * Enumeration SystemState
 */
typedef enum {
    SystemState_UNDEFINED = 0 ,
    SystemState_STARTED  ,
    SystemState_INITIALIZING  ,
    SystemState_INITIALIZED  ,
    SystemState_OPERATIONAL  ,
    SystemState_DIAGNOSTIC  ,
    SystemState_SHUTTINGDOWN  ,
    SystemState_SHUTDOWN  ,
    SystemState_STOPPED  ,
    SystemState_ABORTING  ,
    SystemState_ABORTED
} MROISystemState;
char* MROISystemStateName[11];
long int MROISystemStateNumber;
char* toStringMROISystemState(MROISystemState x);

/*
 * Enumeration HardwareType
 */
typedef enum {
    HardwareType_UNKNOWN = 0 ,
    HardwareType_Array  ,
    HardwareType_UT  ,
    HardwareType_UTM  ,
    HardwareType_WAS  ,
    HardwareType_UTE  ,
    HardwareType_STATION  ,
    HardwareType_FTT  ,
    HardwareType_SIC  ,
    HardwareType_WeatherStation  ,
    HardwareType_AllSkyCamera
} MROIHardwareType;
char* MROIHardwareTypeName[11];
long int MROIHardwareTypeNumber;
char* toStringMROIHardwareType(MROIHardwareType x);


/*
 ************************************
 * Extended data types for Project MROI
 ************************************
 */

/**
 * Angle
 */
```

```
typedef double Angle;
Angle createAngle (double angleInRadians);
Angle readAngle(SocketInputStream* in);
void writeAngle (SocketOutputStream* out, Angle data);
char* toStringAngle (Angle this);

/**
 * AngularRate
 */
typedef double AngularRate;
AngularRate createAngularRate (double rateInRadiansPerSecond);
AngularRate readAngularRate(SocketInputStream* in);
void writeAngularRate (SocketOutputStream* out, AngularRate data);
char* toStringAngularRate (AngularRate this);

/**
 * Complex
 */
typedef double complex Complex;
Complex createComplex (double re, double im);
Complex readComplex(SocketInputStream* in);
void writeComplex (SocketOutputStream* out, Complex data);
char* toStringComplex (Complex this);

/**
 * Duration
 */
typedef long long int Duration;
Duration createDuration (long long int nanoseconds);
Duration readDuration(SocketInputStream* in);
void writeDuration (SocketOutputStream* out, Duration data);
char* toStringDuration (Duration this);

/**
 * FComplex
 */
typedef float complex FComplex;
FComplex createFComplex (float re, float im);
FComplex readFComplex(SocketInputStream* in);
void writeFComplex (SocketOutputStream* out, FComplex data);
char* toStringFComplex (FComplex this);

/**
 * Flux
 */
typedef double Flux;
Flux createFlux (double flux);
Flux readFlux(SocketInputStream* in);
void writeFlux (SocketOutputStream* out, Flux data);
char* toStringFlux (Flux this);

/**
 * Frequency
 */
```

```
typedef double Frequency;
Frequency createFrequency (double frequencyInHertz);
Frequency readFrequency(SocketInputStream* in);
void writeFrequency (SocketOutputStream* out, Frequency data);
char* toStringFrequency (Frequency this);

/**
 * Humidity
 */
typedef double Humidity;
Humidity createHumidity (double humidity);
Humidity readHumidity(SocketInputStream* in);
void writeHumidity (SocketOutputStream* out, Humidity data);
char* toStringHumidity (Humidity this);

/**
 * Length
 */
typedef double Length;
Length createLength (double lengthInMeters);
Length readLength(SocketInputStream* in);
void writeLength (SocketOutputStream* out, Length data);
char* toStringLength (Length this);

/**
 * Pressure
 */
typedef double Pressure;
Pressure createPressure (double pressure);
Pressure readPressure(SocketInputStream* in);
void writePressure (SocketOutputStream* out, Pressure data);
char* toStringPressure (Pressure this);

/**
 * Speed
 */
typedef double Speed;
Speed createSpeed (double speedInMetersPerSecond);
Speed readSpeed(SocketInputStream* in);
void writeSpeed (SocketOutputStream* out, Speed data);
char* toStringSpeed (Speed this);

/**
 * Temperature
 */
typedef double Temperature;
Temperature createTemperature (double temperatureInDegC);
Temperature readTemperature(SocketInputStream* in);
void writeTemperature (SocketOutputStream* out, Temperature data);
char* toStringTemperature (Temperature this);

/**
 * MROITime
 */
```

```
typedef long long int MROITime;
MROITime createMROITime (short int year, short int month, short int day,
    short int hr, short int min, double sec, ControlException* status);
MROITime readMROITime(SocketInputStream* in);
void writeMROITime (SocketOutputStream* out, MROITime data);
char* toStringMROITime (MROITime this);
MROITime createTimeFITS(char *t, ControlException* status);
MROITime createTimeCurrent();
MROITime createTimeRaw(long long int t);

/*
 ****************************************************************
 * Structures for Project MROI Data Model MCDB
 ****************************************************************
 */

/**
 ************************
 * FloatSample
 ************************
 *
 *  The FloatSample structure represents the measurement of a generic quantity
 *  that can be represented as a floating point number.  The units associated
 *  with the measurement are defined in the monitored property associated with
 *  the quantity.
 */
typedef struct {
    MROITime time;
    float value;
} FloatSample;
FloatSample* createFloatSample(MROITime time, float value);
void destroyFloatSample(FloatSample* this);
FloatSample* readFloatSample(SocketInputStream* in);
void writeFloatSample(SocketOutputStream* out, FloatSample* data);
char* toStringFloatSample(FloatSample* this);

/**
 ************************
 * NameValuePair
 ************************
 *
 * The NameValuePair structure represents a named value expressed as a string
 * of characters.
 */
typedef struct {
    char* name;
    char* value;
} NameValuePair;
NameValuePair* createNameValuePair(char* name, char* value);
void destroyNameValuePair(NameValuePair* this);
NameValuePair* readNameValuePair(SocketInputStream* in);
void writeNameValuePair(SocketOutputStream* out, NameValuePair* data);
char* toStringNameValuePair(NameValuePair* this);
```

```
/*
 * Other general methods
 */
typedef struct {
    int size;
    char* buffer;
    int mark;
} StringBuffer;
StringBuffer* createStringBuffer(int size);
void destroyStringBuffer(StringBuffer* this);
int appendStringBuffer(StringBuffer* this, const char* s);
char* getStringBuffer(StringBuffer* this);
int getSizeStringBuffer(StringBuffer* this);

char* createString(char*s, ControlException* status);

char* toStringBoolean(bool x);
char* toStringByte(char x);
char* toStringShort(short int x);
char* toStringInt(long int x);
char* toStringLong(long long x);
char* toStringFloat(float x);
char* toStringDouble(double x);

/**
 ****************************************
 * Class: ControlException
 ****************************************
 *
 * The ControlException is an object containing basic data about some exception
 * that has occurred within a system.   It contains the system to which the
 * exception belongs, its type and time of creation.   In addition to a message
 * explaining the exception, this object contains the name of the source code file
 * and line number of the statement that created the exception.
 *
 * This ControlException object is created as a "null" exception, indicating that
 * there has been no exception.  An exception is indicated by calling the
 * 'setException' method.  This ControlException object may be reused, in the
 * sense that its values may be cleared, using the 'clearException' method, and
 * reset, using the 'setException' method again.
 *
 * A fixed amount of space is allocated for the message (1024 bytes) and filename
 * (512 bytes).  This space is reallocated only if the size of the message or
 * filename exceeds its currently allocated space.  This space is freed when the
 * exception object is destroyed.
 *
 * For convenience in checking whether an exception has occurred, the method
 * 'isStatusOK' returns true is there is no current exception and false if an
 * exception has occurred.
 *
 */
struct _ControlException {
    // The system to which this exception belongs.
    ControlSystem* system;
```

```
    // The enumerated type of exception.
    MROIExceptionType type;
    // The time at which the exception was created.
    MROITime time;
    // The name of the source code file containing the module that created the
    // exception.
    char* filename;
    // The size of the area reserved for the filename.
    long int filenameBuffersize;
    // The line number in the source code file at which the exception was created.
    long int lineNumber;
    // A message explaining the exception.
    char* message;
    // The size of the area reserved for the message.
    long int messageBuffersize;
};

// Constructor: Create a new ControlException object that indicates there is no
// exception.
    ControlException* createControlException (void* system);

// Destructor: Destroy the specified ControlException object.
    void destroyControlException (ControlException* this);

// An exception has occurred. Set this ControlException's values to the specified
// parameters.
    void setException (ControlException* this,
            MROIExceptionType type,
            const char* message,
            const char* filename,
            long int lineNumber);

// Read this exception's values from the specified input stream.
    void readControlException (ControlException* this,
            SocketInputStream* in);

// Write this ContorlException's values to the specified output stream.
    void writeControlException (ControlException* this,
            SocketOutputStream* out);

// Convert this exception to a character string
    char* toStringControlException (ControlException* this);

// Is this current exception's status clear, i.e. is there an exception?
    bool isStatusOK (ControlException* this);

// Clear this exception's values
    void clearException (ControlException* this);

// Return this ControlException's type
    MROIExceptionType getExceptionType (ControlException* this);

// Return this ControlException's message
    const char* getExceptionMessage (ControlException* this);
```

```
// Return this ControlException's time of creation
    MROITime getExceptionTime (ControlException* this);

// Return this ControlException's filename
    const char* getExceptionFilename (ControlException* this);

// Return this ControlException's line number
    long int getExceptionLine (ControlException* this);


/**
 ****************************************
 * Class: MROISocket
 ****************************************
 *
 * There is no description.
 *
 */
struct _MROISocket {
    // The control system to which this MROISocket belongs.
    ControlSystem* system;
    // This socket's IP address.
    const char* ipAddress;
    // This socket's port number.
    long int port;
    // The size, in bytes, of the buffer to receive data.
    long int receiveBuffersize;
    // The size, in bytes, of the buffer to send data.
    long int sendBuffersize;
    // The soTimeout parameter associated with this socket.
    long int soTimeout;
    // The socketFD parameter associated with this socket.
    long int socketFD;
    // The input stream buffer.
    SocketInputStream* in;
    // The output stream buffer.
    SocketOutputStream* out;
    // The exception that is association with this socket.
    ControlException* status;
};

// There is no description.
    MROISocket* createMROISocket (ControlSystem* system,
            long int port,
            long int receiveBuffersize,
            long int sendBuffersize,
            long int soTimeout);

// There is no description.
    void destroyMROISocket (MROISocket* this);

// There is no description.
    SocketInputStream* getSocketInputStream (MROISocket* this);
```

```
// There is no description.
    SocketOutputStream* getSocketOutputStream (MROISocket* this);


/**
 ****************************************
 * Class: MROIServerSocket
 ****************************************
 *
 * There is no description.
 *
 */
struct _MROIServerSocket {
    // The control system to which this MROISocket belongs.
    ControlSystem* system;
    // This socket's IP address.
    const char* ipAddress;
    // This socket's port number.
    long int port;
    // The backlog parameter associated with this socket.
    long int backlog;
    // The soTimeout parameter associated with this socket.
    long int soServerTimeout;
    // The exception that is association with this socket.
    ControlException* status;
};

// There is no description.
    MROIServerSocket* createMROIServerSocket (ControlSystem* system,
            long int port,
            long int backlog,
            long int soServerTimeout);

// There is no description.
    void destroyMROIServerSocket (MROIServerSocket* this);

// There is no description.
    MROISocket* acceptMROIServerSocket (MROIServerSocket* this,
            long int receiveBuffersize,
            long int sendBuffersize,
            long int soTimeout);


/**
 ****************************************
 * Class: SocketInputStream
 ****************************************
 *
 * There is no description.
 *
 */
struct _SocketInputStream {
    // The socket to which this SocketInputStream belongs.
```

```
    MROISocket* socket;
    // The size of the buffer (default is 64K).
    long int buffersize;
    // The allocated buffer.
    char* buffer;
    // The current number of bytes in the buffer.
    long int size;
    // The current position of the read pointer.
    long int mark;
    // The status associated with any operation.
    ControlException* status;
};

// There is no description.
    SocketInputStream* createSocketInputStream (MROISocket* socket);

// There is no description.
    void destroySocketInputStream (SocketInputStream* this);

// There is no description.
    int receiveSocketInputStream (SocketInputStream* in);

// There is no description.
    bool readBoolean (SocketInputStream* in);

// There is no description.
    char readByte (SocketInputStream* in);

// There is no description.
    short int readShort (SocketInputStream* in);

// There is no description.
    long int readInt (SocketInputStream* in);

// There is no description.
    long long int readLong (SocketInputStream* in);

// There is no description.
    float readFloat (SocketInputStream* in);

// There is no description.
    double readDouble (SocketInputStream* in);

// There is no description.
    char* readString (SocketInputStream* in);

// There is no description.
    int readEnum (SocketInputStream* in);

// There is no description.
    int readBooleanArray (SocketInputStream* in,
            int maxNumber,
            bool* data);
```

```
// There is no description.
    int readByteArray (SocketInputStream* in,
            int maxNumber,
            char* data);

// There is no description.
    int readShortArray (SocketInputStream* in,
            int maxNumber,
            short int* data);

// There is no description.
    int readIntArray (SocketInputStream* in,
            int maxNumber,
            long int* data);

// There is no description.
    int readLongArray (SocketInputStream* in,
            int maxNumber,
            long long int* data);

// There is no description.
    int readFloatArray (SocketInputStream* in,
            int maxNumber,
            float* data);

// There is no description.
    int readDoubleArray (SocketInputStream* in,
            int maxNumber,
            double* data);

// There is no description.
    int readStringArray (SocketInputStream* in,
            int maxNumber,
            char** data);

// There is no description.
    int readEnumArray (SocketInputStream* in,
            int maxNumber,
            int* data);


/**
 ****************************************
 * Class: SocketOutputStream
 ****************************************
 *
 * There is no description.
 *
 */
struct _SocketOutputStream {
    // The socket to which this SocketOutputStream belongs.
    MROISocket* socket;
    // The size of the buffer (default is 64K).
    long int buffersize;
```

```
    // The allocated buffer.
    char* buffer;
    // The current position of the write pointer.
    long int mark;
    // The status associated with any operation.
    ControlException* status;
};

// There is no description.
    SocketOutputStream* createSocketOutputStream (MROISocket* socket);

// There is no description.
    void destroySocketOutputStream (SocketOutputStream* this);

// There is no description.
    int sendSocketOutputStream (SocketOutputStream* out);

// There is no description.
    int writeBoolean (SocketOutputStream* out,
            bool data);

// There is no description.
    int writeByte (SocketOutputStream* out,
            char data);

// There is no description.
    int writeShort (SocketOutputStream* out,
            short int data);

// There is no description.
    int writeInt (SocketOutputStream* out,
            long int data);

// There is no description.
    int writeLong (SocketOutputStream* out,
            long long int data);

// There is no description.
    int writeFloat (SocketOutputStream* out,
            float data);

// There is no description.
    int writeDouble (SocketOutputStream* out,
            double data);

// There is no description.
    int writeString (SocketOutputStream* out,
            const char* data);

// There is no description.
    int writeEnum (SocketOutputStream* out,
            int data);

// There is no description.
```

```
    int writeBooleanArray (SocketOutputStream* out,
            int number,
            bool* data);

// There is no description.
    int writeByteArray (SocketOutputStream* out,
            int number,
            char* data);

// There is no description.
    int writeShortArray (SocketOutputStream* out,
            int number,
            short int* data);

// There is no description.
    int writeIntArray (SocketOutputStream* out,
            int number,
            long int* data);

// There is no description.
    int writeLongArray (SocketOutputStream* out,
            int number,
            long long int* data);

// There is no description.
    int writeFloatArray (SocketOutputStream* out,
            int number,
            float* data);

// There is no description.
    int writeDoubleArray (SocketOutputStream* out,
            int number,
            double* data);

// There is no description.
    int writeStringArray (SocketOutputStream* out,
            int number,
            char** data);

// There is no description.
    int writeEnumArray (SocketOutputStream* out,
            int number,
            int* data);


/**
 ****************************************
 * Class: Fault
 ****************************************
 *
 * There is no description.
 *
 */
struct _Fault {
```

```
    // description
    ControlSystem* system;
    // description
    char* faultName;
    // description
    MROITime faultTime;
    // description
    MROISystemState stateAtTimeOfFault;
    // description
    char* monitoredPropertyName;
    // description
    char* message;
    // description
    long int numberData;
    // description
    char** data;
    // description
    long int numberFault;
    // description
    Fault** faultTree;
};

// There is no description.
    Fault* createFault (ControlSystem* system,
            char* faultName,
            MROISystemState stateAtTimeOfFault,
            char* monitoredPropertyName,
            char* message,
            int numberData,
            char** data,
            int numberFault,
            Fault** faultTree);

// There is no description.
    void destroyFault (Fault* this);

// There is no description.
    void writeFault (Fault* this,
            SocketOutputStream* out);

// There is no description.
    char* toStringFault (Fault* this);


/**
 ****************************************
 * Class: Alert
 ****************************************
 *
 * There is no description.
 *
 */
struct _Alert {
    // description
```

```
    ControlSystem* system;
    // description
    MROISystemType systemType;
    // description
    char* systemName;
    // description
    MROISystemState systemState;
    // description
    char* alertName;
    // description
    MROIAlertLevel alertLevel;
    // description
    MROITime alertTime;
    // description
    char* monitoredPropertyName;
    // description
    char* message;
    // description
    Fault* fault;
};

// There is no description.
    Alert* createAlert (ControlSystem* system,
            char* alertName,
            MROIAlertLevel alertLevel,
            char* monitoredPropertyName,
            char* message,
            Fault fault);

// There is no description.
    void destroyAlert (Alert* this);

// There is no description.
    void writeAlert (Alert* this,
            SocketOutputStream* out);

// There is no description.
    char* toStringAlert (Alert* this);


/**
 ****************************************
 * Class: Identification
 ****************************************
 *
 * There is no description.
 *
 */
struct _Identification {
    // description
    ControlSystem* system;
    // TODO
    int toDo;
};
```

```
// There is no description.
    Identification* createIdentification ();

// There is no description.
    void destroyIdentification (Identification* this);

// There is no description.
    void readIdentification (SocketInputStream* in,
            Identification* data);

// There is no description.
    void writeIdentification (SocketOutputStream* out,
            Identification* data);

// There is no description.
    char* toStringIdentification (Identification* this);


/**
 ****************************************
 * Class: OperatorMessage
 ****************************************
 *
 * There is no description.
 *
 */
struct _OperatorMessage {
    // description
    ControlSystem* system;
    // description
    Identification* systemId;
    // description
    MROITime time;
    // description
    char* message;
};

// There is no description.
    OperatorMessage* createOperatorMessage (ControlSystem* system,
            char* message);

// There is no description.
    void destroyOperatorMessage (OperatorMessage* this);

// There is no description.
    void writeOperatorMessage (OperatorMessage* this,
            SocketOutputStream* out);

// There is no description.
    char* toStringOperatorMessage (OperatorMessage* this);


/**
```

```
 ****************************************
 * Class: RemoteConnection
 ****************************************
 *
 * There is no description.
 *
 */
struct _RemoteConnection {
    // description
    ControlSystem* system;
    // TODO
    int toDo;
};

// There is no description.
    RemoteConnection* createRemoteConnection ();

// There is no description.
    void destroyRemoteConnection (RemoteConnection* this);

// There is no description.
    void readRemoteConnection (SocketInputStream* in,
            RemoteConnection* data);

// There is no description.
    void writeRemoteConnection (SocketOutputStream* out,
            RemoteConnection* data);

// There is no description.
    char* toStringRemoteConnection (RemoteConnection* this);


/**
 ****************************************
 * Class: Client
 ****************************************
 *
 * There is no description.
 *
 */
struct _Client {
    // description
    ControlSystem* system;
    // TODO
    int toDo;
};

// There is no description.
    Client* createClient ();

// There is no description.
    void destroyClient (Client* this);

// There is no description.
```

```
    void readClient (SocketInputStream* in,
            Client* data);

// There is no description.
    void writeClient (SocketOutputStream* out,
            Client* data);

// There is no description.
    char* toStringClient (Client* this);


/**
 ****************************************
 * Class: ControlLogger
 ****************************************
 *
 * The ControlLogger is a class that creates an object that manages a log file.
 * The model is Java's logger. See java.util.logging.Logger.
 *
 * The public methods are: 'logSevere', 'logWarning', 'logInfo', 'logConfig',
 * 'logFine', 'logFiner', 'logFinest', 'logException'.
 *
 * The logger is created to write each log entry directly to the file and to
 * assume that the application is not using threads.  These two assumptions may be
 * changed by setting a buffer size (method 'setBufferSize') and by setting the
 * threads option (method 'setThreads').
 *
 */
struct _ControlLogger {
    // The control system to which this logger belongs.
    ControlSystem* system;
    // The log file name associated with this system.
    char* logFilename;
    // The file handle of the log file.
    long int logFile;
    // The current status of the logger.
    ControlException* status;
    // If 'isThreads' is true, each write is protected by a lock.
    bool isThreads;
    // The size of the buffer that holds the log records.
    long int buffersize;
    // The allocated buffer, if any, to hold the log records.
    char * buffer;
};

// Constructor: The createControlLogger method creates a ControlLogger object, as
// well as creating the log file and opening it.    It also creates a
// ControlException object; this status object is always set after any operation.
//
// The ControlLogger object is created to write each log entry directly to the
// file and to assume that the application is not using threads.  If these options
// are to be changed, then the 'setBuffersize' or 'setThreads' methods must be
// called prior to writing any log records.
    ControlLogger* createControlLogger (char* filename,
```

```
            ControlSystem* system);

// Destructor: Free all memory allocated to this ControlLogger and set 'this' to
// NULL.
    void destroyControlLogger (ControlLogger* this);

// Allocate a buffer for the log records.    If the buffer is allocated, writes to
// the log are written to the buffer.    The buffer is written to the file only
// when there is an overflow condition.  If the buffersize parameter is 0, each
// write to the log is written directly to the file.
//
// The 'setBuffersize' method may be called at any time during the execution of
// the application.  If there is an existing buffer, its contents are written to
// the log file and the old buffer is freed before allocating a new buffer.
    void setBuffersize (ControlLogger* this,
            long int buffersize);

// Set the 'threads' option.  This options indicates that there are threads in the
// application.  The result is that there is a lock that must be acquired in
// performing a write to the log.
    void setThreads (ControlLogger* this);

// The '_writeToLog' method is an internal method designed to write log records to
// the log file.  It should not be used by an application.    It is used by all of
// the public methods that are used to write particular types of log records.  If
// the ControlLogger's status indicates an error, this method sets the
// ControlSystem's status and returns; nothing is written to the file.
    void _writeToLog (ControlLogger* this,
            MROILogLevel logLevel,
            MROILogType logType,
            char* message);


/**
 ****************************************
 * Class: ControlSystem
 ****************************************
 *
 * There is no description.
 *
 */
struct _ControlSystem {
    // The name of this type of system
    MROISystemType systemType;
    // The name associated with this package
    const char* packageName;
    // The name of that identifies this instance of the system
    const char* systemName;
    // The logger associated with this system
    ControlLogger* logger;
    // The address of the host that this system runs on
    const char* hostAddress;
    // The main port on this system's host on which the system listens for connections
    long int mainPort;
```

```
// The backlog on the ports associated with this system
long int backlog;
// The default timeout, in milliseconds, for the accept() function.  This is used
// after the initial accept and may be reset by the Executive
long int soTimeout;
// The data port on this system's host on which the system listens for connections
// to the data port
long int dataPort;
// The server socket on the main port
MROIServerSocket* serverSocket;
// The server socket on the data port
MROIServerSocket* serverDataSocket;
// The name of the Telescope Operator system
char* telescopeOperatorName;
// The IP address of the location of the Telescope Operator system
char* telescopeOperatorIPAddress;
// The port used to connect to the Telescope Operator system
long int telescopeOperatorPort;
// The name of the Database Manager system
char* databaseManagerName;
// The IP address of the location of the Database Manager system
char* databaseManagerIPAddress;
// The port used to connect to the Database Manager system
long int databaseManagerPort;
// The name of the Fault Manager system
char* faultManagerName;
// The IP address of the location of the Fault Manager system
char* faultManagerIPAddress;
// The port used to connect to the Fault Manager system
long int faultManagerPort;
// The number of monitor points associated with this system
long int numberMonitorPoint;
// The names of the monitor points associated with this system
const char** monitorPoint;
// The number of commands associated with this system
long int numberCommand;
// The names of commands associated with this system
const char** command;
// The current state of the system
MROISystemState systemState;
// Is the system currently producing monitor data?
bool monitoring;
// The exception status associated with this control system
ControlException* status;
// Whether this C system is implemented without using threads
bool noThreads;
// A pointer to the system's initialization actions that uses threads
void (*initializeAction)(void*) ;
// A pointer to the system's initialization actions that does not use threads
void (*initializeActionNoThread)(void*, void(*)(void*)) ;
// A pointer to the system's shutdown actions that uses threads
void (*shutdownAction)(void*) ;
// A pointer to the system's shutdown actions that does not use threads
void (*shutdownActionNoThread)(void*, void(*)(void*)) ;
```

```
    // A pointer to the system's about-to-abort actions that uses threads
    void (*aboutToAbortAction)(void*) ;
    // A pointer to the system's about-to-abort actions that does not use threads
    void (*aboutToAbortActionNoThread)(void*, void(*)(void*)) ;
};

// The globalError method is called if an unrecoverable error occurs that cannot
// be reported  using the exception mechanism, such as encountering a null
// pointer.  This method terminates  the entire program abruptly.
    void _globalError (const char* message,
            const char* filename,
            long int lineNumber);

// Constructor: Create a control system.
    ControlSystem* createControlSystem (MROISystemType systemType,
            const char* packageName,
            const char* systemName,
            const char* hostAddress,
            long int mainPort,
            long int dataPort,
            long int backlog);

// The _initControlSystem is an internal method used to initialize a ControlSystem
// structure.  It is used by C systems that are extensions of the basic
// ControlSystem.
    void _initControlSystem (void* system,
            MROISystemType systemType,
            const char* packageName,
            const char* systemName,
            const char* hostAddress,
            long int mainPort,
            long int dataPort,
            long int backlog);

// There is no description.
    void destroyControlSystem (void* system);

// There is no description.
    bool isSystemException (void* system);

// An exception has occurred.  Set this ControlSystem's status to indicate an
// exception with the sepcified values.
    void setSystemException (void* system,
            MROIExceptionType type,
            const char* message,
            const char* filename,
            long int lineNumber);

// There is no description.
    void clearSystemException (void* system);

// There is no description.
    char* getSystemExceptionMessage (void* system);
```

```
// There is no description.
    ControlException* getSystemException (void* system);

// There is no description.
    void sendMROIFault (void* system,
            char* faultName,
            char* monitoredPropertyName,
            char* message,
            int numberData,
            char** data,
            int numberFault,
            Fault* faultTree);

// There is no description.
    void sendMROIAlert (void* system,
            char* alertName,
            MROIAlertLevel alertLevel,
            char* monitoredPropertyName,
            char* message,
            Fault fault);

// There is no description.
    void sendMROIOperatorMessage (void* system,
            char* message);

// There is no description.
    char* getLogFilename (void* system,
            char* message);

// There is no description.
    void setLoggerBuffersize (void* system,
            long int buffersize);

// There is no description.
    void setLoggerThreadsOption (void* system,
            bool option);

// Write a log record indicating a severe problem, a condition that usually stops
// execution.
    void logSevere (void* system,
            MROILogType logType,
            char* message);

// Write a log record indicating an adverse condition that does not terminate
// execution.
    void logWarning (void* system,
            MROILogType logType,
            char* message);

// Write a log record indicating any significant information relevant to the
// execution of the system.
    void logInfo (void* system,
            MROILogType logType,
            char* message);
```

```
// Write a log record indicating information relevant to the current configuration
// of the system, usually associated with startup conditions.
    void logConfig (void* system,
            MROILogType logType,
            char* message);

// Write a log record indicating debugging information at a high level of detail.
    void logFine (void* system,
            MROILogType logType,
            char* message);

// Write a log record indicating debugging information at a medium level of detail.
    void logFiner (void* system,
            MROILogType logType,
            char* message);

// Write a log record indicating debugging information at a low level of detail.
    void logFinest (void* system,
            MROILogType logType,
            char* message);

// Write a ControlException object to the log.  This is written as a ‘severe’ log
// entry.
    void logCurrentException (void* system,
            ControlException* exception);

// There is no description.
    ControlLogger* getLogger (void* system);

// There is no description.
    void connectToDatabaseManager (void* system);

// There is no description.
    void disconnectFromDatabaseManager (void* system);

// There is no description.
    MROISystemType getSystemType (void* system);

// There is no description.
    char* getPackageName (void* system);

// There is no description.
    char* getSystemName (void* system);

// There is no description.
    char* getHostAddress (void* system);

// There is no description.
    long int getMainPort (void* system);

// There is no description.
    long int getDataPort (void* system);
```

```
// There is no description.
    long int getBacklog (void* system);

// There is no description.
    long int getSOTimeout (void* system);

// There is no description.
    MROISystemState getSystemState (void* system);

// There is no description.
    RemoteConnection* getDatabaseManagerConnection (void* system);

// There is no description.
    RemoteConnection* getTelescopeOperatorConnection (void* system);

// There is no description.
    RemoteConnection* getFaultManagerConnection (void* system);

// There is no description.
    void setDatabaseManager (void* system,
            char* systemName,
            char* ipAddress,
            long int port);

// There is no description.
    void setFaultManager (void* system,
            char* systemName,
            char* ipAddress,
            long int port);

// There is no description.
    void setTelescopeOperator (void* system,
            char* systemName,
            char* ipAddress,
            long int port);

// There is no description.
    void setSOTimeout (void* system,
            long int timeout);

// There is no description.
    void setLogLevel (void* system,
            MROILogLevel level);

// Start the system.  The system must be in the UNDEFINED state.
    MROISystemState startSystem (void* system);

// This method is only used for C systems implemented with no threads.  It is the
// callback method in the asynchronous method to the initializeAction method of
// the system.
    void initializeSystemReturn (void* system);

// Initialize the system.  The system must be in the STARTED or STOPPED state.
    MROISystemState initializeSystem (void* system);
```

```
// Begin the system initialization process but return immediately.  The system
// must be in the STARTED or STOPPED state.
    MROISystemState beginInitializeSystem (void* system);

// Place the system in the operational state.  The system must be in the
// INITIALIZED state.
    MROISystemState operateSystem (void* system);

// Place the system in the diagnostic mode.  The system must be in the OPERATIONAL
// state.
    MROISystemState diagnosticModeOn (void* system);

// Place the system in the operational mode.  The system must be in the DIAGNOSTIC
// state.
    MROISystemState diagnosticModeOff (void* system);

// This method is only used for C systems implemented with no threads.  It is the
// callback method in the asynchronous method to the shutdownAction method of the
// system.
    void shutdownSystemReturn (void* system);

// Shut down the system.  The system must be in the OPERATIONAL state.
    MROISystemState shutdownSystem (void* system);

// Begin the system shutdown process but return immediately.  The system must be
// in the OPERATIONAL state.
    MROISystemState beginShutdownSystem (void* system);

// This method is only used for C systems implemented with no threads.  It is the
// callback method in the asynchronous method to the aboutToAbortAction method of
// the system.  The system is placed in the ABORTED state.
    void aboutToAbortSystemReturn (void* system);

// The system is about to be aborted; save any crucial data now.  The system may
// be in any state.
    MROISystemState aboutToAbortSystem (void* system);

// The system is about to be aborted.  Begin to save any crucial data now; but
// return immediately.  The system may be in any state.
    MROISystemState beginAboutToAbortSystem (void* system);

// Place the system in the stopped state.  The system must be in the SHUTDOWN
// state.
    MROISystemState stopSystem (void* system);

// Initialize the system asynchronously.  The system must be in the STARTED or
// STOPPED state.
    MROISystemState initializeSystemAsync (void* system);

// Shut down the system asynchronously.  The system must be in the OPERATIONAL
// state.
    MROISystemState shutdownSystemAsync (void* system);
```

```
// The system is about to be aborted; save any crucial data asynchronously.  The
// system may be in any state.
    MROISystemState aboutToAbortSystemAsync (void* system);

// Turn on data monitoring.
    void monitorOn (void* system);

// Turn off data monitoring.
    void monitorOff (void* system);

// Turn on data monitoring.
    bool isMonitoring (void* system);

// There is no description
    void breakConnection (void* system);

// There is no description
    void terminate (void* system);

// There is no description
    void test (void* system);

// Set the names of the monitor points for this system.
    void setControlMonitorPoints (void* system,
            long int numberMonitorPoint,
            const char** monitorPointName);

// Set the names of the commmands for this system.
    void setControlCommands (void* system,
            long int numberCommand,
            const char** commandName);




#endif
```

24.2 EMSS spreadsheet

The spreadsheets defining the interface to the example of the Weather Station within the EMSS system are shown in Figures 3 through 7.

System Interface Definition

Name	Description	Package	Import
EnvironmentalMonitoringSystem	The environmental monitoring system reports on weather, dust, seeing and also includes an all-sky camera.	none	none
WeatherStation	The weather station report temperature, wind speed and direction and other items.	none	none

Full Name	Extends	Parent System	Implement
Environmental Monitoring System	none	none	no
Weather Station	none	EnvironmentalMonitoringSystem	C

Is Asynchronous	Is A Monitor	Work Package	Document Title	Document Number	Document Issue	Document Date
no	no	4.15.00	Environmental Monitoring System Requirements	415-INT-REQ-0010	1	12/1/2008
yes	yes	4.15.00	Environmental Monitoring System Requirements	415-INT-REQ-0010	1	12/1/2008

Figure 3: **EMSS System spreadsheet**

Monitor Points

Name	System	Description	Returns
Temperature	WeatherStation	current temperature in °C	Temperature
WindSpeed	WeatherStation	current wind speed in m/sec	Speed
WindDirection	WeatherStation	current wind direction in radians	Angle

Can Be Null	Throws Exception	Asynchronous	Data Unit	Minimum Value	Maximum Value	Default Value
no	yes	no	°C	-30	60	30
no	yes	no	m/sec	0	100	4
no	yes	no	rad	0	6.28	0.79

System Unit	Raw Data Type	Scale	Offset	Mode	Implement	Archive Interval (secs)
°C	Temperature	none	none	any	yes	5
m/sec	Speed	none	none	any	yes	5
rad	Angle	none	none	any	yes	5

Archive Only On Change	Display Unit	Graph Minimum	Graph Maximum	Graph Title
no	°C	-10	40	Ambiant Air Temperature (°C)
no	m/sec	0	20	Wind Speed (m/sec)
no	°	0	360	Wind Direction (°)

Figure 4: **EMSS Monitor spreadsheet**

Fault Definitions

Fault Name	System	Monitor Point	Description
TooCold	WeatherStation	Temperature	it's too cold
TooHot	WeatherStation	Temperature	it's too hot
HighWind	WeatherStation	WindSpeed	too much wind
Wind	WeatherStation	WindSpeed	high wind warning

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 5: **EMSS Fault spreadsheet**

Control Commands

Name	System	Description	Returns
getAverageWindSpeed	WeatherStation	get the average wind speed over the specified interval of time	Speed

Can Be Null	Throws Exception	Asynchronous	Mode	Implement
no	yes	yes	any	yes

Figure 6: **EMSS Control spreadsheet**

Parameters

Parameter Name	System	Command	Description
minutes	WeatherStation	getAverageWindSpeed	an interval of time in minutes
weatherFilename	WeatherStation	WeatherStation	the name of the internal file in which the weather data are stored
weatherFile	WeatherStation	WeatherStation	the file handle associated with the file

Required	Data Type	Data Unit	Minimum Value	Maximum Value
yes	Duration	minutes	1	60
yes	string	none	none	none
yes	long int	none	none	none

Default Value	System Unit	Raw Data Type	Scale	Offset
5	min	float	none	none
none	none	none	none	none
0	none	none	none	none

Figure 7: **EMSS Parameters spreadsheet**

24.3 Generated code for file: WeatherStation.h

```
#ifndef MROI_WeatherStation_H
#define MROI_WeatherStation_H

#include <MROIControlSystem.h>

typedef struct _WeatherStation WeatherStation;

struct _WeatherStation {
    // This prefix is exactly the same as ControlSystem, which
    // enables us to use 'WeatherStation*' as 'ControlSystem*'.

    // The name of this type of system
    MROISystemType systemType;
    // The name associated with this package
    const char* packageName;
    // The name of that identifies this instance of the system
    const char* systemName;
    // The logger associated with this system
    ControlLogger* logger;
    // The address of the host that this system runs on
    const char* hostAddress;
    // The main port on this system's host on which the system listens for connections
    long int mainPort;
    // The backlog on the ports associated with this system
    long int backlog;
    // The default timeout, in milliseconds, for the accept() function.  This is used
    // after the initial accept and may be reset by the Executive
    long int soTimeout;
    // The data port on this system's host on which the system listens for connections
    // to the data port
    long int dataPort;
    // The server socket on the main port
    MROIServerSocket* serverSocket;
    // The server socket on the data port
    MROIServerSocket* serverDataSocket;
    // The name of the Telescope Operator system
    char* telescopeOperatorName;
    // The IP address of the location of the Telescope Operator system
    char* telescopeOperatorIPAddress;
    // The port used to connect to the Telescope Operator system
    long int telescopeOperatorPort;
    // The name of the Database Manager system
    char* databaseManagerName;
    // The IP address of the location of the Database Manager system
    char* databaseManagerIPAddress;
    // The port used to connect to the Database Manager system
    long int databaseManagerPort;
    // The name of the Fault Manager system
    char* faultManagerName;
    // The IP address of the location of the Fault Manager system
    char* faultManagerIPAddress;
    // The port used to connect to the Fault Manager system
```

```
    long int faultManagerPort;
    // The number of monitor points associated with this system
    long int numberMonitorPoint;
    // The names of the monitor points associated with this system
    const char** monitorPoint;
    // The number of commands associated with this system
    long int numberCommand;
    // The names of commands associated with this system
    const char** command;
    // The current state of the system
    MROISystemState systemState;
    // Is the system currently producing monitor data?
    bool monitoring;
    // The exception status associated with this control system
    ControlException* status;
    bool noThreads;
    // A pointer to the system's initialization actions
    void (*initializeAction)(void*) ;
    void (*initializeActionNoThread)(void*, void(*)(void*));
    // A pointer to the system's shutdown actions
    void (*shutdownAction)(void*) ;
    void (*shutdownActionNoThread)(void*, void(*)(void*));
    // A pointer to the system's about-to-abort actions
    void (*aboutToAbortAction)(void*) ;
    void (*aboutToAbortActionNoThread)(void*, void(*)(void*)) ;

    // The following data items are unique to a WeatherStation system.

    // archiving interval in seconds for current temperature in C
    Duration temperatureInterval;
    // archiving interval in seconds for current wind speed in m/sec
    Duration windSpeedInterval;
    // archiving interval in seconds for current wind direction in radians
    Duration windDirectionInterval;
    // the name of the internal file in which the weather data are stored
    char* weatherFilename;
    // the file handle associated with the file
    long int weatherFile;
};

// Constructor and destructor
    WeatherStation* createStandaloneWeatherStation(char* systemName);
    WeatherStation* createWeatherStation(char* systemName, char* hostAddress,
        long int mainPort, long int dataPort, long int backlog);
    void destroyWeatherStation(WeatherStation* this);

// Actions associated with state changes
    void initializeWeatherStationAction(WeatherStation* this);
    void shutdownWeatherStationAction(WeatherStation* this);
    void aboutToAbortWeatherStationAction(WeatherStation* this);

// Monitor point: Temperature
    Temperature getTemperature(WeatherStation* this, ControlException* err);
// Monitor point: WindSpeed
```

```
    Speed getWindSpeed(WeatherStation* this, ControlException* err);
// Monitor point: WindDirection
    Angle getWindDirection(WeatherStation* this, ControlException* err);
// Monitor point: TemperatureInterval
    Duration getTemperatureInterval(WeatherStation* this);
// Monitor point: WindSpeedInterval
    Duration getWindSpeedInterval(WeatherStation* this);
// Monitor point: WindDirectionInterval
    Duration getWindDirectionInterval(WeatherStation* this);

// Control Command: setTemperatureInterval
    void setTemperatureInterval(WeatherStation* this, Duration temperatureInterval);
// Control Command: setWindSpeedInterval
    void setWindSpeedInterval(WeatherStation* this, Duration windSpeedInterval);
// Control Command: setWindDirectionInterval
    void setWindDirectionInterval(WeatherStation* this, Duration windDirectionInterval);
// Control Command: getAverageWindSpeed
    void getAverageWindSpeed(WeatherStation* this, Duration minutes,
        ControlException* err, void (*callback)(WeatherStation*, Speed, ControlException*));

#endif
```

24.4 Generated code for file: WeatherStationInterface.c

```
# include <WeatherStation.h>
# include <stdlib.h>
# include <string.h>

typedef void (*StateActionPointer)(void*);

WeatherStation* createStandaloneWeatherStation(char* systemName) {
    WeatherStation* this = malloc(sizeof(WeatherStation));
    if (this == NULL)
        _globalError("Cannot allocate memory", __FILE__, __LINE__);
    _initControlSystem(this, SystemType_WeatherStation, "", systemName, NULL, 0, 0, 0);
    if (this->status == NULL) {
        free(this);
        _globalError("Cannot allocate memory", __FILE__, __LINE__);
    }

    this->noThreads = false;
    this->initializeAction = (StateActionPointer)initializeWeatherStationAction;
    this->shutdownAction = (StateActionPointer)shutdownWeatherStationAction;
    this->aboutToAbortAction = (StateActionPointer)aboutToAbortWeatherStationAction;
    this->initializeActionNoThread = NULL;
    this->shutdownActionNoThread = NULL;
    this->aboutToAbortActionNoThread = NULL;

    this->temperatureInterval = 5000000000LL;
    this->windSpeedInterval = 5000000000LL;
    this->windDirectionInterval = 5000000000LL;
    this->weatherFilename = "";
    this->weatherFile = 0;

    return this;

}

WeatherStation* createWeatherStation(char* systemName, char* hostAddress,
        long int mainPort, long int dataPort, long int backlog) {
    WeatherStation* this = malloc(sizeof(WeatherStation));
    if (this == NULL)
        _globalError("Cannot allocate memory", __FILE__, __LINE__);
    _initControlSystem((ControlSystem*)this, SystemType_WeatherStation, "",
        systemName, hostAddress, mainPort, dataPort, backlog);
    if (this->status == NULL) {
        free(this);
        _globalError("Cannot allocate memory", __FILE__, __LINE__);
    }

    this->noThreads = false;
    this->initializeAction = (StateActionPointer)initializeWeatherStationAction;
    this->shutdownAction = (StateActionPointer)shutdownWeatherStationAction;
    this->aboutToAbortAction = (StateActionPointer)aboutToAbortWeatherStationAction;
    this->initializeActionNoThread = NULL;
    this->shutdownActionNoThread = NULL;
```

```
    this->aboutToAbortActionNoThread = NULL;

    this->temperatureInterval = 5000000000LL;
    this->windSpeedInterval = 5000000000LL;
    this->windDirectionInterval = 5000000000LL;
    this->weatherFilename = "";
    this->weatherFile = 0;

    return this;

}

void destroyWeatherStation(WeatherStation* this) {
    // If there is anything we allocated in the constructor, we should free it now.
    destroyControlSystem(this);
}

// Monitor point: TemperatureInterval
Duration getTemperatureInterval(WeatherStation* this) {
    return this->temperatureInterval;
}
// Monitor point: WindSpeedInterval
Duration getWindSpeedInterval(WeatherStation* this) {
    return this->windSpeedInterval;
}
// Monitor point: WindDirectionInterval
Duration getWindDirectionInterval(WeatherStation* this) {
    return this->windDirectionInterval;
}

// Control Command: setTemperatureInterval
void setTemperatureInterval(WeatherStation* this, Duration temperatureInterval) {
    this->temperatureInterval = temperatureInterval;
}
// Control Command: setWindSpeedInterval
void setWindSpeedInterval(WeatherStation* this, Duration windSpeedInterval) {
    this->windSpeedInterval = windSpeedInterval;
}

// Control Command: setWindDirectionInterval
void setWindDirectionInterval(WeatherStation* this, Duration windDirectionInterval) {
    this->windDirectionInterval = windDirectionInterval;
}
```

24.5 Implementation file: WeatherStation.c

```
# include <WeatherStation.h>
# include <stdlib.h>
# include <string.h>
# include <stdbool.h>
# include <errno.h>
# include <math.h>
# include <stdio.h>

void initializeWeatherStationAction(WeatherStation* this) {
    printf("%s\n", "Executing initializeWeatherStationAction");
    // Actions to initialize the WeatherStation go here.
}

void shutdownWeatherStationAction(WeatherStation* this) {
    printf("%s\n", "Executing shutdownWeatherStationAction");
    // Actions to shutdown the WeatherStation go here.
}

void aboutToAbortWeatherStationAction(WeatherStation* this) {
    printf("%s\n", "Executing aboutToAbortWeatherStationAction");
    // Actions to save crucial data go here.
}

// Monitor point: Temperature tbd
Temperature getTemperature(WeatherStation* this, ControlException* err) {
    return 30.0;
}

// Monitor point: WindSpeed tbd
Speed getWindSpeed(WeatherStation* this, ControlException* err) {
    return 4.0;
}

// Monitor point: WindDirection tbd
Angle getWindDirection(WeatherStation* this, ControlException* err) {
    return 0.785;
}

// Control Command: getAverageWindSpeed tbd
void getAverageWindSpeed(WeatherStation* this, Duration minutes,
        ControlException* err, void (*callback)(WeatherStation*, Speed, ControlException*)) {
    callback(this, 0.0, err);
}
```

24.6 Test program: file CTestWeatherStation.c

```
# include <stdio.h>
# include <complex.h>
# include <stdlib.h>
# include <string.h>
# include <stdbool.h>

# include <WeatherStation.h>

void reportState(WeatherStation* this) {
    printf("%s %s\n","WeatherStation state:",
        toStringMROISystemState(getSystemState(this)));
}

int main (int narg, char** arg) {
    WeatherStation* weather1 = createStandaloneWeatherStation ("test");
    printf("%s\n", "WeatherStation created in standalone mode.");
    reportState(weather1);

    startSystem(weather1);
    if (isSystemException(weather1)) {
        printf("%s %s\n", "Could not start system.",
            getSystemExceptionMessage(weather1));
        return 1;
    }
    reportState(weather1);

    initializeSystem(weather1);
    if (isSystemException(weather1)) {
        printf("%s %s\n", "System initialization failed.",
            getSystemExceptionMessage(weather1));
        return 2;
    }
    reportState(weather1);

    operateSystem(weather1);
    if (isSystemException(weather1)) {
        printf("%s %s\n", "Could not make system operational.",
            getSystemExceptionMessage(weather1));
        return 4;
    }
    reportState(weather1);

    ControlException* err = createControlException(weather1);

    Temperature t = getTemperature(weather1, err);
    if (isStatusOK(err))
        printf("%s %6.2f\n", "The temperature is", t);
    else {
        printf("%s %s\n", "Error getting temperature.",
            getExceptionMessage(err));
        return 5;
    }
```

```
    Speed s = getWindSpeed(weather1, err);
    if (isStatusOK(err))
        printf("%s %6.2f\n", "The wind speed is", s);
    else {
        printf("%s %s\n", "Error getting wind speed.",
            getExceptionMessage(err));
        return 6;
    }

    Angle a = getWindDirection(weather1, err);
    if (isStatusOK(err))
        printf("%s %6.2f\n", "The wind direction is", a);
    else {
        printf("%s %s\n", "Error getting wind direction.",
            getExceptionMessage(err));
        return 7;
    }

    shutdownSystem(weather1);
    if (isSystemException(weather1)) {
        printf("%s %s\n", "Error shutting system down.",
            getSystemExceptionMessage(weather1));
        return 8;
    }
    reportState(weather1);

    stopSystem(weather1);
    if (isSystemException(weather1)) {
        printf("%s %s\n", "Could not stop system.",
            getSystemExceptionMessage(weather1));
        return 10;
    }
    reportState(weather1);

    destroyWeatherStation(weather1);
    if (isSystemException(weather1)) {
        printf("%s %s\n", "Error destroying system.",
            getSystemExceptionMessage(weather1));
        return 11;
    }
    printf("%s\n","WeatherStation destroyed.");

    return 0;
}
```