U.S. DEPARTMENT OF ENERGY

PNNL-22600

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The MKS-910 Server

DES-0035

Revision 1

Charlie Hubbard

July 2012

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The MKS-910 Server

Charlie Hubbard

DES-0035
Revision 1
July 2012

Contents

1 Introduction

This document describes the *MKS-910 server* application. This server amounts to a user-mode device driver capable of managing one or more MKS model 910 *DualTrans*^{TM} pressure sensors. It is implemented by the *mks910Server.cpp* source module. The server supports the client interface defined and implemented by the files *mks910ClientLib.h* and *mks910ClientLib.cpp* respectively. A standard text-mode test menu client for the server is implemented by *mks910Menu.cpp*.

The primary documentation for the MKS-910 server source code is the Doxygen-generated HTML documentation associated with each of the above source files. That documentation set is automatically built based on the source code itself. It provides the most detailed, most up-to-date descriptions of the code. The document you are reading now is supplemental, and is intended to provide deeper background for the server and its client API. If contradictions between this document and the Doxygen-generated documentation are found, the Doxygen-generated documentation should be considered correct.

Like most of our hardware interfacing servers, the MKS-910 server is not necessarily intended to provide an exhaustive interface to all of the hardware's capabilities. Instead, only that subset of functionality that we need for our current projects is supported. However, the MKS-910 server is unique in that it actually implements functionality the underlying hardware *does not* have. Specifically, provisions exist in the server to compute a binary gas concentration based on simultaneous readings from the sensor's internal piezo and pirani pressure transducers. The details for this computation are handled by a PNNL-developed library which is fully described by design document ***DES-0059*, *The MKS-910 Concentration Library***. Please refer to that document for details.

2 MKS-910 Hardware

The MKS model 910 *DualTrans*^{TM} pressure sensor is an extended-range pressure sensor with a usable pressure range between $1.0{*}10^{-5}$ and 1500 Torr. This wide pressure range is achieved by incorporating two independent pressure transducers into a single sensor package: a *pirani* style transducer[1] that provides good pressure values at low pressures (below 10 Torr for the MKS-910), and a piezoelectric pressure transducer that provides good pressure values above 10 Torr. Electronics integrated into the MKS-910 package monitor both transducers and select a reading from one or the other (depending on the current pressure) to be reported back as the pressure for the sensor.

[1]Pirani pressure sensors are thermal conductivity devices that function by measuring the rate at which energy from a heating element can be dissipated to the surrounding gas molecules. They function optimally at low pressures with a typical pressure range between $1.0 * 10^{-4}$ and 1.0 Torr (somewhat extended on the MKS-910). Because the rate of energy transfer depends on both the pressure of the gas surrounding the heating element and the thermal conductivity of the gas, pirani gauges must be calibrated for the specific gas mixture they're measuring.

Normally, the sensor reports pressure as an analog output voltage driven by its integrated electronics. The output has a range of 1.0 to 9.2 V, increasing at the rate of 1 V volt per decade. If the sensor was being used strictly as an extended-range pressure sensor, this would be sufficient. However, for our application, the sensor is actually used as an inexpensive, yet accurate, binary gas concentration analyzer. To perform the binary gas calculation, it is necessary to know the instantaneous readings of the piezo and pirani transducers simultaneously. Fortunately, the MKS-910 also offers an RS-232 interface through which the value of each transducer (and an embedded low-precision temperature sensor) can be read at any time. This server communicates with all attached 910 sensors using their RS-232 interfaces and makes no provisions for using the analog output alternative.

3 Serial Communications

As mentioned in the previous section, this server communicates with the MKS-910 sensors it manages via RS-232 serial links using an MKS proprietary, ASCII text-based communication protocol. Full details of the protocol can be found in the MKS-910 users manual (see document *MKS-910-manual.pdf* in the *manuals* subirectory), but highlights of the protocol are listed below:

- The server communicates with the MKS-910 pressure sensor over an RS-232 link, but RS-485 links are also supported by the hardware. RS-485 permits multiple sensors to be connected to the same serial port, and, for that reason, individual sensors must be assigned a sensor-specific address. Even though the server doesn't use RS-485, all messages to the sensor(s), must still contain the address field. The factory default address is "253." The server uses address 253 for all messages to all sensors.
- All messages must begin with an "@" character followed by the three-digit device address. The "@" character is a synchronization character that MKS-910s can use to recognize the beginning of a new message.
- All messages are terminated with a three-character sequence consisting of a semicolon followed by two capital "F"s (";FF").
- Messages sent to the MKS-910 from the server are either *queries* or *commands*. Short (typically two- or three-character) message identifiers are used to identify each specific message. For example, the string "PR1" is used to query the value of the sensor's pirani transducer.
- Command messages are used to set the values of various MKS-910 parameters. Command messages are distinguished by the presence of an exclamation mark ("!") in the message string. As an example, the following command message uses the "GT" command (gas type) to tell the sensor to assume nitrogen is the gas being measured when interpreting the readings from the pirani transducer.

  ```
  @253GT!NITROGEN;FF
  ```

- Query messages are used to request the value of parameters and transducer data from the sensor. Query messages are distinguished by the presence of a question mark ("?") in the message string. As an example, the following query message uses the "SN" query (serial number) to request the sensor's serial number.

  ```
  @253SN?;FF
  ```

- Response messages sent from the MKS-910 to the server contain the string "ACK" to indicate success (followed by result data if the response is to a query message), or the string "NAK" to indicate an error occurred. In both cases, these keywords appear immediately following the address of the device generating the response.

  ```
  @253ACK000012345;FF
  ```

  ```
  @253NAK;FF
  ```

4 Server Theory of Operation

In this section we'll briefly look at the implementation of the MKS-910 server code. For a more complete description, the reader is urged to consult the Doxygen-generated documentation for the `MKS910Server` class, its helper classes, and the server's source code in the file *mks910Server.cpp*.

4.1 Sensor Naming Convention

Each MKS-910 sensor managed by the server is assigned a short, human-readable text name that is unique across the project. That is to say, if a 910 has been assigned the name "conc01," no other project server can have a sensor or control output by that same name. The server uses these text names internally to identify specific 910s, and these are also the names that client applications use when referencing specific 910's through the server. The individual sensor names are assigned in the server's configuration file, which is more fully described in section 6.

4.2 The Need for Data Caching

Each MKS-910 sensor managed by the server regularly returns at least four bits of information that are read from the sensor itself. These are the current piezo transducer reading, the current pirani transducer reading, the current temperature transducer reading and the pirani gas type calibration currently configured on the sensor. Unfortunately, these data are acquired by querying the sensor over a slow serial connection (9600 baud). If multiple sensors are being queried frequently by multiple clients (which is tyically the case), the number of read requests can easily overwhelm the serial connection's ability to keep up. For this reason, a data caching mechanism is implemented in the server.

At approximately one-second intervals, the server queries all data from all attached sensors and stores the results in a local cache. Then, when clients request sensor data, these cached values are returned. This means the data delivered to clients can be as much as one second old. However, returning data out of the local cache happens very, very quickly, making it possible to service a very large number of client data requests without overwhelming the serial links.

4.3 Threads and Mutexes

The server uses *pthread* library style threads internally, starting one new thread for each MKS-910 sensor the server manages. Each thread is responsible for querying its associated sensor for all of its data at one-second intervals and updating that sensor's local data cache.

These threads need to access data structures (primarily the data cache) and other resources (primarily the serial ports) that the main server application also accesses. Because the threads run concurrently with the main application code, there is the danger that a thread will try to access/modify one of these shared resources at the same time the main code is trying to access/modify the same resource. Allowing that to happen is a recipe for disaster. Instead, pthread library *mutexes*[2] are used to guarantee that these shared resources can only be accessed by one thread at a time. More specific details will be given in sections 4.5.1 and 4.5.2.

4.4 Concentration Calculations

The MKS-910 server provides an interface to multiple MKS-910 sensors that clients can use to read pressure and temperature information. However, the main intent of the server is to provide a mechanism for using off-the-shelf MKS-910 units as an accurate and inexpensive binary gas concentration analyzer. This is possible because the 910's *piezo* transducer is not effected by the chemical composition of the gas mixture it is measuring, but its *pirani* transducer is. This means that for any given pressure, as the ratio of two gases in a binary gas mixture (helium mixed in nitrogen for example) changes, the reported *piezo* pressure does not change, but the reported *pirani* pressure does. If both pressures are known, this effect can be used to determine what the current binary gas concentration must be.

In the current implementation, a very large two-dimensional lookup table has been developed with pirani pressure on one axis and piezo pressure on the other axis. The individual cells of the lookup table give the ratio of the two gases at that particular combination of pressure values. This actually works very well, providing numbers approaching the accuracy of special-purpose binary gas analyzers that are physically much larger and ten times more expensive.

The code that manages the lookup table is implemented as an external module rather than embedding it in the *mks910Server.cpp* source code directly, because there are occasionally

[2]A more in-depth look at the roll of mutexes can be found in the section "Mutexes and Condition Variables" in design document ***DES-0005, The Client/Server Architecture***.

external utilities (for data analysis for example) that also need to link against the lookup code. The lookup module is more fully described in design document ***DES-0059***, ***The MKS-910 Concentration Library***.

4.5 A Layered Implementation

As with most of our servers that control hardware via serial links, the MKS-910 server is implemented in layers based on three primary classes. At the lowest layer, there is the `SerialProtocol` class. This class handles the details of low-level serial communication with the pressure sensors. Next there is the `MKS910Unit` class. This class provides all the functionality to configure and operate a single MKS-910 pressure sensor. Each `MKS910Unit` class maintains a dedicated instance of the `SerialProtocol` class, which it uses to communicate with its associated pressure sensor. Finally there is the `MKS910Server` class. This class is responsible for client/server message handling and high-level interaction with the pressure sensors. It maintains one instance of the `MKS910Unit` class for each sensor the server manages. These are used to query and manipulate the individual sensors. In this section, we'll look at each of these three components in detail. The reader is also urged to consult the Doxygen-generated documentation for these three classes.

4.5.1 The `SerialProtocol` Class

The purpose of the `SerialProtocol` class is to handle the low-level details involved in sending and receiving messages to/from individual MKS-910 pressure sensors over an RS-232 serial connection. The primary documentation for this class is its Doxygen-generated HTML pages. The reader should look there for an in-depth look at the class' capabilities.

The `SerialProtocol` class is responsible for providing access to the serial port associated with a specific pressure sensor. It maintains an internal file descriptor to the serial port through which it communicates. When the class is instantiated, the serial port is opened and configured with appropriate baud rate, word size, start bit, stop bit and parity settings. Conversely, when a `SerialProtocol` object is destroyed, the serial port is automatically closed. All communication with the server's pressure sensors occurs through the `SerialProtocol` class.

Because the serial port is one of those resources that can be accessed both by the sensor's data update thread and the main server application at the same time, it requires mutex protection. The `SerialProtocol` class contains an internal mutex member variable (cleverly named '`mutex`') that is used to protect the serial port. The locking and unlocking of the mutex is handled automatically by methods of the class, so users of the class don't have to worry about it.

The class really only provides one public method – `SendReceiveCommand()`. This is the method that all users of the class use to send commands or queries to the associated sensor and read back the sensor responses. It is important that each thread talking through the serial port have exclusive access to the serial port throughout the entire send/receive transaction. This method guarantees that the class mutex is locked just before sending the command (or query) and not unlocked again until the response has been completely received.

4.5.2 The `MKS910Unit` Class

The `MKS910Unit` class provides the high-level interface to a single MKS-910 pressure sensor. This class is responsible for initializing its associated pressure sensor on initial start up, and it provides methods for carrying out all queries and manipulations on the sensor that are needed by the server or the server's clients. This class also provides the thread function (a static class method called `DataUpdateThread()`) that implements the thread that keeps the sensor's local cache up to date.

Each instance of the `MKS910Unit` class...

- has its own internal `SerialProtocol` object that it uses to communicate with its associated pressure sensor
- provides the storage for the local data cache associated with this sensor
- starts a new copy of the `DataUpdateThread()` method in its own, detached thread. At approximately one-second intervals, this thread queries the attached sensor for its various data and uses the results to update the sensor's local data cache.
- provides an internal mutex used to prevent the data update thread and the main server application code from accessing the local data cache at the same time

Finally, the MKS-910 server's *simulator mode* is also implemented at the `MKS910Unit` class level. In this way, even in simulator mode, all client/server message handling code remains the same and runs the same way as it would if the server were managing real hardware. Also, to the extent possible, calls to `MKS910Unit` methods also run the same code as they would if not in simulator mode, and the data update thread is created and destroyed in the same way in both cases.

Simulator mode is implemented via conditional compilation based on the compiler's pre-processor and a #defined symbol called `SIMULATOR` that is passed in from the compiler command line at compile time. When the code is to be compiled in standard mode, the value of `SIMULATOR` is set to zero. When the code is to be compiled in simulator mode, the value of `SIMULATOR` is set to one[3].

In many places in the server source code, mostly in the implementation of the `MKS910Unit` class, one will find the following pattern:

```
#if SIMULATOR == 0
   // we are in real mode
```

[3]Setting the value of the `SIMULATOR` variable is typically handled indirectly via the project Makefile in response to values set on the command line to the GNU *make* utility. See the comments at the top of the project Makefile for more information.

```
   do real stuff
   ...

#else
   // we are in simulator mode

   do simulated stuff
   ...

#endif
```

While in simulator mode, `MKS910Unit` class' data query method is unaffected. It continues to return values from the class' local cache structure as it would in real mode. *Set* methods (used to set zeros, spans and gas type, for example), on the other hand, are different. They no longer send commands to the real hardware. Instead they simply update the cache parameter values directly (if appropriate) and then exit. The data update thread continues to get created in simulator mode in the usual way, so that portion of the code can be tested, but it doesn't actually send any queries to real hardware.

4.5.3 The `MKS910Server` Class

The `MKS910Server` class, derived from our standard `BaseServer` class[4], is what makes the MKS-910 server application an actual server.

On initial server start up, one instance of the `MKS910Server` class is instantiated. It's constructor processes the server's configuration file (see section 6) and instantiates `MKS910Unit` objects for each MKS-910 defined therein.

The `MKS910Server` class maintains an STL map, called `nameUnitMap`, that maps the short text labels by which individual sensors are known, to the specific `MKS910Unit` objects that interface with those sensors. This map is populated as the server's configuration file is processed, and it is referenced again and again by the various client message handlers to locate a specific sensor's associated `MKS910Unit` object.

5 The Client API

The MKS-910 server, like all servers written for our client/server architecture, relies on a client API class for its client interface. Client API classes provide one public method for each message supported by their corresponding server. These methods handle the details of formatting and sending the request message to the server and receiving, parsing and returning the server's response. This hides all the messy details of client/server message passing from the client.

[4]See design document ***DES-0005, The Client/Server Architecture*** for complete details on the `BaseServer` class and our standard client/server implementation.

The client API class for the MKS-910 server is called `MKS910Client`. It is defined in *mks910ClientLib.h* and implemented in *mks910ClientLib.cpp*. The most important thing to note about the client interface is that, from the client perspective, individual MKS-910 sensors are known by small, human-readable text names (like “dps102” or “conc1”). These text names are assigned in the server’s configuration file, which is described fully in section 6).

5.1 Client-Specific Messages

The MKS-910 server can receive and respond to a number of standard messages. These are fully described in design document ***DES-0005, The Client/Server Architecture***, and will not be further discussed here, except to say that the MKS-910 server fully supports the standard state-of-health reporting mechanism implemented by the `BaseServer` class[5] (more details section 6.2.1).

In addition to these standard messages, the server can accept and respond to a number of client-specific messages that are defined by the client API. These are described in detail in this section[6].

5.1.1 MKS_GET_DATA

This message, implemented by the client class’ `GetData()` method, returns the current pressure, temperature and gas concentration values for the specified MKS-910 sensor. See the Doxygen documentation for the `MKS910DataRecord` structure for more details on what information is returned.

5.1.2 MKS_GET_ALL_DATA

This message, implemented by the client class’ `GetAllData()` method, returns an STL map of `MKS910DataRecord` structures (keyed on sensor name), reporting the current values for all MKS-910 sensors managed by the server.

5.2 MKS_SET_PIEZO_ZERO

This message, implemented by the client class’ `SetPiezoZero()` method, is used to set the value of the low end of the piezo transducer’s range. At the time the message is sent, the pressure sensor is supposed to be at a vacuum of $1 * 10^{-2}$ Torr or better. On receipt of the message, the target sensor adjusts its reported piezo value to zero.

[5] Also see design document ***DES-0006, The State-of-Health Server*** for more information on the standard state-of-health reporting mechanism.

[6] The symbolic constants that comprise the following subsection headers come from the file *mks910ClientLib.h*. Please review the Doxygen documentation for those files for more information.

5.3 MKS_SET_PIEZO_SPAN

This message, implemented by the client class' `SetPiezoSpan()` method, is used to set the value of the upper end of the piezo transducer's range. Unlike the `MKS_SET_PIEZO_ZERO` message, this message takes the pressure applied to the to the sensor at the time the message is sent as a parameter (obviously, this pressure value should come from an independent, calibrated sensor). At the time the message is sent, the pressure should be in the range of 100 to 1000 Torr.

5.4 MKS_SET_PIRANI_ZERO

This message, implemented by the client class' `SetPiraniZero()` method, is used to set the value of the low end of the pirani transducer's range. At the time the message is sent, the pressure sensor is supposed to be at a vacuum of $8 * 10^{-6}$ Torr or better. On receipt of the message, the target sensor adjusts its reported pirani value to zero.

5.5 MKS_SET_PIRANI_SPAN

This message, implemented by the client class' `SetPiraniSpan()` method, is used to set the value of the upper end of the pirani transducer's range. Unlike the `MKS_SET_PIRANI_ZERO` message, this message takes the pressure applied to the to the sensor at the time the message is sent as a parameter (obviously, this pressure value should come from an independent, calibrated sensor). At the time the message is sent, the pressure should be at or near atmospheric pressure (760 Torr).

5.5.1 MKS_SET_TARGET_GAS

This message, implemented by the client class' `SetTargetGas()` method, configures the specified sensor to report pressures from its pirani transducer with the assumption that the gas being measured is of the specified type (nitrogen, helium, etc.). This is necessary because the pirani transducer arrives at its pressure value indirectly as a function of thermal conductivity of the gas being measured. For accurate results, the specific gas being measured must be defined.

5.5.2 MKS_GET_NUM_SENSORS

This message, implemented by the client class' `GetNumberofSensors()` method, returns the total number of MKS-910 sensors being managed by the server.

6 The Server Configuration File

The MKS-910 server uses a configuration file to associate human-readable text names with specific MKS-910 sensors and map them to specific RS-232 serial ports.

6.1 Configuration File Syntax

The server's configuration file is an ASCII text file meant to be hand-edited with a text editor like *gedit*, *emacs* or *vim*. In the text file, blank lines and lines that begin with a "#" character are ignored. All other lines are sensor definition lines. Sensor definition lines are comprised of four fields separated by one or more space characters.

- *serial-port* — The name of the serial port the sensor is connected to
- *name* — A short text label that will be assigned to this sensor. This is the name that clients use when referencing a specific sensor. This field is *not* case sensitive.
- *gas* — This is the name of the type of gas this sensor is expected to measure. Valid values include "nitrogen," "n2," "air," "argon," "ar," "hydrogen," "helium," "he," "h20," and "water." This field is *not* case sensitive.
- *description* — This field contains a free-form text description of the sensor

6.2 Example Configuration File

This section contains a complete example of a typical MKS-910 server configuration file.

```
##########################################################################
#
# Blank lines or lines beginning with a '#' are ignored. All other
# lines are relevant.
#
#-------------------------------------------------------------------------

# serial-port    name    gas  description
/dev/ttyMXUSB2   dps101  air  Initial sample concentration
/dev/ttyMXUSB13  dps102  air  Getter outlet
/dev/ttyMXUSB15  dps200  n2   Detector concentration

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

6.2.1 State-of-Health Reporting

All servers written to our client/server architecture specification have at least the potential to respond to state-of-health requests. By default, such client requests are handled automatically by the underlying `BaseServer` class, which results in an empty list of SoH parameters being returned to the client. However, the MKS-910 server has legitimate SoH data to return for each pressure sensor it manages. This is handled by overriding the three default `BaseServer` SoH message handlers (`GetNumSohParams()`, `GetSohParamInfo()`, and `GetSohParams()`) with versions of our own.

For each sensor managed by the server, we return four pieces of information: the pirani transducer reading, the piezo transducer reading, the computed binary gas concentration (as a percentage) and the temperature transducer reading. These values are reported using the *pseudo-sensor* concept discussed in section 5 of design document ***DES-0005, The Client/Server Architecture***. The names of the four reported pseudo-sensors are generated by appending the text ".pirani," ".piezo," ".conc" or ".temp" to the actual sensor's name.

7 The Test Menu Program

For development and testing purposes, a small text-mode menu client application, called *mks910Menu*, (see *mks910Menu.cpp*) has been developed for the MKS-910 server. The program uses the `MKS910Client` class to provide the client API the server supports. It also maintains a client interface to the system event logger[7], so it can record when it starts up and when it shuts down.

The menu program is meant to be started from within a terminal window. It requires no command line arguments. When the program runs, it presents the user with the following text menu.

```
General Server Items:
 -1 - ping server
 -2 - get server statistics
 -3 - get server message response interval histogram
 -4 - get number of SOH parameters
 -5 - get SOH parameter information
 -6 - get SOH parameters
-99 - shutdown server

MKS910 Server Specific Items:
  1 - Get MKS910 Data
  2 - Get All MKS910 Data
  3 - Set piezo zero point
  4 - Set piezo span
  5 - Set pirani zero point
  6 - Set pirani span
  7 - Set Target Gas
  8 - Get Number of Sensors

  0 - Exit Program

Enter Selection >
```

[7]See design document ***DES-0007, The System Event Logger*** for more information.

The first seven menu items correspond to standard messages that all servers can support[8]. Following this are eight items that correspond to the eight client messages described in section 5. Users choose the number of the message they want to send and are prompted for additional parameters as needed.

The menu program is a full client, supporting every client request message the server is able to process. It allows testers to send each of those messages to the server and view the server's responses. It is intended primarily as a development, testing and debugging tool: however, experience has shown that it is also useful as a bare-bones user interface to the server running on real-world systems.

[8]See design document ***DES-0005*, *The Client/Server Architecture*** for more information on the standard client messages.